等式制約解消系のネットワークによるグローバルコンピューティング

(研究課題番号　12480066)

平成12年度～平成14年度　科学研究費補助金 (基盤研究 (B) (2))

研究成果報告書

平成15年3月

研究代表者　井　田　哲　雄

(筑波大学　電子・情報工学系　教授)

平成 12 年～平成 14 年度　科学研究費補助金　研究成果報告書

等式制約解消系のネットワークによるグローバルコンピューティング

はしがき

２０世紀が終わりに近づいた頃に以下のようなことを考え，科研費補助金の支援を受けて本研究を始めました.『21 世紀の計算環境は地球規模でネットワーク化された計算機群によって形成され，その環境の下でのグローバルコンピューティングが最も重要な情報処理の形態となる．計算サービスは，知識データやメディアの配信という形態にとどまらず，知識データと組み合わされた計算・推論のサービスにまで拡大する．本研究は，ネットワーク化した制約解消系による問題解決システムによるグローバルコンピューティングの実現を図る.』

本研究では，等式の求解を行う制約解消系に関する理論的課題，制約解消系の協調に関する問題，グローバルな計算を行うミドルウェアとの接続の技術的問題など，チャレンジングな課題がたくさんありました．試行錯誤を続けているうちに，３年間が瞬く間にすぎてしまいました．同時に，新しい研究課題が次々にでてきています．上に述べましたことからみますと，まだ道半ばという感じがしておりますが，３年をひとつの区切りとして，本研究報告書にこれまでの研究成果をまとめました．

本研究報告書は，次ページ以降に研究成果の概要を述べ，そしてその後に本研究の成果を発表した論文を収録するという形をとっております．本研究が新しいコンピュータサイエンスの進展に少しでも役に立てばと願っております．なお，本研究には何人もの大学院生が参加しております．その結果，博士取得者一人が生まれ，ポスドク研究者が国際的な研究者へと育っていったことも併せてご報告したいと思います．　　　　　（井田哲雄）

研究成果

次ページ以降に成果概要を述べ，研究論文を収録する．

研究経費

平成 12 年度	6,000	(千円)
平成 13 年度	4,100	(千円)
平成 14 年度	4,200	(千円)
合計	14,300	(千円)

研究組織

研究代表者	井田哲雄	(筑波大学　電子・情報工学系　教授)
研究分担者	ミデルドープ　アート	(筑波大学　電子・情報工学系　助教授)
研究分担者	南出靖彦	(筑波大学　電子・情報工学系　講師)
研究協力者	マリン　ミルチェア	(筑波大学　電子・情報工学系　外国人研究員)

Summary of Research

The advent of open environments such as the Internet, has brought a new and very promising way to design software applications: access to the computing capabilities deployed on the web by various service providers. The Internet can be regarded as the hardware of a computer with practically unlimited computing resources. By open programming we perceive the effort to develop applications which make efficient use of the computing resources of an open environment. Such an effort must address issues like the transparent discovery and access of the services needed by the clients, the implementation of a suitable allocation/deallocation policy of the resources needed to accomplish a certain task, and the coordination of the asynchronous activities of the distributed resources in order to achieve the best performance.

We worked on the design and implementation of an open system for collaborative constraint solving. The outcome of our efforts is a system called **Open CFLP**, which provides support for collaborative constraint functional logic programming in open environments. The system provides support for

(a) higher-order functional logic programming over constraint domains equipped with specialized solvers. Reasoning over functional logic programs is realized by specialized solvers based on higher-order lazy narrowing calculi;

(b) transparent access to specialized constraint solvers via the lookup service of a specialized broker;

(c) a collaboration language which enables the user to specify the most common ways in which constraint solvers should collaborate to achieve the desired results.

For (a), we have designed and implemented in *Mathematica* various refinements of higher-order lazy narrowing which are relevant for programming purposes. These refinements are very important because they reduce the huge search space for solutions. In [19, 1, 4, 14] we proposed various refinements for higher-order pattern rewrite systems, and in [10, 2] we have shown how these refinements can be employed in the implementation of a constraint functional logic programming system called **CFLP**.

Although the class of pattern rewrite systems is already very expressive for functional logic programming, it is desirable to allow conditional rewrite rules as well. The effort to identify suitable extensions to the conditional case is still a hot topic in the functional logic community. We succeeded to identify a subclass of conditional rewrite systems which we called deterministic conditional rewrite systems, which seems to be very suitable for programming purposes, and for which we can refine lazy narrowing to explore a much smaller search space for solutions [16, 20]. It turns out that these refinements can be combined with the refinements proposed for pattern rewrite systems to yield sound and complete computational models for functional logic programming with conditional pattern rewrite systems.

For (b), we have designed and implemented a lookup service that allows transparent access to constraint solvers advertised by providers in an open environment such as the Internet. The user asks for a solver characterized by a certain interface and attributes (e.g., constraint solving domain, solving method, etc.) and the lookup service yields a proxy to such a remote solver (if available).

A related issue is support for solver inter-operability. It is reasonable to assume that constraint solvers are heterogeneous. Therefore, we provide support for inter-operability by designing a global language and ontology for communication. The individual solvers are wrapped into objects equipped with translators (codecs) between the global language and their specific language.

We have chosen a global language based the *MathML* vocabulary of XML. The communication protocol is based on CORBA, and transmits *MathML* messages encoded into valuetype objects of XML DOM.

The support for collaboration (c) is motivated by the widely accepted fact that the design and implementation of a general-purpose constraint solving system is not a reasonable task. Instead, it has been recognized that the task of solving complex problems can be carried out by a collaboration of specialized constraint solvers. Since there is a practically unlimited number of specialized constraint solving methods, the open environment provides the best framework for the design os a collaborative constraint solving system: whenever the user needs to apply a specialized constraint solving method, it looks up the open environment for a service that implements that method. The lookup service of **Open CFLP** provides the user with a so called elementary solver which is a proxy to the constraint solving service requested by the user. Thus, each elementary solver can be regarded as a black box which encapsulates a specialized constraint solving method. The elementary solvers can be put to work together by building solver collaborations. The best way to build solver collaborations is by providing the user with a programming language for solver collaborations.

We have designed and implemented a solver collaboration language which provides primitives to describe the most common ways to combine component solvers into a higher-level constraint solving procedure.

Formally, a collaborative acts on a constraint store, i.e. a set $\{c_1, \ldots, c_n\}$ of elementary constraints (e.g., equations or inequalities). Such a store denotes the logical conjunction $G = c_1 \wedge \ldots \wedge c_n$. The outcome of applying *Coll* on a store G is disjunction of formulas $\exists X_1.G_1 \vee \ldots \vee \exists X_m.G_m$ where $G_1, \ldots, G_m$ are new constraint stores which are existentially quantified over the sets of variables $X_1, \ldots, X_m$ respectively. Elementary solvers must be sound (i.e., $sol(\exists X_j.G_j) \subseteq sol(G)$ for all j, where $sol(F)$ is the set of solutions of a formula F) and complete (i.e., any solution of G is a solution of some formula $\exists X_j.G_j$). Intuitively, the result of applying a collaborative *Coll* to G is a disjunction of formulas which are *simpler*, in the sense that they are more explicit in expressing their set of solutions.

The meaning of our syntactic constructs for solver collaboration is described in [22]. Collaborations are assumed to be closed under logical disjunction, that is, $Coll(G_1 \vee \ldots \vee G_n) = Coll(G_1) \vee \ldots \vee Coll(G_n)$. It can be shown that soundness and completeness are preserved by our collaboration primitives.

Since logical disjunctions can be handled independently, we can boost the performance of our system by solving disjoint constraint stores concurrently. We have implemented in *Java* a multi-threaded interpreter for collaboratives expressed in such a coordination language. Whenever the interpreter encounters subcomputations which can be performed concurrently (such as disjunctive constraints), it starts new threads and dispatches these subcomputations to separate agents which run asynchronously on (possibly different) computers of the open environment.

References

[1] M. Marin and T. Ida. Cooperative Constraint Functional Logic Programming. In *9th International Workshop on Functional and Logic Programming WFLP'2000*, pages 382–390, Benicassim, Spain, 2000.

[2] M. Marin and T. Ida. Higher-order Lazy Narrowing in Perspective. In *9th International Workshop on Functional and Logic Programming WFLP'2000*, pages 238–253, Benicassim, Spain, 2000.

[3] M. Marin, T. Ida, and T. Suzuki. Cooperative Constraint Functional Logic Programming. In T. Katayama, T. Tamai, and N. Yonezaki, editors, *International Symposium on Principles of Software Evolution (ISPSE 2000)*, pages 223–230, Kanazawa, Japan, November 1-2 2000. IEEE.

[4] M. Marin, T. Ida, and T. Suzuki. Lazy Narrowing Calculi for Pattern Rewrite Systems. In *Proceedings of Second International Workshop on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2000)*, Timişoara, Romania, October 4-6 2000.

[5] Y. Minamide. A New Criterion for Safe Program Transformations. In *Proceedings of the Forth International Workshop on Higher Order Operational Techniques in Semantics (HOOTS)*, ENTCS.

[6] Q. Li, Y.-K. Guo, T. Ida, and J. Darlington. Minimized Geometric Buchberger Algorithm for Integer Programming. *Annals of Operations Research*, 108:87–109, January 2001.

[7] T. Ida, M. Marin, and T. Suzuki. Higher-order lazy narrowing calculus: a solver for higher-order equations. In *Proceedings of the Eight International Conference on Computer Aided Systems. (EUROCAST 2001)*, LNCS 2178, pages 19–23, Las Palmas de Gran Canaria, Canary Islands, Spain, February 2001.

[8] T. Ida, N. Kobayashi, and M. Marin. An Open Environment for Cooperative Scientific Problem Solving. In *Fourth International Mathematical Symposium (IMS'2001)*, Chiba, Japan, June 25-27 2001.

[9] T. Ida, M. Marin, and N. Kobayashi. An open environment for cooperative equational solving. *Wuhan University Journal of Natural Science*, 6(1-2):169–174, 2001.

[10] A. Marin, T. Ida, and W. Schreiner. CFLP: A Mathematica Implementation of a Distributed Constraint Solving System. *The Mathematica Journal*, 8(2):287–300, 2001.

[11] Y. Minamide. Runtime Behavior of Conversion Interpretation of Subtyping. In *Proceedings of the 13th International Workshop on Implementation of Functional Languages*, LNCS.

[12] T. Suzuki and A. Middeldorp. A Complete Selection Function for Lazy Conditional Narrowing. In *Proceedings of the 5th Symposium on Functional and Logic Programming (FLOPS 2001)*, LNCS 2024, pages 201–215, Tokyo, 2001.

[13] T. Yamada. Confluence and Termination of Simply Typed Term Rewriting Systems. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA 2001)*, LNCS 2051, pages 338–352, Utrecht, 2001.

[14] M. Marin, T. Suzuki, and T. Ida. Refinements of Lazy Narrowing for Left-Linear Fully-extened Pattern Rewrite Systems. Technical Report ISE-TR-01-180, Institute of Information Sciences and Electronics, University of Tsukuba, Japan, 2001. Also available from `http://www.score.is.tsukuba.ac.jp/~mmarin/vita/vita_10.html`.

[15] I. Durand and A. Middeldorp. On the Modularity of Deciding Call-by-Need. In *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, LNCS 2030, pages 199–213, Genova, 2002.

[16] M. Marin and A. Middeldorp. New Completeness Results for Lazy Conditional Narrowing. In *6th International Workshop on Unification (UNIF 2002)*, Copenhagen, Denmark, July 22-26 2002.

[17] N. Kobayashi, M. Marin, T. Ida, and Z. Che. An Open Environment for Collaborative Constraint Functional Logic Programming. In *11th International Workshop on Functional and (Constraint) Logic Programming (WFLP2002)*, Grado, Italy, June 2002.

[18] N. Kobayashi, M. Marin, Z. Che, and T. Ida. Open CFLP: An Open System for Collaborative Constraint Functional Logic Programming. In *8th Intl. Conf. on Applications of Computer Algebra (ACA 2002)*, Volos, Greece, June 25-28 2002.

[19] T. Ida, M. Marin, and T. Suzuki. Reducing Search Space in Solving Higher-Order Equations. In S. Arikawa and A. Shinohara, editors, *Progress in Discovery Science, Final Report of the Japanese Discovery Science Project*, volume 2281 of *LNAI*, pages 19–30. Springer, 2002.

[20] M. Marin. A Deterministic Conditional Lazy Narrowing Calculus. In *Proceedings of 4th International Workshop on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2002)*, Timişoara, Romania, October 9-12 2002.

[21] T. Ida. Equational Reasoning in Programming. In *Proceedings of the 7th Asian Technology Conference in Mathematics*, pages 22–34, December 2002.

[22] N. Kobayashi, M. Marin, and T. Ida. Collaborative constraint functional logic programming system in an open environment. *IEICE Transactions on Information and Systems*, E86-D(1):63–70, January 2003.

Cooperative Constraint Functional Logic Programming

Mircea Marin, Tetsuo Ida
Institute of Information Sciences and Electronics
University of Tsukuba,
1-1-1 Tennoudai, Tsukuba, Ibaraki 305-8573, Japan
mmarin@score.is.tsukuba.ac.jp, ida@score.is.tsukuba.ac.jp

Taro Suzuki
Research Institute of Electric Communication,
Tohoku University, 980-8577 Sendai,Japan
taro@nue.riec.tohoku.ac.jp

Abstract

We describe the current status of the development of CFLP, *a system which aims at the integration of the best features of functional logic programming (FLP), cooperative constraint solving (CCS), and distributed constraint solving. FLP provides support for defining one's own abstractions (user-defined functions and predicates) over a constraint domain in an easy and comfortable way, whereas CCS is employed to solve systems of mixed constraints by iterating specialized constraint solving methods in accordance with a well defined strategy.*

CFLP *is a distributed implementation of a cooperative constraint functional logic programming scheme obtained from the integration of higher-order lazy narrowing for functional logic programming with cooperative constraint solving. The implementation takes advantage of the existence of several constraint solving resources located in a distributed environment, which communicate asynchronously via message passing.*

1 Introduction

Integration of declarative programming paradigms into a unified framework that captures the best features of each of them has attracted much interest during the last decade. Of particular interest is the design and implementation of a system based on a clean combination of functional logic programming (FLP) with cooperative constraint solving (CCS).

CFLP [4] is an experimental system which integrates higher-order functional logic programming with cooperative constraint solving. Its computational model combines higher-order lazy narrowing with the principle of solving systems of mixed constraints with a cooperation of constraint solvers described by a given cooperation strategy. The system is implemented in *Mathematica* and consists of an interpreter based on a higher-order lazy narrowing calculus, and a cooperative constraint solving system.

Our paper is structured as follows. In Section 2 we illustrate the solving capabilities of CFLP. The language of CFLP—syntax and semantics—is outlined in Section 3. Section 4 describes the general architecture of the system and of its main components. Finally, in Section 5 we draw some conclusions and directions of further research.

2 Examples

We describe with two examples the solving capabilities of CFLP.

2.1 Electric Circuit Modeling

The first example shows how electric circuit layouts can be computed with CFLP. This example illustrates the expressive power of the FLP style extended with higher-order constructs such as λ-abstractions and function variables, and constraint solving capabilities for differential equations and systems of polynomial equations.

We first define a function spec which describes the behavior in time t of an electrical component as a function of the current $i[t]$ and voltage $v[t]$ in the circuit. The CFLP rules of spec correspond to a recursive definition, where the base case describes the behavior of elementary circuits such as resistors, capacitors, and inductors, and the inductive case describes the behavior of serial and parallel connections of electrical components.

The CFLP program is given below. We do not explain the underlying electronic laws since it should be easy to read them off from the program.

```
(* declare the CFLP type ElComp with
  associated data constructors
  res, ind, cap, serial, par *)
Constructor[ElComp = res[Float]
                   | ind[Float]
                   | cap[Float]
                   | serial[TyList[ElComp]]
                   | par[TyList[ElComp]]];
(* specify the CFLP program *)
Prog:= {
 spec[res[r],v:Float → Float,i] → True
     ⇐ λ[{t},v[t]] ≈ λ[{t},r * i[t]],
 spec[ind[l],v,i:Float → Float] → True
     ⇐ λ[{t},v[t]] ≈ λ[{t},l*i'[t]],
 spec[cap[c],v:Float → Float,i] → True
     ⇐ λ[{t},i[t]] ≈ λ[{t},c*v'[t]],
 spec[serial[{}],λ[{t},0],i] → True,
 spec[serial[[comp | comps]],v,i] → True ⇐
     {spec[comp,v1,i] ≈ True,
     spec[serial[comps],v2,i] ≈ True,
     λ[{t},v[t]] ≈ λ[{t},v1[t] + v2[t]]},
 spec[par[{}],v:Float → Float,λ[{t},0]] → True,
 spec[par[[comp | comps]],v,i] → True ⇐
     {spec[comp,v,i1] ≈ True,
     spec[par[comps],v,i2] ≈ True,
     λ[{t},i[t]] ≈ λ[{t},i1[t] + i2[t]]}}
```

The variables and operators can be type annotated, and a polymorphic type checker is integrated into the system to verify the type consistency of the CFLP goal and program. The universally quantified variables are underlined. Note the usage of the list construct in the recursive specification of serial and parallel connections of electrical components. The CFLP system recognizes the following list specifications:

- $\{t_1, \ldots, t_n\}$: the list consisting of components $t_1, \ldots, t_n$.
- $[\, h \mid tl \,]$: CFLP list with head h and tail tl in Prolog-style notation.

Consider the problem of finding the behavior in time of the current in a RLC circuit with R $= 2$, an L $= 1$ and $C = 1/2$ (see Fig. 1), under the restrictions that the voltage is constant in time and the current was initially set to 0.

In this case, the goal which we want to solve is

```
G := {spec[serial[{res[2], ind[1], cap[1/2]}],
     λ[{t},50],ī] ≈ True,
     i[0] ≈ 0}
```

The logical variable of the goal is i, and it is overlined. Note

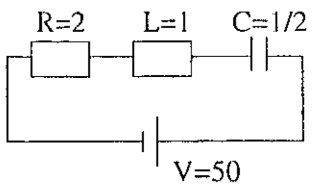


Fig. 1. RLC circuit

the usage of the expression λ[{t}, 50] for specifying that the voltage is constant (50 Ω) in time.

Now we can ask the CFLP system to solve the problem:

```
TSolve[G,Rules → Prog]
```

This call is similar to the Solve call of *Mathematica*, but TSolve has the specific option Rules which provides the functional logic program (i.e., conditional rewrite system) used upon solving the goal G.

The system computes the parametric solution $\{i \mapsto \lambda t.-k\,e^{-t}\sin(t)\}$ which is represented in *Mathematica* by {i → λ[{t}, - k e^{-t} Sin[t]}.

2.2 Program Calculation

This example illustrates the capabilities of a computing environment which can perform full higher-order pattern unification, and higher-order term rewriting.

Consider the problem of writing a program to check whether a list of numbers is *steep*, i.e, if every element of the list is greater than or equal to the average of the elements that follow it. A CFLP program that does this is:

```
Prog::={
  steep[{}] → True,
  steep[[a | x]] → And[a * len[x] ≥ sum[x], steep[x]],
  sum[{}] → 0, sum[[x | y]] → x + sum[y],
  len[{}] → 0, len[[x | y]] → 1 + len[y]}
```

Prog is modular and easy to understand, but very inefficient (quadratic complexity). It is possible—and desirable—to automatically compute an efficient (linear complexity) version steepOptimal of the function steep defined in Prog. To achieve this, we employ the fusion calculational rule

$$\frac{f[c] = e' \quad f[g[\underline{a}, \underline{ns}]] = h[\underline{a}, f[\underline{ns}]]}{f[\mathtt{foldr}[g, e, \underline{ns}]] = \mathtt{foldr}[h, e', \underline{ns}]} \qquad (1)$$

where $\mathtt{foldr} : (\alpha \times \beta \to \beta) \times \beta \times \mathtt{TyList}[\alpha] \to \beta$ is defined as usual:

$$\begin{aligned} \mathtt{foldr}[\underline{g}, \underline{e}, \{\}] &= \underline{e}, \\ \mathtt{foldr}[\underline{g}, \underline{e}, [\underline{n} \mid \underline{ns}]] &= \underline{g}[\underline{n}, \mathtt{foldr}[\underline{g}, \underline{e}, \underline{ns}]]. \end{aligned}$$

To see how this calculational rule can be employed, we first observe that the computation of steep$[[n \mid ns]]$ with

Prog requires the computation of 3 additional quantities: steep[ns], sum[ns] and length[ns]. Thus, Prog actually specifies the computation of the function

$$\texttt{f} = \lambda[\{ns\}, \texttt{c3}[\texttt{steep}[ns], \texttt{sum}[ns], \texttt{length}[ns]]]$$

where c3 is a ternary constructor.

Note that f[$\underline{ns}$] = f[foldr[$g, e, \underline{ns}$]], where g = Cons and $e = \{\}$. From the fusion calculational rule (1) results that we can compute f[ns] efficiently by computing foldr[h, e', ns] instead, where h is the solution of the equation

$$\lambda[\{n, ns\}, \texttt{f}[g[n, ns]]] \approx \lambda[\{n, ns\}, \overline{\texttt{h}}[n, f[ns]]]. \quad (2)$$

and $e' = f[\{\}] = \texttt{c3}[\texttt{True}, 0, 0]$. Then:

$$\texttt{steepOpt} = \lambda[\{ns\}, \texttt{sel-c3-1}[\texttt{foldr}[\texttt{h}, e', ns]]] \quad (3)$$

where sel-c3-1 is the data selector of the first argument of a term constructed with c3 (see Subsect. 3.1 for details). To solve equation (2) with CFLP we perform the following calls:

```
(* Declare data constructor c3 with
 associated type constructor 'Triple' *)
Constructor[Triple=c3[Bool,Float,Float]];

(* add definition of f to Prog *)
AppendTo[Prog, f[ns] → c3[steep[ns], sum[ns], len[ns]]];

(* compute h; during the computation
  'Plus','Times','Power','GreaterEqual','AND'
  are regarded as mere constructors *)
TSolve[λ[{n, ns}, f[[n | ns]]] ≈
      λ[{n, ns}, h̄[n, c3[steep[ns], sum[ns], len[ns]]]],
  Constructor →{Plus, Power,
                  Times, GreaterEqual, And},
  Rules → Prog]
```

CFLP yields the unique solution

```
{{h↦ λ[{x$13, x$14}, c3[
    And[x$13 sel-c3-3[x$14] ≥ sel-c3-2[x$14]],
    sel-c3-1[x$14],
    x$13 + sel-c3-1[x$14], 1 + sel-c3-3[x$14]]]}}
```

3 The Language

CFLP is a distributed software system for solving systems of equations over various constraint domains for which specialized constraint solvers are available.

3.1 Syntax

The language of CFLP is built up from the following symbols:

- Built-in constant symbols: 2/3, -2912, Pi, etc.[1],
- External symbols: the elements of a predefined set $\mathcal{F}_e = \{+, -, *, \ldots\}$,
- Built-in relational symbols:

  equational: $\approx$ (unoriented equality), $\rhd$ (oriented equality), $\doteq$ (unoriented strict equality) and $\gg$ (oriented strict equality)

  logical: $\{\cdot, \cdots\}$ (sequential AND), $\|$ (sequential OR), and $\vee$ (parallel OR),
- Built-in type constructors:
  - base types: Int, Float, Compl, Bool,
  - TyList[α] (polymorphic list type) and $\{\}$ (empty list constant),
- Data constructors: the elements of a set $\mathcal{F}_c$ which contains the built-in data constructors Cons, $\{\}$ (for lists), and the data constructors defined with Constructor declarations,
- Defined function symbols: symbols of a set $\mathcal{F}_d$ which contains:
  - the data selectors introduced by the Constructor declarations,
  - the symbols defined by the rewrite rules of a given program
- Variables: symbols of a set $\mathcal{V}$ of *Mathematica* symbols with no predefined or given meaning.

The set $\mathcal{F} = \mathcal{F}_e \cup \mathcal{F}_c \cup \mathcal{F}_d$ is called the *signature* of CFLP. Next, we build the other syntactic objects of our language:

- Well-typed λ-term (or term): *Mathematica* representation of an expression built over the signature $\mathcal{F}$ and set of variables $\mathcal{V}$, but instead of Function we use λ. E.g., $\lambda[\{x, y\}, x * y]$ is the λ-term for the function which computes the product of its arguments. The set of terms is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$.
- Type-annotated symbol: an expression of the form $v : \tau$ where v is a variable or function symbol and τ is a type expression.
- Equation: an expression of the form $s \cong t$ where $\cong \in \{\approx, \rhd, \doteq, \gg\}$ and s, t are terms of the same type.
- Goal: either an equation, or of the form $\{g_1, \ldots, g_n\}$ (sequential AND goal), $g_1 \| \ldots \| g_n$ (sequential OR goal), or $g_1 \vee \ldots \vee g_n$ (parallel OR-goal), where $g_1, \ldots, g_n$ are goals.

[1]These are the constants recognized by *Mathematica*

- Rewrite rule: expression of the form $f[l_1, \ldots, l_n] \to r \Leftarrow c$ where $f[l_1, \ldots, l_n]$, r are terms of the same base type, $f[l_1, \ldots, l_n]$ is a higher-order pattern, and c is a goal. f is the *defined* function, and c the *condition* of the rewrite rule.

- Program: a (possibly empty) set of rewrite rules.

A `Constructor` declaration has the form

`Constructor[`$\langle$*constr-spec*$_1\rangle, \ldots, \langle$*constr-spec*$_n\rangle$`]`

where

$\langle$*constr-spec*$\rangle$ ::= $\langle$*type-spec*$\rangle$"="$\langle$*data-spec*$_1\rangle$"|" ... "|"$\langle$*data-spec*$_n\rangle$
$\langle$*type-spec*$\rangle$ ::= $\langle$*type-name*$\rangle$ | $\langle$*type-name*$\rangle[\alpha_1, \ldots, \alpha_n]$
$\langle$*data-spec*$\rangle$ ::= $\langle$*type-constr*$\rangle$ | $\langle$*type-constr*$\rangle[\tau_1, \ldots, \tau_n]$
$\alpha_1, \ldots, \alpha_n$ distinct type variables
$\tau_1, \ldots, \tau_n$ type expressions

Example 1 We can declare binary trees with nodes of type α by:

```
Constructor[BTree[α] =
                BNil |
                T2[α,BTree[α],BTree[α]]
```

This declaration introduces the type constructor `BTree` together with its associated data constructors

`BNil`:$\forall\alpha.$`BTree[`α`]`
`T2` : $\forall\alpha.\alpha \times$ `BTree`$[\alpha] \times$ `BTree`$[\alpha] \to$ `BTree`$[\alpha]$.

In addition, the data-selectors `sel-T2-1`, `sel-T2-2` and `sel-T2-3` for the data constructor `T2` are defined implicitly. □

A `Constructor` call extends the set $\mathcal{F}_c$ of the language of CFLP with the newly declared data constructors, and the set $\mathcal{F}_d$ of defined symbols of the language of CFLP with the selectors of the non-constant constructors. For example, if c is a newly-declared n-ary data constructor, then the data selectors `sel-`c`-1`, ..., `sel-`c`-`n are added to $\mathcal{F}_d$.

A program $\mathcal{R}$ adds the set of symbols

$$\{f \mid \exists(f[l_1, \ldots, l_n] \to r \Leftarrow c) \in \mathcal{R}\}$$

to the set $\mathcal{F}_d$ of defined symbols of the CFLP language. We denote by $Subst(\mathcal{F}, \mathcal{V})$ the set of substitutions over a signature $\mathcal{F}$ and set of variables $\mathcal{V}$, by $\mathcal{D}(\theta)$ the domain of a substitution θ, and by ε the empty substitution.

3.2 Semantics

The semantics of the external symbols in $\mathcal{F}_e$ and of the data constructors in $\mathcal{F}_c$ is given by a predefined constraint domain $\mathcal{X}$ equipped with various solvers for solving systems of equational constraints.

A CFLP program $\mathcal{R}$ extends the language of the constraint domain $\mathcal{X}$ with symbols defined in a functional-logic programming style, and to give them a meaning. A program $\mathcal{R}$ induces a rewrite relation $\longrightarrow_{\mathcal{X},\mathcal{R}}$ on $\mathcal{X}$, where pattern matching is defined modulo the equality relation on $\mathcal{X}$. We define the semantics of the equational symbols of CFLP as follows:

- $\mathcal{X}, \mathcal{R} \models s \approx t$ if $s \longrightarrow^*_{\mathcal{X},\mathcal{R}} u \ {}^*_{\mathcal{X},\mathcal{R}}\!\longleftarrow t$ for some term u,

- $\mathcal{X}, \mathcal{R} \models s \rhd t$ if $s \longrightarrow^*_{\mathcal{X},\mathcal{R}} t$,

- $\mathcal{X}, \mathcal{R} \models s \doteq t$ if $s \longrightarrow^*_{\mathcal{X},\mathcal{R}} u \ {}^*_{\mathcal{X},\mathcal{R}}\!\longleftarrow t$ for some constructor term u,

- $\mathcal{X}, \mathcal{R} \models s \gg t$ if t is a constructor term and $s \longrightarrow^*_{\mathcal{X},\mathcal{R}} t$.

3.3 The Problem

CFLP is designed to solve problems of the following type:

Given a program $\mathcal{R}$ and an equation $s \cong t$ with $\cong \in \{\approx, \rhd, \doteq, \gg\}$,

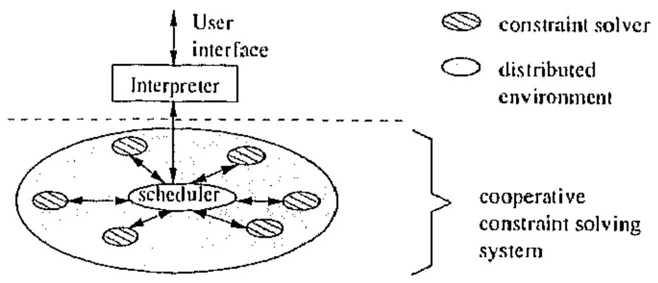
Compute $\theta \in Subst(\mathcal{F}, \mathcal{V})$ such that $\mathcal{X}, \mathcal{R} \models s\theta \cong t\theta$.

We call such a θ an $\mathcal{R}$-*solution* of $s \cong t$. This problem is extended to goals in the natural way. We denote by $\mathcal{U}_\mathcal{R}(G)$ the set of $\mathcal{R}$-solutions of a goal G. In general, we are interested to compute $\mathcal{R}$-*normalized* solutions of a goal G, i.e. substitutions $\theta \in \mathcal{U}_\mathcal{R}(G)$ such that $X\theta$ is $\mathcal{R}$-normalized for all $X \in \mathcal{D}(\theta)$. We denote the set of $\mathcal{R}$-normalized solutions of G by $\mathcal{U}^n_\mathcal{R}(G)$.

To solve such problems, we have designed and implemented a computation model which integrates two operational principles: lazy narrowing for conditional pattern rewrite systems, and concurrent constraint solving. Lazy narrowing solves equations containing symbols from $\mathcal{F}_c \cup \mathcal{F}_d$, whereas concurrent constraint solving is employed to solve systems of equations over $\mathcal{X}$.

4 System Structure

The combination of lazy narrowing with CCS is reflected in the structure of the system: it consists of a functional logic interpreter based on lazy narrowing, which is integrated with a distributed implementation of a cooperative constraint solving system (see below).



4.1 The Interpreter

The interpreter of CFLP is based on a calculus $\mathcal{C}$ which is an extension of a pure lazy narrowing calculus $\mathcal{K}$ for conditional pattern rewrite systems with inference rules to recognize and process equations with external symbols. $\mathcal{C}$ can be described by a set of inference rules which act on *states* of the form $\langle W \mid G \mid \mathit{Store}\rangle$ where W is a set of variables, G is a CFLP goal, and *Store* is a set of constraints $s \approx t$ collected so far. A state $\langle W \mid G \mid \mathit{Store}\rangle$ is interpreted as

$$[\![\langle W \mid G \mid \mathit{Store}\rangle]\!] = \{\gamma \mid \gamma \in \mathcal{U}_{\mathcal{R}}(G), \gamma\restriction_W \text{ is } \mathcal{R}\text{-normalized}, \text{ and } \gamma \in \mathcal{U}_{\mathcal{R}}(e) \text{ for all } e \in \mathit{Store}\}.$$

The inference rules of the calculus $\mathcal{C}$ are presented as relations of the form

$$\langle W \mid G \mid \mathit{Store}\rangle \overset{\mathcal{C}}{\Longrightarrow}_{\theta} \langle W' \mid G' \mid \mathit{Store}'\rangle$$

where θ is a substitution, called *computed* substitution. Such a relation is called $\mathcal{C}$*-step*. The interpreter computes $\mathcal{C}$*-derivations*, i.e. sequences

$$\langle W_0 \mid G_0 \mid \mathit{Store}_0\rangle \overset{\mathcal{C}}{\Longrightarrow}_{\theta_1} \ldots \overset{\mathcal{C}}{\Longrightarrow}_{\theta_N} \langle W_N \mid G_N \mid \mathit{Store}_N\rangle$$

of $\mathcal{C}$-steps, abbreviated

$$\langle W_0 \mid G \mid \mathit{Store}\rangle \overset{\mathcal{C}}{\Longrightarrow}^{*}_{\theta_1 \ldots \theta_N} \langle W_N \mid G_N \mid \mathit{Store}_N\rangle.$$

The $\mathcal{C}$-derivations which are useful in computing a representation of $\mathcal{U}^{n}_{\mathcal{R}}(G)$ are the so-called $\mathcal{C}$-refutations. A $\mathcal{C}$-*refutation* is a finite $\mathcal{C}$-derivation of maximum length

$$\langle \mathcal{V}(G) \mid G \mid \{\}\rangle \overset{\mathcal{C}}{\Longrightarrow}^{*}_{\theta} \langle W' \mid G' \mid \mathit{Store}\rangle \qquad (4)$$

where $\mathcal{V}(G)$ is the set of free variables in G. The set of answers computed by CFLP for a given goal G is

$$\mathit{Answ}_{\mathcal{R}}(G) = \{\ \langle \theta, G', \mathit{Store}\rangle \mid \exists\, \mathcal{C}\text{-refutation } \langle \mathcal{V}(G) \mid G \mid \{\}\rangle \overset{\mathcal{C}}{\Longrightarrow}^{*}_{\theta} \langle W' \mid G' \mid \mathit{Store}\rangle\}$$

The calculus $\mathcal{C}$ is designed to satisfy the following two conditions:

soundness: For any $\langle \theta, G', \mathit{Store}\rangle \in \mathit{Answ}_{\mathcal{R}}(G)$ and any $\mathcal{R}$-solution γ of G' and *Store*, we have $\theta\gamma \in \mathcal{U}_{\mathcal{R}}(G)$

completeness: For any $\gamma \in \mathcal{U}^{n}_{\mathcal{R}}(G)$ there is a $\langle \theta, G', \mathit{Store}\rangle \in \mathit{Answ}_{\mathcal{R}}(G)$ such that $\gamma = \theta\gamma'\ [\mathcal{V}(G)]$ for some $\mathcal{R}$-solution γ' of G' and *Store*.

A $\mathcal{C}$-refutation is constructed in two stages:

1. *Lazy Narrowing Stage:* starting with the state $\langle \mathcal{V}(G) \mid G \mid \{\}\rangle$, the goal G is narrowed until a goal made of equations which can not be narrowed anymore. The equations which can not be narrowed anymore are either:

   - *constraints*, i.e. equations over the constraint domain $\mathcal{X}$, or
   - certain *flex/flex equations*, i.e. equations between terms of the form

     $$\lambda[\{x_1, \ldots, x_p\}, X[s_1, \ldots, s_m]]$$

     with X a free variable.

     Remark: The design a calculus to solve all flex/flex equations is unrealistic, since full higher-order unification is highly intractable [1]. Our design of $\mathcal{C}$ is based on Huèt's idea of pre-unification [3].

2. *Constraint Solving Stage:* the constraints produced during the lazy narrowing stage are solved with a cooperative constraint solver.

4.1.1 The Lazy Narrowing Stage

The rules of $\mathcal{C}$ which realize this stage are:

[V] *parallel OR*
$\langle W \mid G_1 \vee \ldots \vee G_n \mid \mathit{Store}\rangle \overset{\mathcal{C}}{\Longrightarrow}_{\epsilon} \langle W \mid G_k \mid \mathit{Store}\rangle$ where $k \in \{1, \ldots, n\}$

[||] *sequential OR*
$\langle W \mid G_1 \| \ldots \| G_n \mid \mathit{Store}\rangle \overset{\mathcal{C}}{\Longrightarrow}_{\epsilon} \langle W \mid G_k \mid \mathit{Store}\rangle$ where $k \in \{1, \ldots, n\}$

The difference between these rules is that the nondeterminism due to selection of G_k is explored by breadth-first search for [V] and by depth-first search for [||].

[{·}] *sequential AND*
$\langle W \mid \{G_1, \ldots, G_n\} \mid \mathit{Store}\rangle \overset{\mathcal{C}}{\Longrightarrow}_{\theta} \langle W' \mid \{G'_1, \ldots, G'_n\} \mid \mathit{Store}'\rangle$ if $\langle W \mid G_k \mid \mathit{Store}\rangle \overset{\mathcal{C}}{\Longrightarrow}_{\theta} \langle W' \mid G'_k \mid \mathit{Store}'\rangle$ and $G'_i = G_i\theta$ for all $i \neq k$.

[xf] *constraint accumulation*
$\langle W \mid s \cong t \mid \mathit{Store}\rangle \overset{\mathcal{C}}{\Longrightarrow}_{\epsilon} \langle W \mid \{\} \mid \mathit{Store} \cup \{s \approx t\}\rangle$ if $s \approx t$ is a constraint

[xi] *imitation for external symbols*

$$\frac{\langle W \mid \lambda[\{\overline{x_p}\}, f[\overline{s_m}] \cong \lambda[\{\overline{x_p}\}, t] \mid Store\rangle \overset{\mathcal{C}}{\Longrightarrow}_{\epsilon}}{\langle W \mid \lambda[\{\overline{x_p}\}, s_m] \cong \lambda[\{\overline{x_p}\}, Y_m[\overline{x_p}]], \lambda[\{\overline{x_p}\}, Y[\overline{x_p}]] \cong \lambda[\{\overline{x_p}\}, t] \mid Store\rangle}$$

with $Y_1, \ldots, Y_m$ fresh variables and $\cong \in \{r, r^{-1} \mid r \in \{\approx, \doteq, \rhd, \gg\}\}$ if $f \in \mathcal{F}_e$.

[flp] *lazy narrowing step*

These steps are governed by a given lazy narrowing calculus $\mathcal{K}$

$\langle W \mid s \cong t \mid Store\rangle \overset{\mathcal{C}}{\Longrightarrow}_{\theta} \langle W \mid G \mid Store'\rangle$
if $s \cong t \overset{\mathcal{K}}{\Longrightarrow}_{\theta} G'$ is a $\mathcal{K}$-step and $Store' = \{s\theta \approx t\theta \mid (s \approx t) \in Store\}$.

The interpreter of CFLP can make use of different built-in lazy narrowing calculi $\mathcal{K}$, depending on the preference of the user. We have designed and implemented sound and complete lazy narrowing calculi for pattern rewrite systems (PRS for short), left-linear confluent fully-extended PRS, and left-linear constructor fully extended PRS (see [5]). Moreover, we have extended these calculi to handle strict equations and conditional PRSs. Some of these calculi are higher-order generalizations of the deterministic refinements of the lazy calculus LNC [7, 6].

4.1.2 The Constraint Solving Stage

The constraints accumulated in the constraint store during the lazy narrowing stage are submitted to be solved with a cooperative constraint solving system. Formally, this stage can be described by

[cs] *constraint solve*

$\langle W \mid G \mid Store\rangle \overset{\mathcal{C}}{\Longrightarrow}_{\theta} \langle W' \mid G\theta \mid Store'\rangle$
if $\langle \theta, Store'\rangle \in \mathcal{S}(\langle\{\}, Store\rangle)$ (see next subsection) and W' are the free variables in $\{X\theta \mid X \in W\}$.

4.2 The Cooperative Constraint Solving Component

We assume given a constraint domain $\mathcal{X}$ over a signature $\mathcal{F}_e$ of *external* operators, and a set of specialized constraint solvers $CS_1, \ldots, CS_n$. Each solver is a function

$$CS_k : \mathcal{E}qs^{(k)}(\mathcal{F}_e, \mathcal{V}) \to \mathcal{P}_{\text{fin}}(\mathcal{S}ubst(\mathcal{F}_e, \mathcal{V}) \times \mathcal{E}qs^{(k)}(\mathcal{F}_e, \mathcal{V}))$$

where $\mathcal{E}qs^{(k)}(\mathcal{F}_e, \mathcal{V}) \subset \mathcal{P}_{\text{fin}}(\mathcal{E}q(\mathcal{F}_e, \mathcal{V}))$ is the set of constraints that can bw solved with CS_k, $\mathcal{S}ubst(\mathcal{F}_e, \mathcal{V})$ is the set of idempotent substitutions over $\mathcal{F}_e$. The individual solvers are canonical simplifiers, i.e. if $CS_k(S) = \{\langle\theta_i, S_i\rangle \mid 1 \leq i \leq N\}$ then

- γ is a solution of S iff γ is of the form $\theta_p \gamma'_p$ for some $p \in \{1, \ldots, N\}$ and solution γ'_p of S_p.
- every S_i is a CS_k-canonical form, i.e.

$$CS_k(S_i) = \{\langle\{\}, S_i\rangle\}.$$

CS_k is extended to an operator CS^e_k on $\mathcal{P}_{\text{fin}}(\mathcal{S}ubst(\mathcal{F}_e, \mathcal{V}) \times \mathcal{P}_{\text{fin}}(\mathcal{E}q(\mathcal{F}_e, \mathcal{V})))$ defined by

$$CS^e_k(\{\langle\gamma_p, S_p\rangle \mid 1 \leq p \leq N\} = \textstyle\bigcup_{p=1}^{N} \{\langle\gamma_k\theta', S''\gamma_k \cup S'\rangle \mid S' = S \cap \mathcal{E}qs^{(k)}(\mathcal{F}_e, \mathcal{V}), S'' = S - S', \langle\theta', S'\rangle \in CS_k(S_p)\}.$$

The operational principle of the cooperative constraint solving component is defined by a method $\mathcal{S}$ which describes how the computations of $CS_1, \ldots, CS_n$ are combined. We call $\mathcal{S}$ *cooperation strategy*, and define $\mathcal{S}(CS_1, \ldots, CS_n)$ as the fixed-point of $CS^e_n \circ \cdots \circ CS^e_1$. This strategy has been proposed and extensively investigated by Hong [2] in the framework of cooperative CLP. Obviously, when defined, the result of $\mathcal{S}$ is a CS_k-canonical form for all $1 \leq k \leq N$.

Example 2 Assume we want to solve

$$S = \{\lambda t.y'(t) \approx \lambda t.k^2 y(t),\ y(0) \approx 1,\ y(2) \approx 3, y(r) \approx 5\}$$

in variables y, k, r, by using a cooperation of 3 solvers: a solver CS_1 for differential equations, a solver CS_2 for systems of monomial equations, and a solver CS_3 for equations with invertible functions. Then $CS^e_1(\langle\{\}, S\rangle) = S_1$ where $S_1 = \{\langle\{y \mapsto \lambda t.c\, e^{k^2 t}\}, \{c \approx 1,\ c\ e^{2\,k^2} \approx 3,\ c\ e^{r\,k^2} \approx 5\}\rangle\}$ with c a new variable; $CS^e_2(S_1) = S_2$ where $S_2 = \{\langle\{y \mapsto e^{k^2 t}, c \mapsto 1\}, \{e^{2\,k^2} \approx 3,\ e^{r\,k^2} \approx 5\}\rangle\}$; $CS^e_3(S_2) = S_3$ where $S_3 = \{\langle\{y \mapsto \lambda t.e^{k^2 t}, c \mapsto 1\}, \{2\,k^2 \approx \log(3), r\ k^2 \approx \log(5)\}\rangle\}$; $CS^e_1(S_3) = S_3$; $CS^e_2(S_3) = S_4$ where $S_4 = \{\langle\{y \mapsto \lambda t.e^{t\ \log(3)/2}, c \mapsto 1, k \mapsto -\sqrt{\log(3)/2}\}, \{-r\ \log(3)/2 \approx \log(5)\}\rangle, \langle\{y \mapsto \lambda t.e^{t\ \log(3)/2}, c \mapsto 1, k \mapsto \sqrt{\log(3)/2}\}, \{r\ \log(3)/2 \approx \log(5)\}\rangle\}$; $CS^e_1(S_4) = CS^e_2(S_4) = S_4$; and $CS^e_3(S_4) = S_5$ where $S_5 = \{\langle\{y \mapsto \lambda t.e^{t\ \log(3)/2}, c \mapsto 1, k \mapsto -\sqrt{\frac{\log(3)}{2}}, r \mapsto -\frac{2\log(5)}{\log(3)}\}, \{\}\rangle, \langle\{y \mapsto \lambda t.e^{t\ \log(3)/2}, c \mapsto 1, k \mapsto \sqrt{\frac{\log(3)}{2}}, r \mapsto \frac{2\ \log(5)}{\log(3)}\}, \{\}\rangle\}$. In this example, the solution S_5 of S is computed in 9 steps.

To compute $CS^e_k(S)$ we must call CS_k n_k times where n_k is the number of elements in S, and these calls can be realized in parallel. To take advantage of this fact, we have implemented a distributed constraint solving system consisting of

- several instances of solvers running on various machines, and
- a scheduler, which implements the strategy $\mathcal{S}$ by fairly allocating the constraint solvers available to eventually solve each element of S.

The general structure of the distributed constraint solving subsystem of CFLP is depicted in Fig. 2.

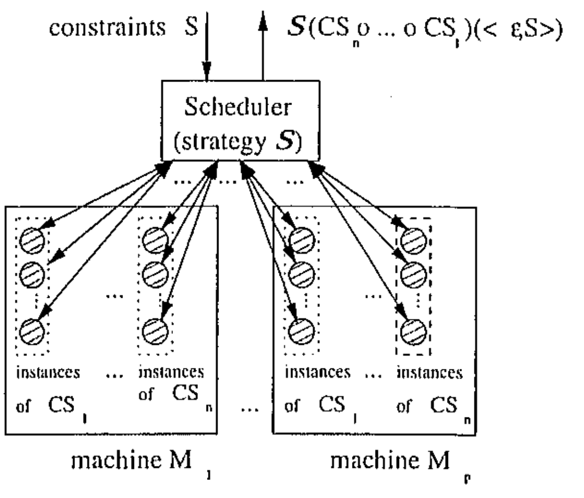


Fig. 2. CFLP: **the cooperative constraint solving component**

The current version of CFLP has integrated four types of constraint solvers:

1. CS_1 : solver which can solve algebraic equations with invertible functions, and yields solutions in terms of formal inverse functions.
2. CS_2 : solves systems of equations between multivariate polynomials over algebraic extensions of the domain of complex numbers,
3. CS_3 : solver for differential equations over algebraic extensions of $\mathbb{C}$,
4. CS_4 : solver for partial differential equations over algebraic extensions of $\mathbb{C}$.

All the components of the cooperative constraint solving system—scheduler and constraint solvers—are implemented in *Mathematica* [8] as *MathLink*-compatible processes which communicate asynchronously by message-passing over *MathLink* connections.

CFLP makes distinction between two types of solving resources:

1. local constraint solvers: these are solvers which run locally as sub-processes of a CFLP session,
2. remote constraint solvers: these solvers are started to run on various machines from outside the CFLP session, and can be shared between different CFLP sessions. The distribution of CFLP provides shell scripts to start and stop running remote constraint solvers.

The user can adjust the computation session by specifying

- the machines $M_1, \dots, M_p$ on which to connect to the remote constraint solvers, and
- the number of local constraint solvers.

5 Conclusion

The system described in this paper is based on a computational model which integrates lazy narrowing for conditional PRS with CCS.

Currently, only a few theoretical results have been generalized to the conditional case. We will continue our research to design efficient and complete calculi for various classes of conditional PRS.

The main intention of our system CFLP was to prove the suitability of our evolvable distributed model for cooperative constraint solving. We didn't focus yet on the design of an efficient implementation.

References

[1] W. Gould. *A matching procedure for ω-order logic*. Scientific Report 4. Air Force Cambridge Research Laboratories, 1966.

[2] H. Hong. Non-linear Constraints Solving over Real Numbers in Constraint Logic Programming (Introducing RISC-CLP). Technical Report 92-08, RISC-Linz, Castle of Hagenberg, Austria, 1992.

[3] G. Huèt. A Unification Algorithm for Typed λ-Calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[4] M. Marin. *Functional Logic Programming with Distributed Constraint Solving*. PhD thesis, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Schloss Hagenberg, April 2000.

[5] M. Marin, T. Ida, and T. Suzuki. On Reducing the Search Space of Higher-Order Lazy Narrowing. In A. Middeldorp and T. Sato, editors, *FLOPS'99*, volume 1722 of *LNCS*, pages 225–240. Springer-Verlag, 1999.

[6] A. Middeldorp and S. Okui. A Deterministic Lazy Narrowing Calculus. *Journal of Symbolic Computation*, 25(6):733–757, 1998.

[7] A. Middeldorp, S. Okui, and T. Ida. Lazy Narrowing: Strong Completeness and Eager Variable Elimination. *Theoretical Computer Science*, 167(1,2):95–130, 1996.

[8] S. Wolfram. *The Mathematica Book*. Third Edition. Wolfram Media and Cambridge University Press, 1996.

Higher-order Lazy Narrowing Calculi in Perspective

Mircea Marin[1], Tetsuo Ida[1], and Taro Suzuki[2]

[1] Institute of Information Sciences and Electronics
University of Tsukuba, Tsukuba 305-8573, Japan
mmarin@score.is.tsukuba.ac.jp
ida@score.is.tsukuba.ac.jp
[2] School of Information Science,
JAIST Hokuriku, 923-1292, Japan
t_suzuki@jaist.ac.jp

Abstract. Higher-order lazy narrowing (HOLN for short) is a computational model for higher-order functional logic programming. It can be viewed as an extension of first-order lazy narrowing with inference rules to solve equations involving λ-abstractions and higher-order variables.
A common feature of the HOLN calculi proposed so far is the high nondeterminism between the inference rules designed to solve equations which involve higher-order variables. In this paper we present various refinements of HOLN towards more deterministic versions. The refinements are defined for classes of higher-order functional logic programs which are useful for programming purposes. Our work draws on two sources: the calculus LN for pattern rewrite systems [Pre98] and the first-order lazy narrowing calculus LNC and its deterministic refinements [MO98].

1 Introduction

Recent years have witnessed a growing interest in extending lazy narrowing with higher-order constructs, in an attempt to improve the expressive power of functional logic programming (FLP). The capability to handle λ-abstractions and function variables is desirable in FLP, mainly because one can use the abstraction principle and can quantify over predicate and function symbols. The cost of such an extension is the high nondeterminism between the inference rules designed to solve equations with function variables. This problem has first been observed in the context of solving higher-order equations in empty equational theories [Gou66], where it is avoided by adopting the idea of pre-unification [Huè75] instead of full higher-order unification. The main idea of pre-unification is to compute pre-unifiers instead of unifiers, by avoiding to solve the problematic flex/flex equations.

Compared with higher-order pre-unification, HOLN as operational principle for higher-order FLP adds one more complication: the high nondeterminism of solving equations where at least one side is a flex term, i.e. a term of the form $\lambda\overline{x}.X(\overline{s_n})$ with X a free variable. The nondeterminism between the inference rules of HOLN can be reduced by restricting it to particular classes of higher-order functional logic programs (i.e., higher-order term rewriting systems) which guarantee that the resulted calculus is complete. For programming purposes, the classes of higher-order functional logic programs must be chosen to be expressive enough.

Recently, various specializations of higher-order lazy narrowing to particular classes of higher-order rewrite systems have been proposed (see, e.g. [SNI97,Pre98,MMIY99,MIS99]) in an attempt to define a suitable operational model for higher-order FLP.

In the sequel we give a brief account of the higher-order lazy narrowing calculi designed by us for higher-order FLP. Figure 1 illustrates the higher-order lazy narrowing calculi which have been proposed so far and are relevant to our research. The arrows indicate design dependencies between calculi. Our calculi are displayed in boldfaced font. For the calculi where completeness holds with respect to a certain equation selection strategy, we have specified the strategy in parentheses immediately after the name of the corresponding calculus.

2 Lazy Narrowing for Applicative Term Algebras

For FLP, the advantage of applicative term algebra over first-order term algebra is the capability to use function variables in writing functional logic programs and goals. Our first investigation of higher-order FLP was set up in the framework of applicative term algebras, where we tried to modify the applicative version of the first-order lazy narrowing calculus LNC into a calculus which is more deterministic. The outcome is the calculus LNCA [MMIY99] for applicative term rewriting systems (ATRS for short). The inference rules of LNC and LNCA are shown in Appendices A and B.

Compared to LNC, LNCA is more deterministic because of the specialization [on] of the rule [o]: the selection of the rewrite rule(s) in an [on]-step is driven by the leftmost innermost symbol instead of the leftmost outermost symbol. As a consequence, the search tree for solutions has fewer branches. However, LNCA-derivations may be longer than the corresponding LNC-derivation for the same computed solution.

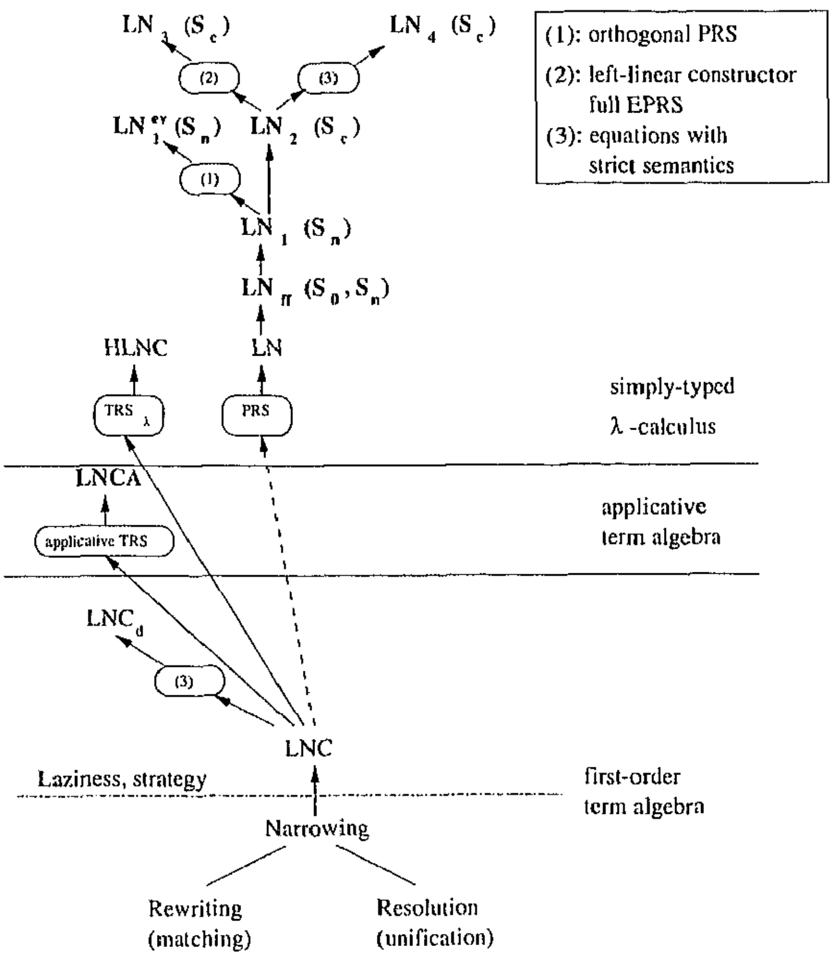


Fig.1. Narrowing calculi: dependency diagram

Main result. We have shown [MMIY99] that by adopting the leftmost equation selection strategy, the calculus LNCA is sound and complete for the computation of normalized solutions. We expect that a further analysis of the similarities between LNC and LNCA will enable to define more deterministic specializations of the calculus LNCA which are sound and complete for large classes of ATRSs.

3 Lazy Narrowing for Simply-Typed Term Algebras

Simply-typed λ-calculus is a fundamental and powerful theory of application and abstraction governed by a basic theory of types. The terms of the simply-typed λ-calculus are assumed to satisfy certain rules, and most often they are identified modulo the rules α, β and η of the λ-calculus. A suitable generalization of first-order lazy narrowing to the algebra of simply-typed λ-terms is supposed to take into account these requirements.

For FLP, simply-typed term algebras are much more powerful than applicative term algebras mainly because of the mechanism of λ-abstraction, which provides support for universal quantification.

Various theoretical frameworks for higher-order FLP in algebras of simply typed λ-terms are being determined by how term rewriting is defined. (See, e.g. [MN94,vO94,vdP94].) In our framework we use the notion of higher-order term rewriting with pattern rewrite systems (PRS for short) proposed by Nipkow [Nip91]. It has the advantage to inherit many crucial features from first-order equational logic (e.g., higher-order pattern unification is unitary). This aspect is useful in defining suitable HOLN calculi for equational theories represented with PRSs.

Our research of higher-order lazy narrowing for PRSs draws on two sources: (a) the higher-order lazy narrowing calculus LN [Pre98] for higher-order pre-unification in equational theories represented with PRS, and (b) the first-order lazy narrowing calculus LNC and its deterministic refinements [MOI96,MO98].

3.1 The calculus $\mathrm{LN}_{\mathrm{ff}}$

A careful analysis of the similarities between LN and LNC revealed the possibility to define a suitable higher-order counterpart of the calculus LNC which we called $\mathrm{LN}_{\mathrm{ff}}$ [MIS99], and to lift to $\mathrm{LN}_{\mathrm{ff}}$ some of the deterministic refinements of LNC proposed in [MO98].

$\mathrm{LN}_{\mathrm{ff}}$ can be regarded as an extension of the calculus LN: whereas LN is designed to solve systems of oriented equations, $\mathrm{LN}_{\mathrm{ff}}$ can solve both oriented and unoriented equations. In addition, $\mathrm{LN}_{\mathrm{ff}}$ can perform full pattern unification. The inference rules of $\mathrm{LN}_{\mathrm{ff}}$ are displayed below. We adopted the terminology and notational conventions used in [Pre98]. We employ the relational symbols $\rhd$ for oriented equality, and $\approx$ for unoriented equality. The symbol $\cong$ denotes

- $\approx, \approx^{-1}$, $\rhd$, or $\rhd^{-1}$ in the inference rules [ffs], [ffd], [i], [p],
- $\approx, \approx^{-1}$, or $\rhd$ in the inference rules [del], [d], [on], [ov].

It is assumed that $\cong$ denotes the same relational symbol in the upper and lower sides of the inference rules of $\mathrm{LN}_{\mathrm{ff}}$.

[del] *deletion*

$$\frac{G, \lambda\overline{x}.t \cong \lambda\overline{x}.t, G'}{G, G'}$$

[d] *decomposition*

$$\frac{G, \lambda\overline{x}.v(\overline{s_n}) \cong \lambda\overline{x}.v(\overline{t_n}), G'}{G, \overline{\lambda\overline{x}.s_n \cong \lambda\overline{x}.t_n}, G'}$$

[i] *imitation*

$$\frac{G, \lambda\overline{x}.X(\overline{s_n}) \cong \lambda\overline{x}.g(\overline{t_m}), G'}{(G, \overline{\lambda\overline{x}.H_m(\overline{s_n}) \cong \lambda\overline{x}.t_m}, G')\theta}$$

where $\theta = \{X \mapsto \lambda\overline{x_n}.g(\overline{H_m(\overline{x_n})})\}$ and $\overline{H_m}$ are fresh variables.

[p] *projection*

$$\frac{G, \lambda\overline{x}.X(\overline{s_n}) \cong \lambda\overline{x}.t, G'}{(G, \lambda\overline{x}.s_i(\overline{H_p(\overline{s_n})}) \cong \lambda\overline{x}.t, G')\theta}$$

where $1 \leq i \leq n$, $\lambda\overline{x}.t$ is rigid, $\theta = \{X \mapsto \lambda\overline{y_n}.y_i(\overline{H_p(\overline{y_n})})\}$, $y_i : \overline{\tau_p} \to \tau$, and $\overline{H_p : \tau_p}$ are fresh variables.

[on] *outermost narrowing at nonvariable position*

$$\frac{G, \lambda\overline{x}.f(\overline{s_n}) \cong \lambda\overline{x}.t, G'}{G, \overline{\lambda\overline{x}.s_n \rhd \lambda\overline{x}.l_n}, \lambda\overline{x}.r \cong \lambda\overline{x}.t, G'}$$

if $f(\overline{l_n}) \to r$ is a fresh variant of an $\overline{x}$-lifted rule.

[ov] *outermost narrowing at variable position*

$$\frac{G, \lambda\overline{x}.X(\overline{s_m}) \cong \lambda\overline{x}.t, G'}{(G, \overline{\lambda\overline{x}.H_n(\overline{s_m}) \rhd \lambda\overline{x}.l_n}, \lambda\overline{x}.r \cong \lambda\overline{x}.t, G')\theta}$$

if $\lambda\overline{x}.t$ is rigid, $f(\overline{l_n}) \to r$ is a fresh variant of an $\overline{x}$-lifted rule and $\theta = \{X \mapsto \lambda\overline{y_m}.f(\overline{H_n(\overline{y_m})})\}$ with $\overline{H_n}$ fresh variables of appropriate types.

[ffs] *flex/flex same*

$$\frac{G, \lambda\overline{x}.X(\overline{y_n}) \cong \lambda\overline{x}.X(\overline{y'_n}), G'}{(G,G')\theta}$$

where $\theta = \{X \mapsto \lambda\overline{y_n}.H(\overline{z_p})\}$ with $\{\overline{z_p}\} = \{y_i \mid 1 \leq i \leq n \text{ and } y_i = y'_i\}$.

[ffd] *flex/flex different*

$$\frac{G, \lambda\overline{x}.X(\overline{y_m}) \cong \lambda\overline{x}.Y(\overline{y'_n}), G'}{(G,G')\theta}$$

where $\theta = \{X \mapsto \lambda\overline{y_m}.H(\overline{z_p}), Y \mapsto \lambda\overline{y'_n}.H(\overline{z_p})\}$ with $\{\overline{z_p}\} = \{\overline{y_m}\} \cap \{\overline{y'_n}\}$.

The equations $\overline{\lambda\overline{x}.s_n \rhd \lambda\overline{x}.t_n}$ generated by the rule [on], and the equations $\overline{\lambda\overline{x}.H_n(\overline{s_m}) \rhd \lambda\overline{x}.t_n}$ generated by the rule [ov] are called *parameter-passing* equations.

In the sequel we will analyze the main properties of LN_{ff} and define possible refinements towards more deterministic versions. An important property used in the design of our refinements is whether an oriented equation is a descendant of (i.e., it is produced from) a parameter-passing equation or not. Henceforth, whenever we want to emphasize that an equation $\lambda\overline{x}.s \rhd \lambda\overline{x}.t$ is a descendant of a parameter-passing equation, we will write it in the form $\lambda\overline{x}.s \blacktriangleright \lambda\overline{x}.t$.

In [MIS99] it has been shown that LN_{ff} is incomplete because we have added the inference rules [ffs] and [ffd] for pattern unification. We recover completeness by adopting a suitable equation selection strategy $\mathcal{S}_0$. In our framework, a strategy is a family of equation selection functions determined by a criterion which takes into account the history of the lazy narrowing derivation [Mar00]. To capture the relevant properties of the history, we introduced the notion of *precursor* of an equation in a goal. Informally, an equation has precursor if it descends from an equation which was selected in an [on]- or [ov]-step; if so, the precursors of that equation are the equations produced from the parameter-passing equations created in the first [on]- or [ov]-step with the property mentioned above.

Solving flex/rigid and rigid/flex equations is highly nondeterministic because all rules [ov], [i], [p] have to be considered to guarantee completeness. We reduce this nondeterminism by defining a sub-strategy $\mathcal{S}_n$ of $\mathcal{S}_0$, i.e. $\mathcal{S}_n \subset \mathcal{S}_0$, which delays the selection of such problematic equations as much as possible.

Main result. LN_{ff} with strategy $\mathcal{S}_0$ is sound and complete.

3.2 The Calculus LN_1

LN_1 is a refinement of the calculus LN_{ff}. It is defined for confluent PRSs, and restricts the application of rule [ov] as shown below.

[ov] *outermost narrowing at variable position*

$$\frac{G, \lambda\overline{x}.X(\overline{s_m}) \cong \lambda\overline{x}.t, G'}{(G, \overline{\lambda\overline{x}.H_n(\overline{s_m}) \blacktriangleright \lambda\overline{x}.l_n}, \lambda\overline{x}.r \cong \lambda\overline{x}.t, G')\theta}$$

if

- $\lambda\overline{x}.X(\overline{s_m})$ is a flex-pattern only if the equation $\lambda\overline{x}.X(\overline{s_m}) \cong \lambda\overline{x}.t$ has no precursors,
- $\lambda\overline{x}.t$ is a rigid term,
- $f(\overline{l_n}) \to r$ is a fresh variant of an $\overline{x}$-lifted rule, and
- $\theta = \{X \mapsto \lambda\overline{y_m}.f(\overline{H_n(\overline{y_m})})\}$ with $\overline{H_n}$ fresh variables of appropriate types.

Main result. The calculus LN_1 with strategy S_n is sound and complete.

3.3 Eager variable elimination

This refinement addresses the possibility to avoid the application of rule [on] to certain equations of the form $\lambda\overline{x}.f(\overline{s_n}) \rhd \lambda\overline{x}.t$ with f defined function symbol, and was inspired by the eager variable elimination problem described in [MOI96] for the calculus LNC. In the first-order case, it has been shown that for orthogonal term rewriting systems, the application of the variable elimination rule

$$\frac{G, t \approx X, G'}{(G, G')\theta} \quad \text{where } \theta = \{X \mapsto t\} \text{ if } X \notin \mathcal{V}(t)$$

prior to other applicable inference rules preserves the completeness of the calculus, if the selected equation is a descendant of a parameter-passing equation. It is not hard to see that the higher-order counterpart of the eager variable elimination problem refers to the possibility to avoid applying [on] to descendants of parameter-passing equations. We denote by C^{ev} the calculus resulted from a higher-order lazy narrowing calculus C by eliminating the application of rule [on] to descendants of parameter-passing equations.

The orthogonality assumption in the proof of this result is necessary because it is made use of the standardization theorem. We noted that the standardization theorem was proved for higher-order PRSs [vO96]. By using this result, we succeeded to prove that for orthogonal PRSs we can avoid to apply [on] to descendants of parameter-passing equations without losing completeness.

Main result. The calculus LN_1^{ev} is sound and complete.

3.4 Lazy Narrowing with Elimination of Redundant Equations

Upon computations of normalized solutions with the calculus LN_1 with strategy $\mathcal{S}_c$, it may happen to generate new equations which do not contribute to the computation of the final solution. We call such equations *redundant*, and propose the following criterion to detect them.

Definition 1 (redundant equation) *Let $G \Rightarrow^+ G'$ be an* LN_1*-derivation which respects strategy $\mathcal{S}_n$, and $e = \lambda\overline{x}.s \rhd \lambda\overline{x}.X(\overline{y})$ an equation in G' with $\mathcal{V}(\lambda\overline{y}.s) \cap \{\overline{x}\} = \emptyset$. Then e is* redundant *if*

- *e is a descendant of a parameter-passing equation,*
- *$X \notin \mathcal{V}(e')$ for all equations $e' \in G' \setminus \{e\}$, and*
- *$X \notin \mathcal{V}(\lambda\overline{x}.s)$.*

We denote by LN_2 the calculus obtained by adding the inference rule

$$[\mathrm{rm}] \qquad \frac{G, e, G'}{G, G'} \quad \text{if } e \text{ is redundant.}$$

to the calculus LN_1, and by making [rm] the inference rule with the second highest priority, after [del]. We denote by $\mathcal{S}_c$ the strategy obtained from $\mathcal{S}_n$ by allowing the selection of any redundant equation.

Main result. LN_2 with strategy $\mathcal{S}_c$ is sound and complete.

3.5 Lazy narrowing for left-linear constructor PRSs

The restriction of programs to left-linear constructor TRSs is widely accepted in functional logic programming. The higher-order counterpart of this class of programs are the left-linear fully-extended constructor PRSs (LECPRS for short). These are PRSs consisting of rewrite rules of the form $l \to r$ with l a fully-extended pattern (see definition below) which is free of defined function symbols.

Definition 2 (fully-extended pattern) *A simply-typed λ-term is called* fully-extended pattern *if whenever $X(\overline{s_n})$ is a subterm of l at position p then $\overline{s_n}$ is the sequence of all distinct bound variables in the scope of $l|_p$.*

For the case of functional logic programs represented by LECPRSs we propose a new calculus, which we call LN_3. LN_3 is obtained by modifying the calculus LN_1 to avoid the creation of descendants of parameter-passing equations of the form $\lambda\overline{x}.s \rhd \lambda\overline{x}.t$ with defined symbols in $\lambda\overline{x}.t$, in a way that preserves completeness. As a consequence, the computation with LN_1-derivation becomes more deterministic because we can discard all LN_3-derivations in which occur descendants of parameter-passing equations of the form $\lambda\overline{x}.s \rhd \lambda\overline{x}.t$ with defined symbols in $\lambda\overline{x}.t$.

The modification LN_3 of the calculus LN_1 is realized by

1. adding a new inference rule [c],
2. giving to the newly added inference rule the highest priority, and
3. discarding the derivations with descendants of parameter-passing equations of the form $\lambda\overline{x}.s \rhd \lambda\overline{x}.t$ with defined symbols in $\lambda\overline{x}.t$.

The [c]-rule. In the first-order case, a well-known result is that for left-linear constructor term rewriting systems, the completeness is preserved if we compute only with LNC-refutations where

- the leftmost equation selection strategy is used, and
- the descendants of parameter-passing equations have constructor terms to the right-hand side.

An immediate consequence of this observation is that solving descendants of parameter-passing equations of the form $X \approx t$ becomes completely deterministic with LNC: the [v]-rule is sufficient for solving such equations.

Unfortunately, the above-mentioned property of LNC has no direct correspondent in the calculus LN_2. This is so mainly because there is no higher-order counterpart of the leftmost equation selection strategy. This fact is illustrated in the example below.

Example 1 *Consider the left-linear constructor PRS* $\mathcal{R} = \{f(X) \to X\}$ *and the goal* $G = f(Y(X)) \rhd a$. *Then any* LN_2*-derivation which respects strategy* $\mathcal{S}_c$ *starts with*

$$\underline{f(Y(X)) \rhd a} \Rightarrow_{[\mathrm{on}]} G = Y(X) \rhd X_1, \underline{X_1 \rhd a}$$
$$\Rightarrow_{[\mathrm{ov}],\sigma=\{X_1 \mapsto f(H)\}} G' = Y(X) \rhd f(H), H(X) \rhd X_2, X_2 \rhd a$$

The application of [ov] *in the second inference step introduces the defined function symbol* f *in the right-hand side of the leftmost parameter-passing equation of* G'. □

To avoid this behavior of LN_2 with strategy $\mathcal{S}_c$, we add a new rule:

$$[\mathrm{c}]\ \frac{G, \lambda\overline{z}.s(\overline{y}) \blacktriangleright \lambda\overline{z}.X(\overline{y}), G', \lambda\overline{x}.X(\overline{t}) \cong \lambda\overline{x}.u, G''}{G, \lambda\overline{z}.s(\overline{y}) \blacktriangleright \lambda\overline{z}.X(\overline{y}), G', \lambda\overline{x}.s(\overline{t}) \cong \lambda\overline{x}.u, G''}$$

if $\lambda\overline{x}.u$ is rigid and $\mathcal{V}(s) \cap \{\overline{z}\} = \emptyset$.

The equation selected by the [c]-rule is $\lambda\overline{x}.X(\overline{t}) \cong \lambda\overline{x}.u$ and its descendant is defined to be $\lambda\overline{x}.s(\overline{t}) \cong \lambda\overline{x}.u$. The notions of precursor and descendant are carried over to the [c]-rule in the natural way.

The new calculus. We denote by LN_3 the calculus obtained from LN_2 by adding the rule [c] and by applying it instead of [ov] and [i] whenever possible.

Main result. LN_3 with strategy $\mathcal{S}_c$ is sound and complete.

3.6 Lazy Narrowing for Equations with Strict Semantics

In first-order functional logic programming, two expressions are considered to be strictly equal if they reduce to the same ground constructor term (see, e.g. [AEH94,GLMP91,MO98]).

We propose the following higher-order counterpart of the notion of strict solution of an equation.

Definition 3 (strict solution) *A substitution θ is a* strict *solution of an equation $s \cong t$, where $\cong \in \{\approx, \rhd\}$, if there exists a ground constructor term u such that:*

- $s\theta \rightarrow^*_{\mathcal{R}} u$ *and* $t\theta \rightarrow^*_{\mathcal{R}} u$, *if* $\cong$ *is* $\approx$,
- $s\theta \rightarrow^*_{\mathcal{R}} u$ *and* $t\theta = u$, *if* $\cong$ *is* $\rhd$.

To accomodate the notion of strict solution into our higher-order FLP setting, we distinguish four types of equality: unoriented equality, oriented equality, strict unoriented equality, and strict oriented equality. Correspondingly, we employ the equality symbols $\approx$, $\rhd$, $\doteq$ and $\gg$, and specialize the calculus LN_2 to a calculus LN_4 consisting of four mutually disjoint subcalculi:

$\mathrm{LN}_4^{\approx}$: inference rules for selected unoriented equations
$\mathrm{LN}_4^{\rhd}$: inference rules for selected oriented equations,
$\mathrm{LN}_4^{\doteq}$: inference rules for selected strict unoriented equations,
$\mathrm{LN}_4^{\gg}$: inference rules for selected strict oriented equations.

Whereas subcalculi $\mathrm{LN}_4^{\approx}$ and $\mathrm{LN}_4^{\rhd}$ coincide with the corresponding subcalculi $\mathrm{LN}_2^{\approx}$ and $\mathrm{LN}_2^{\rhd}$ of LN_2, the subcalculi $\mathrm{LN}_4^{\doteq}$ and $\mathrm{LN}_4^{\gg}$ behave more deterministic than $\mathrm{LN}_2^{\approx}$ and $\mathrm{LN}_2^{\rhd}$ (see Appendices C, D).

Main result. The calculus LN_4 with strategy $\mathcal{S}_c$ is sound and complete.

4 Conclusions and Future Work

Various refinements of a higher-order lazy narrowing calculus have been proposed, in an attempt to reduce its high nondeterminism and to make it suitable as operational model for functional logic programming. All calculi proposed by us are sound and complete.

A powerful mechanism for FLP is lazy narrowing with conditional term rewriting systems. The extension of lazy narrowing to the conditional term rewriting case has been successfully pursued in the first-order case, and proposals for higher-order versions of conditional lazy narrowing are already available. (See, e.g. [Pre98].) As future work, we intend to generalize our framework of higher-order lazy narrowing with PRS to the conditional case, and to find deterministic refinements for classes of conditional PRS which are useful for programming purposes.

References

[AEH94] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*, pages 268–278, Portland, 1994.

[GLMP91] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel-LEAF: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42(2):138–185, 1991.

[Gou66] W.E. Gould. *A matching procedure for ω-order logic*. Scientific Report 4. Air Force Cambridge Research Laboratories, 1966.

[Huè75] G. Huèt. A Unification Algorithm for Typed λ-Calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Mar00] M. Marin. *Functional Logic Programming with Distributed Constraint Solving*. PhD thesis, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Schloss Hagenberg, April 2000.

[MIS99] M. Marin, T. Ida, and T. Suzuki. On Reducing the Search Space of Higher-Order Lazy Narrowing. In A. Middeldorp and T. Sato, editors, *FLOPS'99*, volume 1722 of *LNCS*, pages 225–240. Springer-Verlag, 1999.

[MMIY99] M. Marin, A. Middeldorp, T. Ida, and T. Yanagi. LNCA: A Lazy Narrowing Calculus for Applicative Term Rewriting Systems. Technical Report ISE-TR-99-158, Institute of Information Sciences and Electronics, University of Tsukuba, Tsukuba, Japan, 1999.

[MN94] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. Technical report, Institute für Informatik, TU München, 1994.

[MO98] A. Middeldorp and S. Okui. A Deterministic Lazy Narrowing Calculus. *Journal of Symbolic Computation*, 25(6):733–757, 1998.

[MOI96] A. Middeldorp, S. Okui, and T. Ida. Lazy Narrowing: Strong Completeness and Eager Variable Elimination. *Theoretical Computer Science*, 167(1,2):95–130, 1996.

[Nip91] T. Nipkow. Higher-order critical pairs. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 342–349, Amsterdam, the Netherlands, July 1991. IEEE Computer Society Press.

[Pre98] C. Prehofer. *Solving Higher-Order Equations. From Logic to Programming.* Foundations of Computing. Birkäuser Boston, 1998.

[SNI97] T. Suzuki, K. Nakagawa, and T. Ida. Higher-Order Lazy Narrowing Calculus: A Computation Model for a Higher-order Functional Logic Language. In *Proceedings of Sixth International Joint Conference, ALP '97 - HOA '97*, volume 1298 of *LNCS*, pages 99–113, Southampton, 1997.

[vdP94] J. van de Pol. Termination proofs for higher-order rewrite systems. In J. Heering, K. Meinke, M. Möller, and T. Nipkow, editors, *Higher-Order Algebra, Logic and Term Rewriting*, volume 816 of *LNCS*, pages 305–325. Springer-Verlag, 1994.

[vO94] V. van Oostrom. *Confluence for Abstract and Higher-Order Rewriting.* PhD thesis, Vrije Universiteit, Amsterdam, 1994.

[vO96] V. van Oostrom. Higher-order Families. In *International Conference on Rewriting Techniques and Applications*, LNCS, 1996.

A The Calculus LNC

[d] *decomposition*

$$\frac{G, f(\overline{s_n}) \approx f(\overline{t_n}), G'}{G, \overline{s_n \approx t_n}, G'}$$

[v] *variable elimination*

$$\frac{G, X \approx t, G'}{(G, G')\theta} \quad \frac{G, t \approx X, G'}{(G, G')\theta}$$

where $\theta = \{X \mapsto t\}$ if $X \notin \mathcal{V}(t)$.

[i] *imitation*

$$\frac{G, f(\overline{s_n}) \approx X, G'}{(G, \overline{s_n \approx X_n}, G')\theta} \quad \frac{G, X \approx f(\overline{s_n}), G'}{(G, \overline{X_n \approx s_n}, G')\theta}$$

where $\theta = \{X \mapsto f(\overline{X_n})\}$ with $\overline{X_n}$ fresh variables.

[o] *outermost narrowing*

$$\frac{G, f(\overline{s_n}) \approx t, G'}{G, \overline{s_n \approx l_n}, r \approx t, G'} \quad \frac{G, t \approx f(\overline{s_n}), G'}{G, \overline{s_n \approx l_n}, t \approx r, G'}$$

if $f(\overline{l_n}) \to r$ is a fresh variant of a rewrite rule.

[t] *trivial equation*

$$\frac{G, t \approx t, G'}{G, G'}$$

B The Calculus LNCA

[of] *outermost narrowing for head-function terms*

$$\frac{f\ \overline{s_m}\ \overline{t_n} \simeq t, G}{\overline{s_m \approx u_m}, r\ \overline{t_n} \approx t, G}$$

if $f\ \overline{u_m} \to r$ is a fresh variant of a rewrite rule

[ov] *outermost narrowing for head-variable terms*

$$\frac{X\ \overline{s_m}\ \overline{t_n} \simeq t, G}{(\overline{s_m \approx v_m}, r\ \overline{t_n} \approx t, G)\theta}$$

if there exists a fresh variant $f\ \overline{u_k}\ \overline{v_m} \to r$ of a rewrite rule, $m > 0$, and $\theta = \{X \mapsto f\ \overline{u_k}\}$.

[if] *imitation for head-function terms*

$$\frac{f\ \overline{s_m}\ \overline{t_n} \approx X\ \overline{u_n}, G}{(\overline{s_m \approx X_m}, \overline{t_n \approx u_n}, G)\theta} \qquad \frac{X\ \overline{u_n} \approx f\ \overline{s_m}\ \overline{t_n}, G}{(\overline{s_m \approx X_m}, \overline{u_n \approx t_n}, G)\theta}$$

if $m > 0$, $\theta = \{X \mapsto f\ \overline{X_m}\}$ with $X_1, \ldots, X_m$ fresh variables.

[iv] *imitation for head-variable terms*

$$\frac{Y\ \overline{s_m}\ \overline{t_n} \approx X\ \overline{u_n}, G}{(\overline{s_m \approx X_m}, \overline{t_n \approx u_n}, G)\theta} \qquad \frac{X\ \overline{u_n} \approx Y\ \overline{s_m}\ \overline{t_n}, G}{(\overline{s_m \approx X_m}, \overline{u_n \approx t_n}, G)\theta}$$

if $m > 0$, $X \neq Y$ and $\theta = \{X \mapsto Y\ \overline{X_m}\}$ with $X_1, \ldots, X_m$ fresh variables.

[df] *decomposition for head-function terms*

$$\frac{f\ \overline{s_n} \approx f\ \overline{t_n}, G}{\overline{s_n \approx t_n}, G}$$

[dv] *decomposition for head-variable terms*

$$\frac{X\ \overline{s_n} \approx X\ \overline{t_n}, G}{\overline{s_n \approx t_n}, G}$$

[vf] *variable elimination for head-function terms*

$$\frac{f\ \overline{s_m}\ \overline{t_n} \approx X\ \overline{u_n}, G}{(\overline{t_n \approx u_n}, G)\theta} \qquad \frac{X\ \overline{u_n} \approx f\ \overline{s_m}\ \overline{t_n}, , G}{(\overline{u_n \approx t_n}, G)\theta}$$

if $X \notin \mathcal{V}(f\ \overline{s_m})$ and $\theta = \{X \mapsto f\ \overline{s_m}\}$.

[vv] *variable elimination for head-variable terms*

$$\frac{Y\ \overline{s_m}\ \overline{t_n} \approx X\ \overline{u_n}, G}{(\overline{t_n \approx u_n}, G)\theta} \qquad \frac{X\ \overline{u_n} \approx Y\ \overline{s_m}\ \overline{t_n}, G}{(\overline{u_n \approx t_n}, G)\theta}$$

if $X \notin \mathcal{V}(Y\ \overline{s_m})$ and $\theta = \{X \mapsto Y\ \overline{s_m}\}$.

C The subcalculus $\mathrm{LN}_4^{\doteq}$

[d] *decomposition*

$$\frac{G, \lambda\overline{x}.v(\overline{s_n}) \doteq \lambda\overline{x}.v(\overline{t_n}), G'}{G, \overline{\lambda\overline{x}.s_n \doteq \lambda\overline{x}.t_n}, G'}$$

if $v \in \mathcal{F}_c \cup \{\overline{x}\}$

[i] *imitation*

$$\frac{G, \lambda\overline{x}.X(\overline{s_n}) \cong \lambda\overline{x}.g(\overline{t_m}), G'}{(G, \overline{\lambda\overline{x}.H_m(\overline{s_n}) \cong \lambda\overline{x}.t_m}, G')\theta}$$

where $\cong \in \{\doteq, \doteq^{-1}\}$, $g \in \mathcal{F}_c$, $\theta = \{X \mapsto \lambda\overline{x_n}.g(\overline{H_m(\overline{x_n})})\}$ and $\overline{H_m}$ are fresh variables.

[p] *projection*

$$\frac{G, \lambda\overline{x}.X(\overline{s_n}) \cong \lambda\overline{x}.t, G'}{(G, \lambda\overline{x}.s_i(\overline{H_p(\overline{s_n})}) \cong \lambda\overline{x}.t, G')\theta}$$

where $\cong \in \{\doteq, \doteq^{-1}\}$, $1 \leq i \leq n$, $\lambda\overline{x}.t$ is rigid, $\theta = \{X \mapsto \lambda\overline{y_n}.y_i(\overline{H_p(\overline{y_n})})\}$, $y_i : \overline{\tau_p} \to \tau$, and $\overline{H_p : \tau_p}$ are fresh variables of appropriate types.

[ov] *outermost narrowing at variable position*

$$\frac{G, \lambda\overline{x}.X(\overline{s_m}) \cong \lambda\overline{x}.v(\overline{t}), G'}{(G, \overline{\lambda\overline{x}.H_n(\overline{s_m}) \blacktriangleright \lambda\overline{x}.l_n}, \lambda\overline{x}.r \cong \lambda\overline{x}.v(\overline{t}), G')\theta}$$

if $\cong \in \{\doteq, \doteq^{-1}\}$, $v \in \{\overline{x}\} \cup \mathcal{F}_c$, $\underline{f(\overline{l_n}) \to r}$ is a fresh variant of an $\overline{x}$-lifted rule, $\theta = \{X \mapsto \lambda\overline{y_m}.f(\overline{H_n(\overline{y_m})})\}$ with $\overline{H_n}$ fresh variables of appropriate types, and $\overline{s_m}$ are distinct bound variables only if the selected equation has precursors.

[on] *outermost narrowing at nonvariable position*

$$\frac{G, \lambda\overline{x}.f(\overline{s_n}) \cong \lambda\overline{x}.t, G'}{G, \overline{\lambda\overline{x}.s_n \blacktriangleright \lambda\overline{x}.l_n}, \lambda\overline{x}.r \cong \lambda\overline{x}.t, G'}$$

if $\cong \in \{\doteq, \doteq^{-1}\}$ and $f(\overline{l_n}) \to r$ is a fresh variant of an $\overline{x}$-lifted rule.

[ffs] *flex/flex same*

$$\frac{G, \lambda\overline{x}.X(\overline{y_n}) \doteq \lambda\overline{x}.X(\overline{y'_n}), G'}{(G, G')\theta}$$

where $\theta = \{X \mapsto \lambda\overline{y_n}.H(\overline{z_p})\}$ with $\{\overline{z_p}\} = \{y_i \mid 1 \leq i \leq n \text{ and } y_i = y'_i\}$.

[ffd] *flex/flex different*

$$\frac{G, \lambda\overline{x}.X(\overline{y_m}) \doteq \lambda\overline{x}.Y(\overline{y'_n}), G'}{(G, G')\theta}$$

where $\theta = \{X \mapsto \lambda\overline{y_m}.H(\overline{z_p}), Y \mapsto \lambda\overline{y'_n}.H(\overline{z_p})\}$ with $\{\overline{z_p}\} = \{\overline{y_m}\} \cap \{\overline{y'_n}\}$.

[c] [c]-*rule*

$$\frac{G, \lambda\overline{z}.s(\overline{y}) \blacktriangleright \lambda\overline{z}.X(\overline{y}), G', \lambda\overline{x}.X(\overline{t}) \cong \lambda\overline{x}.u, G''}{G, \lambda\overline{z}.s(\overline{y}) \blacktriangleright \lambda\overline{z}.X(\overline{y}), G', \lambda\overline{x}.s(\overline{t}) \cong \lambda\overline{x}.u, G''}$$

if $\cong \in \{\doteq, \doteq^{-1}\}$, $\lambda\overline{x}.u$ is rigid and $\mathcal{V}(s) \cap \{\overline{z}\} = \emptyset$.

D The subcalculus $\mathrm{LN}_4^{\gg}$

The inference rules [d], [i], [p], [ffs], [ffd] of the subcalculus $\mathrm{LN}_4^{\gg}$ are obtained from the corresponding inference rules of $\mathrm{LN}_4^{\doteq}$ by replacing $\doteq$ with $\gg$. The rules [ov], [on] and [c] of $\mathrm{LN}_4^{\gg}$ are shown below.

[ov] *outermost narrowing at variable position*

$$\frac{G, \lambda\overline{x}.X(\overline{s_m}) \gg \lambda\overline{x}.v(\overline{t}), G'}{(G, \overline{\lambda\overline{x}.H_n(\overline{s_m}) \blacktriangleright \lambda\overline{x}.l_n}, \lambda\overline{x}.r \gg \lambda\overline{x}.v(\overline{t}), G')\theta}$$

if $v \in \{\overline{x}\} \cup \mathcal{F}_c$, $f(\overline{l_n}) \to r$ is a fresh variant of an $\overline{x}$-lifted rule, $\theta = \{X \mapsto \lambda\overline{y_m}.f(\overline{H_n(\overline{y_m})})\}$ with $\overline{H_n}$ fresh variables of appropriate types, and $\overline{s_m}$ are distinct bound variables only if the selected equation has precursors.

[on] *outermost narrowing at nonvariable position*

$$\frac{G, \lambda\overline{x}.f(\overline{s_n}) \gg \lambda\overline{x}.t, G'}{G, \overline{\lambda\overline{x}.s_n \blacktriangleright \lambda\overline{x}.l_n}, \lambda\overline{x}.r \gg \lambda\overline{x}.t, G'}$$

if $f(\overline{l_n}) \to r$ is a fresh variant of an $\overline{x}$-lifted rule.

[c] [c]-*rule*

$$\frac{G, \lambda\overline{z}.s(\overline{y}) \blacktriangleright \lambda\overline{z}.X(\overline{y}), G', \lambda\overline{x}.X(\overline{t}) \gg \lambda\overline{x}.u, G''}{G, \lambda\overline{z}.s(\overline{y}) \blacktriangleright \lambda\overline{z}.X(\overline{y}), G', \lambda\overline{x}.s(\overline{t}) \gg \lambda\overline{x}.u, G''}$$

if $\lambda\overline{x}.u$ is rigid and $\mathcal{V}(s) \cap \{\overline{z}\} = \emptyset$.

Cooperative Constraint Functional Logic Programming

Mircea Marin, Tetsuo Ida
Institute of Information Sciences and Electronics
University of Tsukuba,
1-1-1 Tennoudai, Tsukuba, Ibaraki 305-8573, Japan
mmarin@score.is.tsukuba.ac.jp, ida@score.is.tsukuba.ac.jp

Taro Suzuki
Research Institute of Electric Communication,
Tohoku University, 980-8577 Sendai,Japan
taro@nue.riec.tohoku.ac.jp

Abstract

We describe the current status of the development of CFLP, *a system which aims at the integration of the best features of functional logic programming (FLP), cooperative constraint solving (CCS), and distributed constraint solving. FLP provides support for defining one's own abstractions (user-defined functions and predicates) over a constraint domain in an easy and comfortable way, whereas CCS is employed to solve systems of mixed constraints by iterating specialized constraint solving methods in accordance with a well defined strategy.*

CFLP *is a distributed implementation of a cooperative constraint functional logic programming scheme obtained from the integration of higher-order lazy narrowing for functional logic programming with cooperative constraint solving. The implementation takes advantage of the existence of several constraint solving resources located in a distributed environment, which communicate asynchronously via message passing.*

1 Introduction

Integration of declarative programming paradigms into a unified framework that captures the best features of each of them has attracted much interest during the last decade. Of particular interest is the design and implementation of a system based on a clean combination of functional logic programming (FLP) with cooperative constraint solving (CCS).

CFLP [4] is an experimental system which integrates higher-order functional logic programming with cooperative constraint solving. Its computational model combines higher-order lazy narrowing with the principle of solving systems of mixed constraints with a cooperation of constraint solvers described by a given cooperation strategy. The system is implemented in *Mathematica* and consists of an interpreter based on a higher-order lazy narrowing calculus, and a cooperative constraint solving system.

Our paper is structured as follows. In Section 2 we illustrate the solving capabilities of CFLP. The language of CFLP—syntax and semantics—is outlined in Section 3. Section 4 describes the general architecture of the system and of its main components. Finally, in Section 5 we draw some conclusions and directions of further research.

2 Examples

We describe with two examples the solving capabilities of CFLP.

2.1 Electric Circuit Modeling

The first example shows how electric circuit layouts can be computed with CFLP. This example illustrates the expressive power of the FLP style extended with higher-order constructs such as λ-abstractions and function variables, and constraint solving capabilities for differential equations and systems of polynomial equations.

We first define a function spec which describes the behavior in time t of an electrical component as a function of the current $i[t]$ and voltage $v[t]$ in the circuit. The CFLP rules of spec correspond to a recursive definition, where the base case describes the behavior of elementary circuits such as resistors, capacitors, and inductors, and the inductive case describes the behavior of serial and parallel connections of electrical components.

The CFLP program is given below. We do not explain the underlying electronic laws since it should be easy to read them off from the program.

```
(* declare the CFLP type ElComp with
   associated data constructors
   res, ind, cap, serial, par *)
Constructor[ElComp = res[Float]
                   | ind[Float]
                   | cap[Float]
                   | serial[TyList[ElComp]]
                   | par[TyList[ElComp]]];
(* specify the CFLP program *)
Prog:= {
 spec[res[r],v:Float → Float,i] → True
      ⇐ λ[{t},v[t]] ≈ λ[{t},r * i[t]],
 spec[ind[l],v,i:Float → Float] → True
      ⇐ λ[{t},v[t]] ≈ λ[{t},l*i'[t]],
 spec[cap[c],v:Float → Float,i] → True
      ⇐ λ[{t},i[t]] ≈ λ[{t},c*v'[t]],
 spec[serial[{}],λ[{t},0],i] → True,
 spec[serial[[comp | comps]],v,i] → True ⇐
      {spec[comp,v1,i] ≈ True,
      spec[serial[comps],v2,i] ≈ True,
      λ[{t},v[t]] ≈ λ[{t},v1[t] + v2[t]]},
 spec[par[{}],v:Float → Float,λ[{t},0]] → True,
 spec[par[[comp | comps]],v,i] → True ⇐
      {spec[comp,v,i1] ≈ True,
      spec[par[comps],v,i2] ≈ True,
      λ[{t},i[t]] ≈ λ[{t},i1[t] + i2[t]]}}
```

The variables and operators can be type annotated, and a polymorphic type checker is integrated into the system to verify the type consistency of the CFLP goal and program. The universally quantified variables are underlined. Note the usage of the list construct in the recursive specification of serial and parallel connections of electrical components. The CFLP system recognizes the following list specifications:

- $\{t_1, \ldots, t_n\}$: the list consisting of components $t_1, \ldots, t_n$,
- $[\,h \mid tl\,]$: CFLP list with head h and tail tl in Prolog-style notation.

Consider the problem of finding the behavior in time of the current in a RLC circuit with $\mathtt{R} = 2$, an $\mathtt{L} = 1$ and $C = 1/2$ (see Fig. 1), under the restrictions that the voltage is constant in time and the current was initially set to 0.

In this case, the goal which we want to solve is

```
G := {spec[serial[{res[2], ind[1], cap[1/2]}],
     λ[{t},50],ī] ≈ True,
     i[0] ≈ 0}
```

The logical variable of the goal is i, and it is overlined. Note

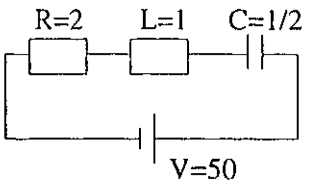


Fig. 1. RLC circuit

the usage of the expression `λ[{t},50]` for specifying that the voltage is constant (50 Ω) in time.

Now we can ask the CFLP system to solve the problem:

```
TSolve[G,Rules → Prog]
```

This call is similar to the `Solve` call of *Mathematica*, but `TSolve` has the specific option `Rules` which provides the functional logic program (i.e., conditional rewrite system) used upon solving the goal G.

The system computes the parametric solution $\{i \mapsto \lambda t.-k\, e^{-t} \sin(t)\}$ which is represented in *Mathematica* by `{i → λ[{t}, - k e^{-t} Sin[t]}`.

2.2 Program Calculation

This example illustrates the capabilities of a computing environment which can perform full higher-order pattern unification, and higher-order term rewriting.

Consider the problem of writing a program to check whether a list of numbers is *steep*, i.e, if every element of the list is greater than or equal to the average of the elements that follow it. A CFLP program that does this is:

```
Prog::={
   steep[{}] → True,
   steep[[a | x]] → And[a * len[x] ≥ sum[x], steep[x]],
   sum[{}] → 0, sum[[x | y]] → x + sum[y],
   len[{}] → 0, len[[x | y]] → 1 + len[y]}
```

`Prog` is modular and easy to understand, but very inefficient (quadratic complexity). It is possible—and desirable—to automatically compute an efficient (linear complexity) version `steepOptimal` of the function `steep` defined in `Prog`. To achieve this, we employ the fusion calculational rule

$$\frac{f[e] = e' \quad f[g[\underline{a}, \underline{ns}]] = h[\underline{a}, f[\underline{ns}]]}{f[\mathtt{foldr}[g, e, \underline{ns}]] = \mathtt{foldr}[h, e', \underline{ns}]} \qquad (1)$$

where $\mathtt{foldr} : (\alpha \times \beta \to \beta) \times \beta \times \mathtt{TyList}[\alpha] \to \beta$ is defined as usual:

$$\begin{aligned} \mathtt{foldr}[\underline{g}, \underline{e}, \{\}] &= \underline{e}, \\ \mathtt{foldr}[\underline{g}, \underline{e}, [\underline{n} \mid \underline{ns}]] &= \underline{g}[\underline{n}, \mathtt{foldr}[\underline{g}, \underline{e}, \underline{ns}]]. \end{aligned}$$

To see how this calculational rule can be employed, we first observe that the computation of $\mathtt{steep}[[n \mid ns]]$ with

Prog requires the computation of 3 additional quantities: steep[ns], sum[ns] and length[ns]. Thus, Prog actually specifies the computation of the function

$$\mathtt{f} = \lambda[\{ns\}, \mathtt{c3}[\mathtt{steep}[ns], \mathtt{sum}[ns], \mathtt{length}[ns]]]$$

where c3 is a ternary constructor.

Note that $\mathtt{f}[\underline{ns}] = \mathtt{f}[\mathtt{foldr}[g, e, \underline{ns}]]$, where $g = \mathtt{Cons}$ and $e = \{\}$. From the fusion calculational rule (1) results that we can compute f[ns] efficiently by computing foldr[h, e', ns] instead, where h is the solution of the equation

$$\lambda[\{n, ns\}, \mathtt{f}[g[n, ns]]] \approx \lambda[\{n, ns\}, \overline{\mathtt{h}}[n, f[ns]]]. \quad (2)$$

and $e' = f[\{\}] = \mathtt{c3}[\mathtt{True}, 0, 0]$. Then:

$$\mathtt{steepOpt} = \lambda[\{ns\}, \mathtt{sel\text{-}c3\text{-}1}[\mathtt{foldr}[\mathtt{h}, e', ns]]] \quad (3)$$

where sel-c3-1 is the data selector of the first argument of a term constructed with c3 (see Subsect. 3.1 for details). To solve equation (2) with CFLP we perform the following calls:

```
(* Declare data constructor c3 with
  associated type constructor 'Triple' *)
Constructor[Triple=c3[Bool,Float,Float]];

(* add definition of f to Prog *)
AppendTo[Prog, f[ns] → c3[steep[ns], sum[ns], len[ns]]];

(* compute h; during the computation
   'Plus','Times','Power','GreaterEqual','AND'
   are regarded as mere constructors *)
TSolve[λ[{n, ns}, f[[n | ns]]] ≈
        λ[{n, ns}, h̄[n, c3[steep[ns], sum[ns], len[ns]]]],
   Constructor →{Plus, Power,
                      Times, GreaterEqual, And},
   Rules → Prog]
```

CFLP yields the unique solution

```
{{h→ λ[{x$13, x$14}, c3[
   And[x$13 sel-c3-3[x$14] ≥ sel-c3-2[x$14]],
   sel-c3-1[x$14],
   x$13 + sel-c3-1[x$14], 1 + sel-c3-3[x$14]]}}
```

3 The Language

CFLP is a distributed software system for solving systems of equations over various constraint domains for which specialized constraint solvers are available.

3.1 Syntax

The language of CFLP is built up from the following symbols:

- Built-in constant symbols: 2/3, -2912, Pi, etc.[1],
- External symbols: the elements of a predefined set $\mathcal{F}_e = \{+, -, *, \ldots\}$,
- Built-in relational symbols:

  equational: $\approx$ (unoriented equality), $\rhd$ (oriented equality), $\doteq$ (unoriented strict equality) and $\gg$ (oriented strict equality)

  logical: $\{\cdot, \cdots\}$ (sequential AND), $||$ (sequential OR), and $\vee$ (parallel OR),
- Built-in type constructors:
  - base types: Int, Float, Compl, Bool,
  - TyList[α] (polymorphic list type) and {} (empty list constant),
- Data constructors: the elements of a set $\mathcal{F}_c$ which contains the built-in data constructors Cons, {} (for lists), and the data constructors defined with Constructor declarations,
- Defined function symbols: symbols of a set $\mathcal{F}_d$ which contains:
  - the data selectors introduced by the Constructor declarations,
  - the symbols defined by the rewrite rules of a given program
- Variables: symbols of a set $\mathcal{V}$ of *Mathematica* symbols with no predefined or given meaning.

The set $\mathcal{F} = \mathcal{F}_e \cup \mathcal{F}_c \cup \mathcal{F}_d$ is called the *signature* of CFLP. Next, we build the other syntactic objects of our language:

- Well-typed λ-term (or term): *Mathematica* representation of an expression built over the signature $\mathcal{F}$ and set of variables $\mathcal{V}$, but instead of Function we use λ. E.g., $\lambda[\{x, y\}, x * y]$ is the λ-term for the function which computes the product of its arguments. The set of terms is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$.
- Type-annotated symbol: an expression of the form $v : \tau$ where v is a variable or function symbol and τ is a type expression.
- Equation: an expression of the form $s \cong t$ where $\cong \in \{\approx, \rhd, \doteq, \gg\}$ and s, t are terms of the same type.
- Goal: either an equation, or of the form $\{g_1, \ldots, g_n\}$ (sequential AND goal), $g_1 || \ldots || g_n$ (sequential OR goal), or $g_1 \vee \ldots \vee g_n$ (parallel OR-goal), where $g_1, \ldots, g_n$ are goals.

[1]These are the constants recognized by *Mathematica*

- Rewrite rule: expression of the form $f[l_1,\ldots,l_n] \to r \Leftarrow c$ where $f[l_1,\ldots,l_n]$, r are terms of the same base type, $f[l_1,\ldots,l_n]$ is a higher-order pattern, and c is a goal. f is the *defined* function, and c the *condition* of the rewrite rule.

- Program: a (possibly empty) set of rewrite rules.

A `Constructor` declaration has the form

`Constructor[`$\langle$*constr-spec*$_1\rangle,\ldots,\langle$*constr-spec*$_n\rangle$`]`

where

$\langle$*constr-spec*$\rangle$ ::= $\langle$*type-spec*$\rangle$"="$\langle$*data-spec*$_1\rangle$"|" ... "|"$\langle$*data-spec*$_n\rangle$
$\langle$*type-spec*$\rangle$ ::= $\langle$*type-name*$\rangle$ | $\langle$*type-name*$\rangle[\alpha_1,\ldots,\alpha_n]$
$\langle$*data-spec*$\rangle$ ::= $\langle$*type-constr*$\rangle$ | $\langle$*type-constr*$\rangle[\tau_1,\ldots,\tau_n]$
$\alpha_1,\ldots,\alpha_n$ distinct type variables
$\tau_1,\ldots,\tau_n$ type expressions

Example 1 We can declare binary trees with nodes of type α by:

```
Constructor[BTree[α] =
            BNil |
            T2[α,BTree[α],BTree[α]]
```

This declaration introduces the type constructor `BTree` together with its associated data constructors

`BNil`:$\forall\alpha$.`BTree`$[\alpha]$
`T2` : $\forall\alpha.\alpha \times$ `BTree`$[\alpha] \times$ `BTree`$[\alpha] \to$ `BTree`$[\alpha]$.

In addition, the data-selectors `sel-T2-1`, `sel-T2-2` and `sel-T2-3` for the data constructor `T2` are defined implicitly. □

A `Constructor` call extends the set $\mathcal{F}_c$ of the language of CFLP with the newly declared data constructors, and the set $\mathcal{F}_d$ of defined symbols of the language of CFLP with the selectors of the non-constant constructors. For example, if c is a newly-declared n-ary data constructor, then the data selectors `sel-`c`-1`$,\ldots,$ `sel-`c`-`n are added to $\mathcal{F}_d$.

A program $\mathcal{R}$ adds the set of symbols

$$\{f \mid \exists(f[l_1,\ldots,l_n] \to r \Leftarrow c) \in \mathcal{R}\}$$

to the set $\mathcal{F}_d$ of defined symbols of the CFLP language. We denote by $Subst(\mathcal{F},\mathcal{V})$ the set of substitutions over a signature $\mathcal{F}$ and set of variables $\mathcal{V}$, by $\mathcal{D}(\theta)$ the domain of a substitution θ, and by ϵ the empty substitution.

3.2 Semantics

The semantics of the external symbols in $\mathcal{F}_e$ and of the data constructors in $\mathcal{F}_c$ is given by a predefined constraint domain $\mathcal{X}$ equipped with various solvers for solving systems of equational constraints.

A CFLP program $\mathcal{R}$ extends the language of the constraint domain $\mathcal{X}$ with symbols defined in a functional-logic programming style, and to give them a meaning. A program $\mathcal{R}$ induces a rewrite relation $\longrightarrow_{\mathcal{X},\mathcal{R}}$ on $\mathcal{X}$, where pattern matching is defined modulo the equality relation on $\mathcal{X}$. We define the semantics of the equational symbols of CFLP as follows:

- $\mathcal{X},\mathcal{R} \models s \approx t$ if $s \longrightarrow^*_{\mathcal{X},\mathcal{R}} u \;{}^*_{\mathcal{X},\mathcal{R}}\!\longleftarrow t$ for some term u,

- $\mathcal{X},\mathcal{R} \models s \rhd t$ if $s \longrightarrow^*_{\mathcal{X},\mathcal{R}} t$,

- $\mathcal{X},\mathcal{R} \models s \doteq t$ if $s \longrightarrow^*_{\mathcal{X},\mathcal{R}} u \;{}^*_{\mathcal{X},\mathcal{R}}\!\longleftarrow t$ for some constructor term u,

- $\mathcal{X},\mathcal{R} \models s \gg t$ if t is a constructor term and $s \longrightarrow^*_{\mathcal{X},\mathcal{R}} t$.

3.3 The Problem

CFLP is designed to solve problems of the following type:

Given a program $\mathcal{R}$ and an equation $s \cong t$ with $\cong \in \{\approx, \rhd, \doteq, \gg\}$,

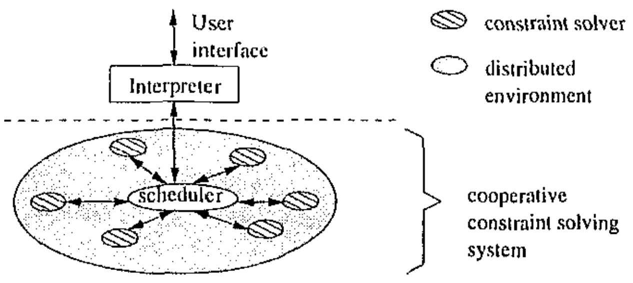
Compute $\theta \in Subst(\mathcal{F},\mathcal{V})$ such that $\mathcal{X},\mathcal{R} \models s\theta \cong t\theta$.

We call such a θ an *$\mathcal{R}$-solution* of $s \cong t$. This problem is extended to goals in the natural way. We denote by $\mathcal{U}_{\mathcal{R}}(G)$ the set of $\mathcal{R}$-solutions of a goal G. In general, we are interested to compute *$\mathcal{R}$-normalized* solutions of a goal G, i.e. substitutions $\theta \in \mathcal{U}_{\mathcal{R}}(G)$ such that $X\theta$ is $\mathcal{R}$-normalized for all $X \in \mathcal{D}(\theta)$. We denote the set of $\mathcal{R}$-normalized solutions of G by $\mathcal{U}^n_{\mathcal{R}}(G)$.

To solve such problems, we have designed and implemented a computation model which integrates two operational principles: lazy narrowing for conditional pattern rewrite systems, and concurrent constraint solving. Lazy narrowing solves equations containing symbols from $\mathcal{F}_c \cup \mathcal{F}_d$, whereas concurrent constraint solving is employed to solve systems of equations over $\mathcal{X}$.

4 System Structure

The combination of lazy narrowing with CCS is reflected in the structure of the system: it consists of a functional logic interpreter based on lazy narrowing, which is integrated with a distributed implementation of a cooperative constraint solving system (see below).



4.1 The Interpreter

The interpreter of CFLP is based on a calculus $\mathcal{C}$ which is an extension of a pure lazy narrowing calculus $\mathcal{K}$ for conditional pattern rewrite systems with inference rules to recognize and process equations with external symbols. $\mathcal{C}$ can be described by a set of inference rules which act on *states* of the form $\langle W \mid G \mid Store\rangle$ where W is a set of variables, G is a CFLP goal, and *Store* is a set of constraints $s \approx t$ collected so far. A state $\langle W \mid G \mid Store\rangle$ is interpreted as

$$[\![\langle W \mid G \mid Store\rangle]\!] = \{\gamma \mid \gamma \in \mathcal{U}_{\mathcal{R}}(G), \gamma\lceil_W \text{ is } \mathcal{R}\text{-normalized, and } \gamma \in \mathcal{U}_{\mathcal{R}}(e) \text{ for all } e \in Store\}.$$

The inference rules of the calculus $\mathcal{C}$ are presented as relations of the form

$$\langle W \mid G \mid Store\rangle \overset{\mathcal{C}}{\Longrightarrow}_{\theta} \langle W' \mid G' \mid Store'\rangle$$

where θ is a substitution, called *computed* substitution. Such a relation is called $\mathcal{C}$*-step*. The interpreter computes $\mathcal{C}$*-derivations*, i.e. sequences

$$\langle W_0 \mid G_0 \mid Store_0\rangle \overset{\mathcal{C}}{\Longrightarrow}_{\theta_1} \ldots \overset{\mathcal{C}}{\Longrightarrow}_{\theta_N} \langle W_N \mid G_N \mid Store_N\rangle$$

of $\mathcal{C}$-steps, abbreviated

$$\langle W_0 \mid G \mid Store\rangle \overset{\mathcal{C}}{\Longrightarrow}^{*}_{\theta_1 \ldots \theta_N} \langle W_N \mid G_N \mid Store_N\rangle.$$

The $\mathcal{C}$-derivations which are useful in computing a representation of $\mathcal{U}^{n}_{\mathcal{R}}(G)$ are the so-called $\mathcal{C}$-refutations. A $\mathcal{C}$-*refutation* is a finite $\mathcal{C}$-derivation of maximum length

$$\langle \mathcal{V}(G) \mid G \mid \{\}\rangle \overset{\mathcal{C}}{\Longrightarrow}^{*}_{\theta} \langle W' \mid G' \mid Store\rangle \qquad (4)$$

where $\mathcal{V}(G)$ is the set of free variables in G. The set of answers computed by CFLP for a given goal G is

$$Answ_{\mathcal{R}}(G) = \{\ \langle \theta, G', Store\rangle \mid \exists\, \mathcal{C}\text{-refutation } \langle \mathcal{V}(G) \mid G \mid \{\}\rangle \overset{\mathcal{C}}{\Longrightarrow}^{*}_{\theta} \langle W' \mid G' \mid Store\rangle\}$$

The calculus $\mathcal{C}$ is designed to satisfy the following two conditions:

soundness: For any $\langle \theta, G', Store\rangle \in Answ_{\mathcal{R}}(G)$ and any $\mathcal{R}$-solution γ of G' and *Store*, we have $\theta\gamma \in \mathcal{U}_{\mathcal{R}}(G)$

completeness: For any $\gamma \in \mathcal{U}^{n}_{\mathcal{R}}(G)$ there is a $\langle \theta, G', Store\rangle \in Answ_{\mathcal{R}}(G)$ such that $\gamma = \theta\gamma'\ [\mathcal{V}(G)]$ for some $\mathcal{R}$-solution γ' of G' and *Store*.

A $\mathcal{C}$-refutation is constructed in two stages:

1. *Lazy Narrowing Stage:* starting with the state $\langle \mathcal{V}(G) \mid G \mid \{\}\rangle$, the goal G is narrowed until a goal made of equations which can not be narrowed anymore. The equations which can not be narrowed anymore are either:
   - *constraints*, i.e. equations over the constraint domain $\mathcal{X}$, or
   - certain *flex/flex equations*, i.e. equations between terms of the form

     $$\lambda[\{x_1, \ldots, x_p\}, X[s_1, \ldots, s_m]]$$

     with X a free variable.

     Remark: The design a calculus to solve all flex/flex equations is unrealistic, since full higher-order unification is highly intractable [1]. Our design of $\mathcal{C}$ is based on Huët's idea of pre-unification [3].
2. *Constraint Solving Stage:* the constraints produced during the lazy narrowing stage are solved with a cooperative constraint solver.

4.1.1 The Lazy Narrowing Stage

The rules of $\mathcal{C}$ which realize this stage are:

[V] *parallel OR*
$\langle W \mid G_1 \vee \ldots \vee G_n \mid Store\rangle \overset{\mathcal{C}}{\Longrightarrow}_{\varepsilon} \langle W \mid G_k \mid Store\rangle$
where $k \in \{1, \ldots, n\}$

[||] *sequential OR*
$\langle W \mid G_1 \| \ldots \| G_n \mid Store\rangle \overset{\mathcal{C}}{\Longrightarrow}_{\varepsilon} \langle W \mid G_k \mid Store\rangle$
where $k \in \{1, \ldots, n\}$

The difference between these rules is that the nondeterminism due to selection of G_k is explored by breadth-first search for [V] and by depth-first search for [||].

[{·}] *sequential AND*
$\langle W \mid \{G_1, \ldots, G_n\} \mid Store\rangle \overset{\mathcal{C}}{\Longrightarrow}_{\theta} \langle W' \mid \{G'_1, \ldots, G'_n\} \mid Store'\rangle$
if $\langle W \mid G_k \mid Store\rangle \overset{\mathcal{C}}{\Longrightarrow}_{\theta} \langle W' \mid G'_k \mid Store'\rangle$ and $G'_i = G_i\theta$ for all $i \neq k$.

[xf] *constraint accumulation*
$\langle W \mid s \cong t \mid Store\rangle \overset{\mathcal{C}}{\Longrightarrow}_{\varepsilon} \langle W \mid \{\} \mid Store \cup \{s \approx t\}\rangle$ if $s \approx t$ is a constraint

[xi] *imitation for external symbols*

$$\frac{\langle W \mid \lambda[\{\overline{x_p}\}, f[\overline{s_m}] \cong \lambda[\{\overline{x_p}\}, t] \mid Store\rangle \xRightarrow{\mathcal{C}}_{\epsilon}}{\langle W \mid \lambda[\{\overline{x_p}\}, s_m] \cong \lambda[\{\overline{x_p}\}, Y_m[\overline{x_p}]], \; \lambda[\{\overline{x_p}\}, Y[\overline{x_p}]] \cong \lambda[\{\overline{x_p}\}, t] \mid Store\rangle}$$

with $Y_1, \ldots, Y_m$ fresh variables and $\cong \in \{r, r^{-1} \mid r \in \{\approx, \doteq, \rhd, \gg\}\}$ if $f \in \mathcal{F}_e$.

[flp] *lazy narrowing step*

These steps are governed by a given lazy narrowing calculus $\mathcal{K}$

$\langle W \mid s \cong t \mid Store\rangle \xRightarrow{\mathcal{C}}_{\theta} \langle W \mid G \mid Store'\rangle$
if $s \cong t \xRightarrow{\mathcal{K}}_{\theta} G'$ is a $\mathcal{K}$-step and $Store' = \{s\theta \approx t\theta \mid (s \approx t) \in Store\}$.

The interpreter of CFLP can make use of different built-in lazy narrowing calculi $\mathcal{K}$, depending on the preference of the user. We have designed and implemented sound and complete lazy narrowing calculi for pattern rewrite systems (PRS for short), left-linear confluent fully-extended PRS, and left-linear constructor fully extended PRS (see [5]). Moreover, we have extended these calculi to handle strict equations and conditional PRSs. Some of these calculi are higher-order generalizations of the deterministic refinements of the lazy calculus LNC [7, 6].

4.1.2 The Constraint Solving Stage

The constraints accumulated in the constraint store during the lazy narrowing stage are submitted to be solved with a cooperative constraint solving system. Formally, this stage can be described by

[cs] *constraint solve*

$\langle W \mid G \mid Store\rangle \xRightarrow{\mathcal{C}}_{\theta} \langle W' \mid G\theta \mid Store'\rangle$
if $\langle \theta, Store'\rangle \in \mathcal{S}(\langle\{\}, Store\rangle)$ (see next subsection) and W' are the free variables in $\{X\theta \mid X \in W\}$.

4.2 The Cooperative Constraint Solving Component

We assume given a constraint domain $\mathcal{X}$ over a signature $\mathcal{F}_e$ of *external* operators, and a set of specialized constraint solvers $CS_1, \ldots, CS_n$. Each solver is a function

$$CS_k : \mathcal{E}qs^{(k)}(\mathcal{F}_e, \mathcal{V}) \to \mathcal{P}_{\text{fin}}(Subst(\mathcal{F}_e, \mathcal{V}) \times \mathcal{E}qs^{(k)}(\mathcal{F}_e, \mathcal{V}))$$

where $\mathcal{E}qs^{(k)}(\mathcal{F}_e, \mathcal{V}) \subset \mathcal{P}_{\text{fin}}(\mathcal{E}q(\mathcal{F}_e, \mathcal{V}))$ is the set of constraints that can bw solved with CS_k, $Subst(\mathcal{F}_e, \mathcal{V})$ is the set of idempotent substitutions over $\mathcal{F}_e$. The individual solvers are canonical simplifiers, i.e. if $CS_k(S) = \{\langle\theta_i, S_i\rangle \mid 1 \le i \le N\}$ then

- γ is a solution of S iff γ is of the form $\theta_p\gamma'_p$ for some $p \in \{1, \ldots, N\}$ and solution γ'_p of S_p.
- every S_i is a CS_k-canonical form, i.e.

$$CS_k(S_i) = \{\langle\{\}, S_i\rangle\}.$$

CS_k is extended to an operator CS_k^e on $\mathcal{P}_{\text{fin}}(Subst(\mathcal{F}_e, \mathcal{V}) \times \mathcal{P}_{\text{fin}}(\mathcal{E}q(\mathcal{F}_e, \mathcal{V})))$ defined by

$$\begin{aligned} CS_k^e(\{\langle\gamma_p, S_p\rangle \mid 1 \le p \le N\} = \textstyle\bigcup_{p=1}^{N} \{\langle\gamma_k\theta', S''\gamma_k \cup S'\rangle \mid \\ S' = S \cap \mathcal{E}qs^{(k)}(\mathcal{F}_e, \mathcal{V}), S'' = S - S', \\ \langle\theta', S'\rangle \in CS_k(S_p)\}. \end{aligned}$$

The operational principle of the cooperative constraint solving component is defined by a method $\mathcal{S}$ which describes how the computations of $CS_1, \ldots, CS_n$ are combined. We call $\mathcal{S}$ *cooperation strategy*, and define $\mathcal{S}(CS_1, \ldots, CS_n)$ as the fixed-point of $CS_n^e \circ \cdots \circ CS_1^e$. This strategy has been proposed and extensively investigated by Hong [2] in the framework of cooperative CLP. Obviously, when defined, the result of $\mathcal{S}$ is a CS_k-canonical form for all $1 \le k \le N$.

Example 2 Assume we want to solve

$$S = \{\lambda t.y'(t) \approx \lambda t.k^2 y(t),\ y(0) \approx 1,\ y(2) \approx 3, y(r) \approx 5\}$$

in variables y, k, r, by using a cooperation of 3 solvers: a solver CS_1 for differential equations, a solver CS_2 for systems of monomial equations, and a solver CS_3 for equations with invertible functions. Then
$CS_1^e(\langle\{\}, S\rangle) = S_1$ where $S_1 = \{\langle\{y \mapsto \lambda t.c\, e^{k^2 t}\}, \{c \approx 1,\ c\ e^{2\,k^2} \approx 3,\ c\ e^{r\,k^2} \approx 5\}\rangle\}$ with c a new variable; $CS_2^e(S_1) = S_2$ where $S_2 = \{\langle\{y \mapsto e^{k^2 t}, c \mapsto 1\}, \{e^{2\,k^2} \approx 3,\ e^{r\,k^2} \approx 5\}\rangle\}$; $CS_3^e(S_2) = S_3$ where $S_3 = \{\langle\{y \mapsto \lambda t.e^{k^2 t}, c \mapsto 1\}, \{2\,k^2 \approx \log(3), r\ k^2 \approx \log(5)\}\rangle\}$; $CS_1^e(S_3) = S_3$; $CS_2^e(S_3) = S_4$ where $S_4 = \{\langle\{y \mapsto \lambda t.e^{t\ \log(3)/2}, c \mapsto 1, k \mapsto -\sqrt{\log(3)/2}\}, \{-r\ \log(3)/2 \approx \log(5)\}\rangle, \langle\{y \mapsto \lambda t.e^{t\ \log(3)/2}, c \mapsto 1, k \mapsto \sqrt{\log(3)/2}\}, \{r\ \log(3)/2 \approx \log(5)\}\rangle\}$; $CS_1^e(S_4) = CS_2^e(S_4) = S_4$; and $CS_3^e(S_4) = S_5$ where $S_5 = \{\langle\{y \mapsto \lambda t.e^{t\ \log(3)/2}, c \mapsto 1, k \mapsto -\sqrt{\frac{\log(3)}{2}}, r \mapsto -\frac{2\log(5)}{\log(3)}\}, \{\}\rangle, \langle\{y \mapsto \lambda t.e^{t\ \log(3)/2}, c \mapsto 1, k \mapsto \sqrt{\frac{\log(3)}{2}}, r \mapsto \frac{2\ \log(5)}{\log(3)}\}, \{\}\rangle\}$. In this example, the solution S_5 of S is computed in 9 steps.

To compute $CS_k^e(S)$ we must call CS_k n_k times where n_k is the number of elements in S, and these calls can be realized in parallel. To take advantage of this fact, we have implemented a distributed constraint solving system consisting of

- several instances of solvers running on various machines, and
- a scheduler, which implements the strategy $\mathcal{S}$ by fairly allocating the constraint solvers available to eventually solve each element of S.

The general structure of the distributed constraint solving subsystem of CFLP is depicted in Fig. 2.

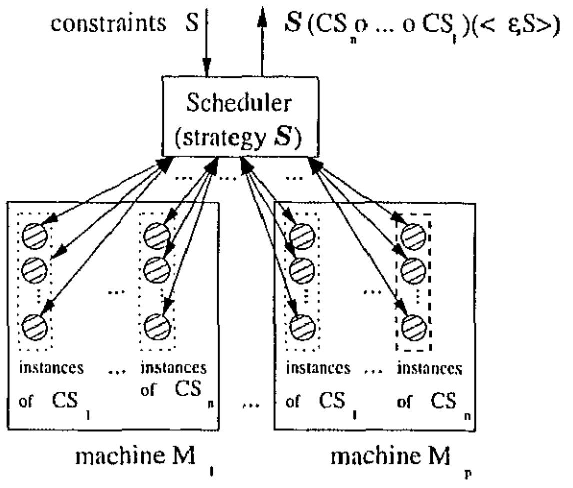


Fig. 2. CFLP: **the cooperative constraint solving component**

The current version of CFLP has integrated four types of constraint solvers:

1. CS_1 : solver which can solve algebraic equations with invertible functions, and yields solutions in terms of formal inverse functions.
2. CS_2 : solves systems of equations between multivariate polynomials over algebraic extensions of the domain of complex numbers,
3. CS_3 : solver for differential equations over algebraic extensions of $\mathbb{C}$,
4. CS_4 : solver for partial differential equations over algebraic extensions of $\mathbb{C}$.

All the components of the cooperative constraint solving system—scheduler and constraint solvers—are implemented in *Mathematica* [8] as *MathLink*-compatible processes which communicate asynchronously by message-passing over *MathLink* connections.

CFLP makes distinction between two types of solving resources:

1. local constraint solvers: these are solvers which run locally as sub-processes of a CFLP session,
2. remote constraint solvers: these solvers are started to run on various machines from outside the CFLP session, and can be shared between different CFLP sessions. The distribution of CFLP provides shell scripts to start and stop running remote constraint solvers.

The user can adjust the computation session by specifying

- the machines $M_1, \ldots, M_p$ on which to connect to the remote constraint solvers, and
- the number of local constraint solvers.

5 Conclusion

The system described in this paper is based on a computational model which integrates lazy narrowing for conditional PRS with CCS.

Currently, only a few theoretical results have been generalized to the conditional case. We will continue our research to design efficient and complete calculi for various classes of conditional PRS.

The main intention of our system CFLP was to prove the suitability of our evolvable distributed model for cooperative constraint solving. We didn't focus yet on the design of an efficient implementation.

References

[1] W. Gould. *A matching procedure for ω-order logic*. Scientific Report 4. Air Force Cambridge Research Laboratories, 1966.

[2] H. Hong. Non-linear Constraints Solving over Real Numbers in Constraint Logic Programming (Introducing RISC-CLP). Technical Report 92-08, RISC-Linz, Castle of Hagenberg, Austria, 1992.

[3] G. Huët. A Unification Algorithm for Typed λ-Calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[4] M. Marin. *Functional Logic Programming with Distributed Constraint Solving*. PhD thesis, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Schloss Hagenberg, April 2000.

[5] M. Marin, T. Ida, and T. Suzuki. On Reducing the Search Space of Higher-Order Lazy Narrowing. In A. Middeldorp and T. Sato, editors, *FLOPS'99*, volume 1722 of *LNCS*, pages 225–240. Springer-Verlag, 1999.

[6] A. Middeldorp and S. Okui. A Deterministic Lazy Narrowing Calculus. *Journal of Symbolic Computation*, 25(6):733–757, 1998.

[7] A. Middeldorp, S. Okui, and T. Ida. Lazy Narrowing: Strong Completeness and Eager Variable Elimination. *Theoretical Computer Science*, 167(1,2):95–130, 1996.

[8] S. Wolfram. *The Mathematica Book*. Third Edition. Wolfram Media and Cambridge University Press, 1996.

LAZY NARROWING CALCULI FOR PATTERN REWRITE SYSTEMS

Mircea MARIN
Tetsuo IDA
Institute of Information Sciences and Electronics
University of Tsukuba, Tsukuba 305-8573, Japan
Taro SUZUKI
Institute of Electric Communication
Tohoku University, Sendai 980-8577, Japan

Presented at the 2nd International Workshop on
SYMBOLIC AND NUMERIC ALGORITHMS FOR SCIENTIFIC COMPUTATION
Timişoara, October 4–6, 2000

Abstract. Higher-order lazy narrowing (HLN for short) is a computational model for higher-order functional logic programming. It can be viewed as an extension of first-order lazy narrowing with inference rules to solve equations involving λ-abstractions and higher-order variables.

A common feature of the HLN calculi proposed so far is the high nondeterminism between the inference rules designed to solve equations which involve higher-order variables. In this paper we present various optimizations of HLN towards more deterministic versions. The optimizations are defined for classes of higher-order functional logic programs which are useful for programming purposes. Our work draws on two sources: the calculus LN for pattern rewrite systems [12] and the first-order lazy narrowing calculus LNC and its optimizations [7].

AMS Subject Classification: 68N18, 03B15.

Keywords and phrases: higher-order logic, functional logic programming, lazy narrowing

1 INTRODUCTION

Recent years have witnessed a growing interest in extending lazy narrowing with higher-order constructs, in an attempt to improve the expressive power of functional logic programming (FLP). The capability to handle λ-abstractions and function variables is desirable in FLP, mainly because one can use the abstraction principle and can quantify over predicate and function symbols. The cost of such an extension is the high nondeterminism between the inference rules designed to solve equations with function variables. This problem has first been observed in the context of solving higher-order equations in empty equational theories [1], where it is avoided by adopting the idea of pre-unification [3] instead of full higher-order unification. The main idea of pre-unification is to compute pre-unifiers instead of unifiers, by avoiding to solve the problematic flex/flex equations.

Compared with higher-order pre-unification, HLN as operational semantics for higher-order FLP adds one more complication: the high nondeterminism of solving equations where at least one side is a *flex term*, i.e. a term of the form $\lambda x_1, \ldots, x_m.X(s_1, \ldots, s_n)$ with X a free variable. The nondeterminism between the inference rules of HLN can be reduced by restricting it to particular classes of higher-order functional logic programs (i.e., higher-order rewrite systems) which guarantee that the resulted calculus is complete. For programming purposes, the classes of higher-order functional logic programs must be chosen to be expressive enough.

Recently, various specializations of higher-order lazy narrowing to particular classes of higher-order rewrite systems have been proposed (see, e.g. [14, 12, 6, 5]) in an attempt to define a suitable operational model for higher-order FLP.

In the sequel we give a brief account of the higher-order lazy narrowing calculi designed by us for higher-order FLP. Our theoretical framework are simply-typed term algebras where functional logic programs are represented by pattern rewrite systems.

2 PRELIMINARIES

2.0.1 Notions and Notation.

We assume given a set $\mathcal{F}$ of operators with associated arities, and a set $\mathcal{V}$ of variables. The set $\mathcal{T}(\mathcal{F}, \mathcal{V})$ of terms built with symbols from $\mathcal{F}$ and variables from $\mathcal{V}$ is defined in the usual way. We denote by *Vars*(t) the set of variables which appear in a term t, and by *Subst*$(\mathcal{F}, \mathcal{V})$ the set of finite bindings of variables to terms, We call such bindings *substitutions*, and denote them with greek letters. If t is a term and $\theta = \{X_1 \mapsto t_1, \ldots, X_k \mapsto t_k\}$ a substitution, then $t\theta$ is the term obtained by replacing the occurrences of $X_1, \ldots, X_k$ in t with the terms $t_1, \ldots, t_k$ respectively. A *fresh variant* of a syntactic expression e (term, equation, goal, rewrite rule, ...) is an expression $e\theta$ obtained by applying a fresh variable renaming θ to all variables in e. A fresh variable renaming is a substitution of the form $\{X_1 \mapsto Y_1, \ldots, X_p \mapsto Y_p\}$ with $Y_1, \ldots, Y_p$ distinct variables which didn't occur in the expressions constructed so far.

Many problems from mathematics and sciences can be expressed as *E-unification problems,* i.e:

Given a finite axiomatization $E = \{l_1 \approx r_1, \ldots, l_n \approx r_n\}$ of an equational theory, and a system of equations $G = (s_1 \approx t_1, \ldots, s_m \approx t_m)$

Solve G in the theory axiomatized by E. That is, we want to compute the solutions of G, i.e. bindings θ of variables to terms, such that the equality $s_k\theta \approx t_k\theta$ can be proved modulo E for all $k \in \{1, \ldots, n\}$. (Notation $E \vdash s_k\theta \approx t_k\theta$.) The set of solutions of G is denoted by $\mathcal{U}_{\mathcal{R}}(G)$.

A standard axiomatization of the provability relation $E \vdash s \approx t$ is:

$$\frac{l \approx r \in E}{E \vdash l \approx r} \qquad \frac{}{E \vdash t \approx t} \qquad \frac{E \vdash s \approx t}{E \vdash t \approx s} \qquad \frac{E \vdash s \approx t \quad E \vdash t \approx u}{E \vdash s \approx u}$$

$$\frac{E \vdash s \approx t}{E \vdash s\theta \approx t\theta} \qquad \frac{E \vdash s_1 \approx t_1 \ \ldots \ E \vdash s_n \approx t_n}{E \vdash f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n)}$$

It is often the case that E can be represented as a term rewrite system (TRS for short), i.e. as a set of oriented equations of the form $f(l_1, \ldots, l_n) \to r$, abbreviated $f(\overline{l_n}) \to r$, where $f(\overline{l_n})$ and r are terms such that $\mathit{Vars}(r) \subseteq \mathit{Vars}(f(\overline{l_n}))$. Such equations are called *rewrite rules.* Functional logic programming (FLP for short) is concerned with solving E-unification problems in theories axiomatized by TRS.

As logic programs are represented by sets of Horn clauses, functional logic programs are represented by TRSs. In the presence of a TRS, the set $\mathcal{F}$ is split into two disjoint sets: the set $\mathcal{F}_d = \{f \mid \exists f(\overline{l_n}) \to r \in \mathcal{R}\}$ of *defined symbols*, and the set $\mathcal{F}_c = \mathcal{F} - \mathcal{F}_d$ of *constructors.*

A TRS $\mathcal{R}$ induces a rewriting relation $\to_{\mathcal{R}}$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$. Rewriting is based on the idea of replacing "equals by equals", and is defined by: $s \to_{\mathcal{R}} t$ if there exists a position p in s and a rewrite rule $(l \to r) \in \mathcal{R}$ such that $s|_p = l\sigma$ and $t = s[r\theta]$. Among the desirable properties of a TRS we mention

termination: there are no infinite rewrite sequences $s_1 \to_{\mathcal{R}} s_2 \to_{\mathcal{R}} \cdots$,

confluence: if $\mathcal{R} \vdash s \approx t$ then there exists u such that $s \to_{\mathcal{R}}^* u$ and $t \to_{\mathcal{R}}^* u$ (abbr. $s \downarrow_{\mathcal{R}} t$). Here, $\to_{\mathcal{R}}^*$ denotes the reflexive-transitive closure of $\to_{\mathcal{R}}$

left-linearity if $f(\overline{l_n}) \to r \in \mathcal{R}$ then all variables in $f(\overline{l_n})$ are distinct.

In FLP we are usually interested to compute $\mathcal{R}$-normalized solutions of a given equational goal $(s_1 \approx t_1, \ldots, s_n \approx t_n)$ (abbreviated by $\overline{s_n \approx t_n}$.), i.e. substitutions $\theta = \{X_1 \to t_1, \ldots, X_p \mapsto t_p\} \in \mathcal{U}_{\mathcal{R}}(G)$ such that $t_1, \ldots, t_p$ are $\mathcal{R}$-normal forms (i.e. they can not be rewritten). We denote by $\mathcal{U}_{\mathcal{R}}^n(G)$ the set of $\mathcal{R}$-normalized solutions of a goal G.

The most popular operational principle of FLP is narrowing [4]. The FLP languages available nowadays are based on calculi which realize the narrowing principle. These calculi are required to satisfy the following requirements:

Given a FLP program $\mathcal{R}$ and an equational goal $G = \overline{s_n \approx t_n}$

Compute a set $Ans_{\mathcal{R}}(G) \subseteq Subst(\mathcal{F}, \mathcal{V})$ such that

Soundness: $Ans_{\mathcal{R}}(G) \subseteq \mathcal{U}_{\mathcal{R}}(G)$,
Completeness: $\forall \gamma \in \mathcal{U}_{\mathcal{R}}^n(G). \exists \theta \in Ans_{\mathcal{R}}(G).\ \theta \leq \gamma\ [Vars(G)]$.

2.1 The Calculus LNC

Of particular interest is the lazy narrowing calculus LNC [8, 7]: it consists of five inference rules (see Fig. 1) by which we can refute goals, i.e. compute derivations of the form $G \overset{\text{LNC}}{\Longrightarrow}{}_{\theta}^{*} \Box$ where G is a goal consisting of unoriented equations, θ is a substitution, and $\Box$ is the empty goal (with no equations).

LNC has the following important properties:

Soundness: If $G \overset{\text{LNC}}{\Longrightarrow}{}_{\theta}^{*} \Box$ then $\theta \in \mathcal{U}_{\mathcal{R}}(G)$,

Completeness: $\forall \gamma \in \mathcal{U}_{\mathcal{R}}^n(G).\ \exists\ G \overset{\text{LNC}}{\Longrightarrow}{}_{\theta}^{*} \Box$ such that $\theta \leq \gamma\ [Vars(G)]$.

In addition, LNC can be optimized in several ways [7], and therefore it is a suitable computation model for functional logic programming. E.g., completeness is preserved if we compute only with LNC-derivations which respect strategy S_{left}, i.e. the leftmost equation is always selected.

Thus, if we define $Ans_{\mathcal{R}}^{\text{LNC}}(G) = \{\theta \mid G \overset{\text{LNC}}{\Longrightarrow}{}_{\theta}^{*} \Box\}$ then $Ans_{\mathcal{R}}^{\text{LNC}}(G)$ is a complete subset of $\mathcal{U}_{\mathcal{R}}^n(G)$.

$\simeq \in \{\approx, \approx^{-1}\}$

[d] $G, f(\overline{s_n}) \approx f(\overline{t_n}), G' \overset{\text{LNC}}{\Longrightarrow}_{[d],\varepsilon} G, \overline{s_n \approx t_n}, G'$

[i] $G, X \simeq f(\overline{s_n}), G' \overset{\text{LNC}}{\Longrightarrow}_{[i],\theta} G\theta, \overline{X_n \simeq s_n\theta}, G'\theta$
where $\theta = \{X \mapsto f(\overline{X_n})\}$

[v] $G, X \simeq t, G' \overset{\text{LNC}}{\Longrightarrow}_{[iv],\theta} G\theta, G'\theta$
where $\theta = \{X \mapsto t\}$ if $X \notin \mathit{Vars}(s)$

[t] $G, X \approx X, G' \overset{\text{LNC}}{\Longrightarrow}_{[t],\varepsilon} G, G'$

[o] $G, f(\overline{s_n}) \simeq t, G' \overset{\text{LNC}}{\Longrightarrow}_{[o],\varepsilon} G, \overline{s_n \approx l_n}, r \simeq t, G'$
if $f(\overline{l_n}) \to r$ is a fresh variant of a rewrite rule in $\mathcal{R}$

Figure 1: LNC: inference rules

Remark 1 *The properties of LNC are preserved if we drop the confluence condition for $\mathcal{R}$ and weaken the definition of solution in the following way: $\gamma \in \mathcal{U}_{\mathcal{R}}(s \approx t)$ if $s\gamma \downarrow_{\mathcal{R}} t\gamma$* □

2.2 The Calculus LN_1

One way to extend lazy narrowing to the higher-order case is via the following generalizations:

$$\begin{array}{rcl} \text{term algebra} & \longrightarrow & \text{simply-typed term algebra,} \\ \text{rewriting with TRS} & \longrightarrow & \text{rewriting with PRS [10, 11].} \end{array}$$

This approach has been pursued by Prehofer in the design of his calculus LN [12]. His framework adopts the following restrictions:

1. terms are identified with their long $\beta\eta$-normal forms,
2. $\mathcal{R}$ consists of relations of the form $f(\overline{l_n}) \to r$ where $f(\overline{l_n}), r$ are terms of the same base type, $\mathit{Vars}(f(\overline{l_n})) \supseteq \mathit{Vars}(r)$ and $f(\overline{l_n})$ is a higher-order pattern [9]. Here, $\mathit{Vars}(t)$ denotes the set of free variables in the term t. Such an $\mathcal{R}$ is called *pattern rewrite system* (PRS),
3. the equational goals consist of oriented equations. To make this distinction clear, we write an oriented equation in the form $s \rhd t$, and define $\mathcal{R} \vdash s \rhd t$ if $s \to^*_{\mathcal{R}} t$.

Also, because solving flex/flex equations is in general intractable, the calculus LN does not attempt to solve them.

The inference rules of LN are shown in Fig. 2. We have adopted the terminology and notational conventions from [12].

In this case, a refutation is a finite LN-derivation of maximum length, i.e. a sequence of LN-steps $G \overset{\text{LN}}{\Longrightarrow}{}^*_\theta F$ where F is a flex/flex goal with no trivial equations. In this case, the set of answers generated by LN is

$$Ans^{\text{LN}}_{\mathcal{R}}(G) = \{\langle\theta, F\rangle \mid \exists \text{ LN-refutation } G \overset{\text{LN}}{\Longrightarrow}{}^*_\theta F\}$$

Computing with LN-refutations has the following properties:

Soundness: $\forall\langle\theta, F\rangle \in Ans^{\text{LN}}_{\mathcal{R}}(G).\ \forall\gamma \in \mathcal{U}_{\mathcal{R}}(F) : \theta\gamma \in \mathcal{U}_{\mathcal{R}}(G)$,

Completeness: $\forall\gamma \in \mathcal{U}_{\mathcal{R}}(G).\ \exists\langle\theta, F\rangle \in Ans^{\text{LN}}_{\mathcal{R}}(G).\ \exists\gamma' \in \mathcal{U}_{\mathcal{R}}(F)$: $\gamma = \theta\gamma'\ [\mathit{Vars}(G)]$.

2.3 LN versus LNC

The table below shows the similarities and differences between the lazy narrowing calculi LNC and LN.

Calculus	LNC	LN
Sound	yes	yes
Strongly complete	no	yes
Complete	yes (strategy $\mathcal{S}_{\text{left}}$)	yes
Computing power	solves all equs.	does not solve flex/flex equs.
	solves unoriented equs.	solves oriented equs.
	complete w.r.t. $\mathcal{U}^{n}_{\mathcal{R}}(G)$	complete w.r.t. $\mathcal{U}_{\mathcal{R}}(G)$
Optimizations	[7]	[12]

3 THE CALCULUS LN$_1$

LN$_1$ is our first proposal of a higher-order lazy narrowing calculus. LN$_1$ is a generalization of the calculi LNC and LN that allows to extend some of the important properties of LNC to the higher-order case. LN$_1$ generalizes the calculi LNC and LN in the following way:

1. LN$_1$ solves goals consisting of both oriented and unoriented equations,

2. LN_1 has additional inference rules to solve certain flex/flex equations. With this extension, LNC and the first-order version of LN_1 have the same computing power, if we adopt strategy $\mathcal{S}_{\mathrm{left}}$ (See Subsection 3.1)

3. LN_1 is designed to compute solutions which are $\mathcal{R}$-normalized with respect to a given set of variables.

Formally, LN_1 is designed to solve the following problem

Given a confluent PRS $\mathcal{R}$ and a pair $\langle W \mid G\rangle$ consisting of a set of variables W and a goal G

Compute a complete subset of

$$\mathcal{U}_{\mathcal{R}}(\langle W \mid G\rangle) = \{\gamma \in \mathcal{U}_{\mathcal{R}}(G) \mid \gamma\restriction_W \text{ is } \mathcal{R}\text{-normalized}\}$$

i.e., a set $Ans_{\mathcal{R}}(G)$ of pairs $\langle \theta, F\rangle$ such that

soundness: $\theta\gamma \in \mathcal{U}_{\mathcal{R}}(G)$ for all $\gamma \in \mathcal{U}_{\mathcal{R}}(F)$, and

completeness: $\forall\gamma \in \mathcal{U}_{\mathcal{R}}(\langle W \mid G\rangle)$. $\exists\ \langle\theta, F\rangle \in Ans_{\mathcal{R}}(\langle W \mid G\rangle)$. $\exists\gamma' \in \mathcal{U}_{\mathcal{R}}(F)$ such that $\gamma = \theta\gamma'$ $[\mathit{Vars}(G)]$.

3.0.1 Inference rules.

The inference rules of LN_1 are relations of the form

$$\langle W \mid G\rangle \overset{\mathrm{LN}_1}{\Longrightarrow}_{\theta} \langle W' \mid G'\rangle$$

where θ is a pattern substitution and $W' = \mathit{Vars}(W\theta)$. LN_1 consists of the rules [t], [d], [i], [p], [ffs], [ffd], [on], [ov] shown in Fig. 3. Essentially, the inference rules [t], [d], [i], [p], [ffs], [ffd] are those required to perform full pattern unification [13], whereas [on] and [ov] are specific to lazy narrowing.

The following comments are intended to clarify some of our notational conventions. We assume that $\cong$ is a placeholder for

- $\approx, \approx^{-1}, \rhd$ or $\rhd^{-1}$ in inference rules [i], [p],
- $\approx, \approx^{-1}$ or $\rhd$ in inference rules [on], [ov],
- $\approx$ or $\rhd$ in inference rules [t], [d], [ffs], [ffd].

We also adopt the idea to use *marked equations* of the form $s \blacktriangleright t$ or $s \blacktriangleright_{\mathrm{d}} t$. $s \blacktriangleright t$ denotes an oriented equation $s \rhd t$ which is (descendant of a) *parameter-passing* equation. $s \blacktriangleright_{\mathrm{d}} t$ denotes an oriented equation $s \rhd t$ which is (descendant of a) parameter-passing equation to which we do not apply [on] or [ov]. Note that only [i] and [d] can remove the marker d of an equation. The construct $s \cong_{(\mathrm{d})} t$ denotes either the marked equation $s \cong_{\mathrm{d}} t$ or the unmarked equation $s \cong t$.

When used both in the left and the right side of an inference rule, the placeholder $\cong$ is assumed to denote the same equational symbol with the same orientation. We also assume that the marker (d) is preserved.

These markers serve for particular optimizations, and are ignored when irrelevant.

Theorem 2 LN_1 *is sound and strongly complete.*

3.1 LN_1 versus LN and LNC

LN_1 is an extension of the calculus LN. It is more powerful than LN because it can solve both oriented and unoriented equations, and also certain flex/flex equations that LN does not solve. Also, LN_1 is more deterministic than LN because rule [ov] is applicable in fewer situations.

The calculus LN_1 is also a generalization of the calculus LNC. To see why this is so, let LNC_v be the calculus obtained from LNC by replacing rule [v] with the rule first-order version of rule [ffd]. The calculi LNC and LNC_v have the same computing power because any [v]-step of LNC can be simulated by a sequence of [i]- and [ffd]-steps of LNC_v. We claim that for any LNC_v-refutation which respects strategy $\mathcal{S}_{\mathrm{left}}$

$$\Pi : G_0 \overset{\mathrm{LNC}}{\Longrightarrow}_{\alpha_1,\theta_1} G_1 \overset{\mathrm{LNC}}{\Longrightarrow}_{\alpha_2,\theta_2} \cdots \overset{\mathrm{LNC}}{\Longrightarrow}_{\alpha_N,\theta_N} G_N = \Box$$

we can construct a corresponding LN_1-refutation

$$\Pi' : \langle \mathit{Vars}(G), G\rangle \overset{\mathrm{LN}_1}{\Longrightarrow}_{\alpha_1,\theta_1} \langle W_1 \mid G_1\rangle \overset{\mathrm{LN}_1}{\Longrightarrow}_{\alpha_2,\theta_2} \cdots \overset{\mathrm{LN}_1}{\Longrightarrow}_{\alpha_N,\theta_N} \langle W_N \mid \Box\rangle$$

which respects strategy $\mathcal{S}_{\mathrm{left}}$. Our claim is a consequence of the observation that all leftmost equations e_k in G_k satisfy the condition $\mathcal{V}_c(e_k) \subseteq W_k$, and therefore [ffd] is always applicable. Thus our lifting of the LNC_v-refutation Π to the LN_1-refutation Π' is valid, and we can conclude that LN_1 with strategy $\mathcal{S}_{\mathrm{left}}$ has the same computing power as LNC_v with strategy $\mathcal{S}_{\mathrm{left}}$.

4 OPTIMIZATIONS

In the sequel we address the problem of reducing the non-determinism of LN_1 which is due to the many choices to select the inference rule to be applied to a selected equation. All our optimizations are defined for rewrite systems which are left-linear and confluent. The restriction of programs to left-linear (term or pattern) rewrite systems is widely accepted in functional logic. The following lemma gives a first glimpse of the structure of LN_1-refutations when $\mathcal{R}$ is a left-linear PRS. It is similar to Lemma 3.1 [7] for LNC with left-linear CS.

Lemma 3 *Let $\mathcal{R}$ be a left-linear PRS and*

$$\Pi : \langle W \mid G\rangle \overset{\mathrm{LN}_1}{\Longrightarrow}{}^{*}_{\theta}\langle W' \mid G'_1, \lambda\overline{x}.s \blacktriangleright \lambda\overline{x}.t, G'_2\rangle$$

an LN_1*-derivation. Then*

(i) $(Vars(G'_1, \lambda\overline{x}.s) \cup Vars(G\theta)) \cap Vars(\lambda\overline{x}.t) = \emptyset$,

(ii) $\lambda\overline{x}.t$ *is a linear pattern.*

4.1 Lazy Narrowing with Elimination of Redundant Equations

Upon computations of normalized solutions with the calculus LN_1 with strategy $\mathcal{S}_c$, it may happen to generate new equations which do not contribute to the computation of the final solution. We call such equations *redundant*, and propose the following criterion to detect them.

Definition 4 (redundant equation) *An equation e is* redundant *in an* LN_1*-goal* $\langle W \mid G_1, e, G_2\rangle$ *if e is of the form* $\lambda\overline{x}.s \blacktriangleright \lambda\overline{x}.t$ *with* $Vars(\lambda\overline{x}.t) \cap Vars(G_2) = \emptyset$.

4.1.1 The calculus LN_2.

LN_2 is our first optimization of the calculus LN_1. It is defined for left-linear PRS which are *fully-extended* (EPRS for short), i.e. consist of rewrite rules $f(l_1, \ldots, l_n) \to r$ with $f(l_1, \ldots, l_n)$ a fully-extended pattern.

Definition 5 (fully-extended pattern) *A simply-typed λ-term t is called* fully-extended pattern *if whenever $X(\overline{s_n})$ is a subterm of t at position p then $\overline{s_n}$ is the sequence of all distinct bound variables in the scope of $t|_p$.*

For example, $\lambda x_1, x_2.f(X(x_1, x_2), \lambda x_3.g(Y(x_1, x_3, x_2)))$ is a fully extended pattern, whereas $\lambda x_1, x_2.f(X(x_1, x_2), \lambda x_3.g(Y(x_1, x_2)))$ is not.

We define $\mathrm{LN}_2 = \mathrm{LN}_1 \cup \{\mathrm{rm}]\}$ where the inference rule [rm] defined below has highest priority:

[rm] $\langle W \mid G_1, e, G_2\rangle \overset{\mathrm{LN}_2}{\Longrightarrow}_{[\mathrm{rm}],\varepsilon} \langle W \mid G\rangle$ if e is redundant in $\langle W \mid G_1, e, G_2\rangle$.

Theorem 6 *If $\mathcal{R}$ is an EPRS then* LN_2 *is sound and strongly complete.*

4.2 Restricting Lazy Narrowing at Nonvariable Position

This optimization is a generalization of the solution to the eager variable elimination problem proposed in [7] for TRS which satisfy the standardization theorem (e.g., left-linear confluent). We prove a similar result for the calculus LN_2: if $\mathcal{R}$ is a left-linear confluent EPRS then we can drop the application of [on] to equations of the form $\lambda\overline{x}.f(\overline{s}) \blacktriangleright \lambda\overline{x}.X(\overline{y})$ without losing completeness.

Let LN_2^{ev} be the calculus obtained from LN_2 by dropping the application of [on] to (descendants of) parameter-passing equations of the form $\lambda\overline{x}.f(\overline{s}) \blacktriangleright \lambda\overline{x}.X(\overline{y})$ with $f \in \mathcal{F}_d$.

Theorem 7 *If $\mathcal{R}$ is a left-linear confluent EPRS then* LN_2^{ev} *is sound and strongly complete.*

4.3 Lazy Narrowing for Left-Linear Constructor EPRS

Our optimization for constructor EPRS (PRS for short) has been inspired by a similar optimization of the calculus LNC for left-linear CS [7]. This optimization addresses the possibility to avoid the generation of (descendants of) parameter-passing equations of the form $\lambda\overline{x}.s \blacktriangleright \lambda\overline{x}.t$ with $\lambda\overline{x}.t \notin \mathcal{T}(\mathcal{F}_c, \mathcal{V})$, when $\mathcal{R}$ is a left-linear constructor EPRS. An obvious advantage of this optimization is that the nondeterminism between rules [on] and [d] disappears for selected equations $\lambda\overline{x}.s \blacktriangleright \lambda\overline{x}.t$.

Note that in the first-order case it is sufficient to use the selection strategy $\mathcal{S}_{\mathrm{left}}$ to avoid generating equations $\lambda\overline{x}.s \blacktriangleright \lambda\overline{x}.t$ with $\lambda\overline{x}.t \notin \mathcal{T}(\mathcal{F}_c, \mathcal{V})$. Unfortunately, this method can not be generalized to the higher-order case, because we don't solve all flex/flex equations. The following example illustrates this fact.

Example 1 Consider the left-linear constructor PRS $\mathcal{R} = \{f(X) \to X\}$, the goal $G = f(Y(X)) \rhd a$, and the LN_2-derivation

$$\langle\{X, Y\} \mid \underline{f(Y(X)) \rhd a}\rangle \overset{\mathrm{LN}_2}{\Longrightarrow}_{[\mathrm{on}]} \overset{\mathrm{LN}_2}{\Longrightarrow}_{[\mathrm{d}]} \langle\{X, Y\} \mid Y(X) \blacktriangleright_{\mathrm{d}} X_1, \underline{X_1 \rhd a}\rangle$$
$$\overset{\mathrm{LN}_2}{\Longrightarrow}_{[\mathrm{ov}]} \overset{\mathrm{LN}_2}{\Longrightarrow}_{[\mathrm{i}],\sigma} \langle\{X, Y\} \mid G' = Y(X) \blacktriangleright f(H), H(X) \blacktriangleright X_2, X_2 \rhd a\rangle$$

where $\sigma = \{X_1 \mapsto f(H)\}$. The application of [ov] in the second inference step introduces the defined symbol f in the right-hand side of the leftmost parameter-passing equation of G'. □

To avoid this behavior, we define the calculus $\mathrm{LN}_3 = \mathrm{LN}_2 \cup \{[\mathrm{c}]\}$ where

$$[\mathrm{c}] \quad \langle W \mid G, \lambda\overline{z}.s \blacktriangleright \lambda\overline{z}.X(\overline{z}), G', \lambda\overline{x}.X(\overline{t}) \cong \lambda\overline{x}.u, G''\rangle \xrightarrow{\mathrm{LN}_3}_{[\mathrm{c}],\epsilon}$$
$$\langle W \mid G, \lambda\overline{z}.s \blacktriangleright \lambda\overline{z}.X(\overline{z}), G', \lambda\overline{x}.s(\overline{t}) \cong \lambda\overline{x}.u, G''\rangle$$

and give to [c] the highest priority. We also define a suitable equation selection strategy.

Definition 8 (strategy S_c) *An equation $e \in G$ to which [ov] is applicable is selected in a goal $\langle W \mid G\rangle$ only if all the parameter-passing equations which precede it are of the form $\lambda\overline{x}.s \blacktriangleright \lambda\overline{x}.X(\overline{x})$.*

Theorem 9 *Let $\mathcal{R}$ be a left-linear confluent constructor EPRS. Then*

(i) LN_3 *is sound and strongly complete.*

(ii) If Π *is an* LN_3 *derivation which respects strategy S_c then all equations $\lambda\overline{x}.s \blacktriangleright \lambda\overline{x}.t$ in* Π *satisfy the condition $\lambda\overline{x}.t \in \mathcal{T}(\mathcal{F}_c, \mathcal{V})$.*

5 CONCLUSIONS AND FUTURE WORK

Various optimizations of a higher-order lazy narrowing calculus have been proposed, in an attempt to reduce its high nondeterminism and to make it a suitable operational semantics for higher-order FLP. All calculi proposed by us are sound and complete.

A powerful mechanism for FLP is lazy narrowing with conditional rewrite systems. The extension of lazy narrowing to the conditional term rewriting case has been successful in the first-order case [2], and proposals for higher-order versions of conditional lazy narrowing are already available. (See, e.g. [12].) As future work, we intend to generalize our framework of higher-order lazy narrowing with PRS to the conditional case, and to determine optimizations for classes of conditional PRS which are useful for programming purposes.

References

[1] W. Gould. *A matching procedure for ω-order logic.* Scientific Report 4. Air Force Cambridge Research Laboratories, 1966.

[2] M. Hamada, A. Middeldorp, and T. Suzuki. Completeness Results for a Lazy Conditional Narrowing Calculus. In *DMTCS/CATS'99*, pages 217–231, Auckland, 1999. Springer-Verlag Singapore.

[3] G. Huèt. A Unification Algorithm for Typed λ-Calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[4] J.-M. Hullot. Canonical Forms and Unification. In *Proceedings of the 5th Conference on Automated Deduction*, volume 87 of *LNCS*, pages 318–334. Springer, 1980.

[5] M. Marin, T. Ida, and T. Suzuki. On Reducing the Search Space of Higher-Order Lazy Narrowing. In A. Middeldorp and T. Sato, editors, *FLOPS'99*, volume 1722 of *LNCS*, pages 225–240. Springer-Verlag, 1999.

[6] M. Marin, A. Middeldorp, T. Ida, and T. Yanagi. LNCA: A Lazy Narrowing Calculus for Applicative Term Rewriting Systems. Technical Report ISE-TR-99-158, Institute of Information Sciences and Electronics, University of Tsukuba, Tsukuba, Japan, 1999.

[7] A. Middeldorp and S. Okui. A Deterministic Lazy Narrowing Calculus. *Journal of Symbolic Computation*, 25(6):733–757, 1998.

[8] A. Middeldorp, S. Okui, and T. Ida. Lazy Narrowing: Strong Completeness and Eager Variable Elimination. *Theoretical Computer Science*, 167(1,2):95–130, 1996.

[9] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1:497–536, 1991.

[10] T. Nipkow. Functional unification of higher-order patterns. In *Proceedings of 8th IEEE Symposium on Logic in Computer Science*, pages 64–74, 1993.

[11] T. Nipkow and C. Prehofer. Higher-order rewriting and equational reasoning. In P. S. W. Bibel, editor, *Automated Deduction - A Basis for Applications*, volume 1, chapter Formal Models and Semantics, pages 399–430. Kluwer, 1998.

[12] C. Prehofer. *Solving Higher-Order Equations. From Logic to Programming.* Foundations of Computing. Birkäuser Boston, 1998.

[13] W. Snyder and J. Gallier. Higher-order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8:101–140, 1989.

[14] T. Suzuki, K. Nakagawa, and T. Ida. Higher-Order Lazy Narrowing Calculus: A Computation Model for a Higher-order Functional Logic Language. In *Proceedings of Sixth International Joint Conference, ALP '97 - HOA '97*, volume 1298 of *LNCS*, pages 99–113, Southampton, 1997.

Received: ....
Accepted: ....

Address
....
....
....

$\bowtie \in \{\rhd, \rhd^{-1}\}$, $\bowtie_{(\mathrm{d})} \in \{\bowtie, \bowtie_{\mathrm{d}}\}$

[t] *removal of trivial equations*

$$G, \lambda\overline{x}.t \rhd_{\mathrm{d}} \lambda\overline{x}.t, G' \overset{\mathrm{LN}}{\Longrightarrow}_{\varepsilon} G, G'$$

[d] *decomposition*

$$G, \lambda\overline{x}.v(\overline{s_n}) \rhd_{\mathrm{d}} \lambda\overline{x}.v(\overline{t_n}), G' \overset{\mathrm{LN}}{\Longrightarrow}_{\varepsilon} G, \overline{\lambda\overline{x}.s_n \rhd \lambda\overline{x}.t_n}, G' \quad \text{if } v \in \mathcal{F} \cup \{\overline{x}\}$$

[i] *imitation*

$$G, \lambda\overline{x}.X(\overline{s_n}) \bowtie \lambda\overline{x}.g(\overline{t_m}), G' \overset{\mathrm{LN}}{\Longrightarrow}_{\theta} (G, \overline{\lambda\overline{x}.H_m(\overline{s_n}) \bowtie_{(\mathrm{d})} \lambda\overline{x}.t_m}, G')\theta$$

where $\theta = \{X \mapsto \lambda\overline{x_n}.g(\overline{H_m(\overline{x_n})})\}$ and $\overline{H_m}$ are fresh variables.

[p] *projection*

$$G, \lambda\overline{x}.X(\overline{s_n}) \bowtie_{(\mathrm{d})} \lambda\overline{x}.t, G' \overset{\mathrm{LN}}{\Longrightarrow}_{\theta} (G, \lambda\overline{x}.s_i(\overline{H_p(\overline{s_n})}) \bowtie_{(\mathrm{d})} \lambda\overline{x}.t, G')\theta$$

where $1 \le i \le n$, $\lambda\overline{x}.t$ is rigid, $\theta = \{X \mapsto \lambda\overline{y_n}.y_i(\overline{H_p(\overline{y_n})})\}$, $y_i : \overline{\tau_p} \to \tau$, and $\overline{H_p : \tau_p}$ are fresh variables.

[on] *lazy narrowing at nonvariable*

$$G, \lambda\overline{x}.f(\overline{s_n}) \rhd \lambda\overline{x}.t, G' \overset{\mathrm{LN}}{\Longrightarrow}_{\varepsilon} G, \lambda\overline{x}.f(\overline{s_n} \rhd_{\mathrm{d}} \lambda\overline{x}.f(\overline{l_n}), \lambda\overline{x}.r \rhd \lambda\overline{x}.t, G'$$

if $f(\overline{l_n}) \to r$ is an $\overline{x}$-lifted rule.

[ov] *outermost narrowing at variable*

$$G, \lambda\overline{x}.X(\overline{s}) \rhd \lambda\overline{x}.t, G' \overset{\mathrm{LN}}{\Longrightarrow}_{\varepsilon}, \lambda\overline{x}.X(\overline{s}) \rhd_{\mathrm{d}} \lambda\overline{x}.f(\overline{l_n}), \lambda\overline{x}.r \rhd \lambda\overline{x}.t, G'$$

if $\lambda\overline{x}.t$ is rigid, $f(\overline{l_n}) \to r$ is an $\overline{x}$-lifted rule and $\theta = \{X \mapsto \lambda\overline{y_m}.f(\overline{H_n(\overline{y_m})})\}$ with $\overline{H_n}$ fresh variables of appropriate types.

Figure 2: The lazy narrowing calculus LN: inference rules

[t] *removal of trivial equations*

$$\langle W \mid G, \lambda\overline{x}.t \cong_{(d)} \lambda\overline{x}.t, G'\rangle \overset{\mathrm{LN_l}}{\Longrightarrow}_{[t],\varepsilon} \langle W \mid G, G'\rangle$$

[d] *decomposition*

$$\langle W \mid G, \lambda\overline{x}.v(\overline{s_n}) \cong_{(d)} \lambda\overline{x}.v(\overline{t_n}), G'\rangle \overset{\mathrm{LN_l}}{\Longrightarrow}_{[d],\varepsilon} \langle W \mid G, \overline{\lambda\overline{x}.s_n \cong \lambda\overline{x}.t_n}, G'\rangle$$

if $v \in \mathcal{F} \cup \{\overline{x}\}$.

[i] *imitation*

$$\langle W \mid G, \lambda\overline{x}.X(\overline{s_n}) \cong_{(d)} \lambda\overline{x}.g(\overline{t_m}), G'\rangle \overset{\mathrm{LN_l}}{\Longrightarrow}_{[i],\theta} \langle W' \mid (G, \overline{\lambda\overline{x}.H_m(\overline{s_n}) \cong \lambda\overline{x}.t_m}, G')\theta\rangle$$

where $\theta = \{X \mapsto \lambda\overline{x_n}.g(\overline{H_m(\overline{x_n})})\}$ and $\overline{H_m}$ are fresh variables.

[p] *projection*

$$\langle W \mid G, \lambda\overline{x}.X(\overline{s_n}) \cong_{(d)} \lambda\overline{x}.t, G'\rangle \overset{\mathrm{LN_l}}{\Longrightarrow}_{[p],\theta} \langle W' \mid (G, \lambda\overline{x}.s_i(\overline{H_p(\overline{s_n})}) \cong_{(d)} \lambda\overline{x}.t, G')\theta\rangle$$

where $1 \le i \le n$, $\lambda\overline{x}.t$ is rigid, $\theta = \{X \mapsto \lambda\overline{y_n}.y_i(\overline{H_p(\overline{y_n})})\}$, $y_i : \overline{\tau_p} \to \tau$, and $\overline{H_p : \tau_p}$ are fresh variables.

[ffs] *flex/flex same*

$$\langle W \mid G, \lambda\overline{x}.X(\overline{y_n}) \cong_{(d)} \lambda\overline{x}.X(\overline{y'_n}), G'\rangle \overset{\mathrm{LN_l}}{\Longrightarrow}_{[ffs],\theta} \langle W' \mid G\theta, G'\theta\rangle$$

where $X \in W$, $\theta = \{X \mapsto \lambda\overline{y_n}.H(\overline{z})\}, \{\overline{z}\} = \{y_i \mid y_i = y'_i, 1 \le i \le n\}$.

[ffd] *flex/flex different*

$$\langle W \mid G, \lambda\overline{x}.X(\overline{y}) \cong_{(d)} \lambda\overline{x}.Y(\overline{y'}), G'\rangle \overset{\mathrm{LN_l}}{\Longrightarrow}_{[ffd],\theta} \langle W' \mid G\theta, G'\theta\rangle$$

where $\theta = \{X \mapsto \lambda\overline{y}.H(\overline{z}), Y \mapsto \lambda\overline{y'}.H(\overline{z})\}$, $\{\overline{z}\} = \{\overline{y}\} \cap \{\overline{y'}\}$,$X \in W$, and $Y \in W$ if $\cong$ is $\approx$.

[on] *outermost narrowing at nonvariable position*

$$\begin{array}{l}\langle W \mid G, \lambda\overline{x}.f(\overline{s_n}) \cong \lambda\overline{x}.t, G'\rangle \overset{\mathrm{LN_l}}{\Longrightarrow}_{[on],\varepsilon} \\ \langle W \mid G, \lambda\overline{x}.f(\overline{s_n}) \blacktriangleright_{\mathrm{d}} \lambda\overline{x}.f(\overline{l_n}), \lambda\overline{x}.r \cong \lambda\overline{x}.t, G'\rangle\end{array}$$

if $f(\overline{l_n}) \to r$ is an $\overline{x}$-lifted rule.

[ov] *outermost narrowing at variable position*

$$\begin{array}{l}\langle W \mid G, \lambda\overline{x}.X(\overline{s_m}) \cong \lambda\overline{x}.t, G'\rangle \overset{\mathrm{LN_l}}{\Longrightarrow}_{[ov],\varepsilon} \\ \langle W \mid G, \lambda\overline{x}.X(\overline{s_m}) \blacktriangleright_{\mathrm{d}} \lambda\overline{x}.f(\overline{l_n}), \lambda\overline{x}.r \cong \lambda\overline{x}.t, G'\rangle\end{array}$$

where $\lambda\overline{x}.t$ is rigid, $f(\overline{l_n}) \to r$ is an $\overline{x}$-lifted rule, and $\lambda\overline{x}.X(\overline{s_m})$ is not a flex pattern with $X \in W$.

Figure 3: The calculus $\mathrm{LN_l}$: inference rules

A New Criterion for Safe Program Transformations

Yasuhiko Minamide

Institute of Information Sciences and Electronics
University of Tsukuba
and
PRESTO
Japan Science & Technology Corporation

Email: minamide@score.is.tsukuba.ac.jp

Abstract

Previous studies on safety of program transformations with respect to performance considered two criteria: preserving performance within a constant factor and preserving complexity. However, as the requirement of program transformations used in compilers the former seems too restrictive and the latter seems too loose. We propose a new safety criterion: a program transformation preserves performance within a factor proportional to the size of a source program. This criterion seems natural since several compilation methods have effects on performance proportional to the size of a program. Based on this criterion we have shown that two semantics formalizing the size of stack space are equivalent. We also discuss the connection between this criterion and the properties of local program transformations rewriting parts of a program.

1 Introduction

Recent compilers utilize advanced program transformations to obtain high-performance executable code. For these advanced program transformations, it is not so straightforward to guarantee that they are safe with respect to performance. In fact, some program transformations have been shown to improve the performance of most programs, while degrading the performance of some programs severely [9,12].

To remedy this situation, several papers have discussed the safety of program transformations based on semantics formalizing the performance of programs [7,6,11,2,8]. In those studies, two safety criteria for whole-program transformations were discussed. However, these criteria do not seem appropriate to impose on program transformations used in compilers, for reasons we discuss below. In this paper we focus on the space requirement of programs in call-by-value functional languages, but the general framework can be adopted for other languages and performance metrics.

The first of the above two criteria ensures that a program transformation preserves space requirement within a constant factor. If a program transformation satisfies this property, we say that the program transformation is space efficient. Many program transformations seem to satisfy this criterion, but there are some useful transformations that do not satisfy this criterion. Furthermore, to show this criterion is satisfied we must formalize the details of semantics formalizing space requirements. In the study of the CPS transformation, it was necessary to revise the space profiling semantics of a call-by-value functional language by Blelloch and Greiner [3], to show that this transformation satisfies this criterion [8].

The second criterion is space safety: a program transformation is space safe if it does not raise the complexity of programs. Clearly, program transformations used in a compiler must satisfy this criterion. However, we think that this criterion is too loose. This criterion does not impose any restriction for programs without inputs. However, showing space safety is simpler than showing space efficiency, in the sense that it is possible to adopt a simpler profiling semantics that ignores details such as sizes of closures and stack frames.

In this paper we propose a new criterion that falls between the above two criteria. The new criterion is that a program transformation preserves the space requirement within a factor proportional to the size of a source program. Most useful transformations used in a compiler seem to satisfy this criterion. This criterion seems natural because several compilation methods have effects on performance proportional to the size of a program. Furthermore, we can show this criterion based on semantics ignoring some details as we show space safety.

Based on the new criterion we have shown that two semantics of a simple call-by-value functional language profiling stack space are equivalent. One models evaluation by an interpreter and the other models execution based on compilation. They are not equivalent in the sense of space efficiency, but they are equivalent in the sense of our new criterion. We also show that A-normalization preserves stack space modeled by the second semantics. This backs the claim that the second semantics models stack space required for execution based on compilation.

The criterion we propose is a property of a whole-program transformation. On the other hand, some transformations used in compilers are based on local program transformations. We therefore also study the connection between the properties of local transformations and the properties of global transformations. We will show that some restricted class of local transformations induces whole-program transformations satisfying our new criterion.

This paper is organized as follows. We begin by reviewing the two safety criteria discussed in previous studies and discussing why they are not suitable as the criterion we impose on the transformations used in a compiler. In Section 3 we introduce our new safety criterion and the equivalence of semantics on a programming language induced by the criterion. In Section 4, based on this new safety criterion, we show that two operational semantics of a call-by-value functional language are equivalent. In Section 5 we discuss the connection between the properties of local transformations and our new criterion. Finally, we give our conclusions and directions for future work.

2 Safety criteria of program transformations

We review the safety criteria of program transformations with respect to performance discussed in previous studies. To formalize safety criteria we must first develop semantics to formalize performance of programs. We call such semantics profiling semantics. For a simple programming language, profiling semantics can be given as a partial function $eval(M)$:

$$eval(M) = (v, n)$$

This function gives the computation result v and a non-negative integer n, which represents such performance values of programs as execution time or space required for execution.

In this paper, to simplify our discussion, we focus on the space requirement of a program and ignore the computation result. For this purpose we define $space(M)$ as below:

$$space(M) = n \textit{ iff } eval(M) = (v, n) \textit{ for some } v$$

This function is only defined if the evaluation of M terminates. We call this semantics for specifying space required for execution of a program *space semantics.*

Let us now review two space safety criteria of program transformations discussed in previous work [1,3,8]. In this paper, we consider a program transformation as a binary relation between programs in a source language and a target language. Let us consider a program transformation $\leadsto$ between a source language and a target language that have space semantics $space(M)$ and $space'(M')$ respectively: $M \leadsto M'$ means that M is translated to M' by the program transformation.

The first criterion ensures that a program transformation preserves the space required for execution of a program within a constant factor.

Definition 2.1 We say that a program transformation $\leadsto$ is space efficient if constants k_1 and k_2 exist such that:

$$space(M) = n \implies space'(M') \leq k_1 n + k_2$$

for any programs M and M' with $M \rightsquigarrow M'$.

We admit constants k_1 and k_2 because they seem dependent on the details of definition of space semantics and not essential. Moreover, there are several transformations in which $k_1 > 1$ is required to show this property. This property was first discussed by Blelloch and Greiner for an implementation of NESL [3]. Minamide also showed that the CPS transformation is space efficient [8].

The second criterion is called space safety: a transformation is *space safe* if it does not increase the space complexity of programs [1]. To formalize this idea we must consider programs with an input; thus we can consider a programming language with input commands and the semantics $space(M, I)$, which formalizes the space required for execution of the program M for the input I.

Definition 2.2 We say that a program transformation $\rightsquigarrow$ is space safe if for any programs M and M' such that $M \rightsquigarrow M'$, constants k_1 and k_2 exist such that for any input I the following holds:

$$space(M, I) = n \implies space'(M', I) \leq k_1 n + k_2$$

The key difference from space efficiency is that the constants k_1 and k_2 are program-dependent. Space efficiency usually implies space safety because space efficiency provides the constants k_1 and k_2 to show space safety without depending on programs.

Although many program transformations used in compilers seem space efficient, some useful transformations are not space efficient, but only space safe. Furthermore, to show that a program transformation is space efficient we must consider too many details of the operational semantics of the source language. In the study of the CPS transformation it was necessary to revise the semantics proposed by Blelloch and Greiner [3] to show that the CPS transformation was space efficient [8].

Code motion is a typical example of a transformation that is space safe, but not space efficient. Consider the following expression where `M` is a pure expression that contains the variable n but does not contain a function application:

```
fun loop 0 = ()
  | loop n = let val x = M  in  loop (n - 1) end
```

The value of `M` is loop-invariant, thus we want to hoist the binding of `x` as follows:

```
val x = M
fun loop 0 = ()
  | loop n = loop (n - 1)
```

This usually improves the performance of the program. However, if the function `loop` is used only as `loop 0` or is not actually used, this transformation results in extra computation of `M`. The extra time and space to evaluate `M` is not uniformly bounded by a constant, but depends on `M`.

On the other hand, space safety seems too weak as a requirement of transformations used in compilers. Space safety is trivial for programs without inputs. Even for programs with inputs, it is impossible to estimate the performance of a program since the constants k_1 and k_2 are program-dependent. Thus in this paper we propose a new criterion that falls between space efficiency and space safety.

3 A new criterion

In this section, we propose a new safety criterion for program transformations. First, to simplify our discussion, we compare two space semantics of a programming language.

Let us consider two space semantics $\mathit{space}_1(M)$ and $\mathit{space}_2(M)$ of a programming language. As a natural extension of space efficiency, we can consider the following property: there exists a polynomial f such that

$$\mathit{space}_1(M) = n \implies \mathit{space}_2(M) \leq f(n)$$

However, this property is not suitable as a criterion that $\mathit{space}_2(M)$ is safe with respect to $\mathit{space}_1(M)$. By extending the language and the semantics with inputs, we have the following property:

$$\mathit{space}_1(M, I) = n \implies \mathit{space}_2(M, I) \leq f(n)$$

Even if this holds, it might happen that for an input of size n space_1 requires n space, but space_2 requires n^2 space, because $f(x) = x^2$. Thus, this extended property does not imply safety and is therefore not suitable as a criterion that space_2 is safe with respect to space_1.

When we consider various semantics of a language, the difference in space usage often depends on the size of a program. Thus, it is natural to consider the following relation of semantics.

Definition 3.1 We say that the semantics space_1 is weakly simulated by space_2 if

$$\mathit{space}_1(M) = n \implies \mathit{space}_2(M) \leq f_1(|M|)n + f_2(|M|)$$

where $f_1(x)$ and $f_2(x)$ are polynomials with positive coefficients and $|M|$ is the size of M.

Hereafter in this paper we simply say "a polynomial" for "a polynomial with positive coefficients."

This relation induces equivalence of semantics as follows.

Definition 3.2 We say that semantics $space_1$ is weakly space equivalent to $space_2$ if $space_1$ is weakly simulated by $space_2$ and $space_2$ is weakly simulated by $space_1$.

Most reasonably defined semantics of a language seem weakly space equivalent. Moreover, it is possible to define simpler semantics weakly space equivalent to the semantics considered in previous studies.

Example 3.3 Let $space_1$ be a semantics of a functional language that accounts for the sizes of closures: the size of a closure with n free variables is $n+1$. Let $space_2$ be a semantics where the sizes of closures are ignored: the sizes of closures are always 1. Then $space_1$ and $space_2$ are weakly equivalent, since the sizes of the closures constructed during evaluation of a program are bounded by the size of the program.

Example 3.4 Let $space_1$ and $space_2$ be a semantics that accounts for the size of each stack frame and a semantics that ignores the size of each stack frame, respectively. Then $space_1$ and $space_2$ are weakly equivalent, since the sizes of the stack frames constructed during evaluation of a program are bounded by the size of the program.

Although these examples are rather straightforward, they show that we can adopt a simple space semantics when we consider weak simulation. We will also show that two space semantics profiling stack space are equivalent in Section 4. The proof of this equivalence requires detailed analysis of the space semantics.

Now we extend weak simulation as a safety criterion for program transformations. Consider a program transformation $\rightsquigarrow$ between a source language and a target language that have space semantics $space(M)$ and $space'(M')$, respectively, as before.

Definition 3.5 We say that a program transformation $\rightsquigarrow$ is weakly space efficient if polynomials $f_1(x)$ and $f_2(x)$ exist such that:

$$space(M) = n \implies space'(M') \leq f_1(|M|)n + f_2(|M|)$$

for any programs M and M' with $M \rightsquigarrow M'$.

This criterion clearly falls between space efficiency and space safety. It admits that a program transformation degrades performance within a factor dependent on the size of a program. This gives us much more freedom to design program transformations used in compilers than is possible with space efficiency.

To construct a compiler that is a weakly efficient transformation as a whole, the composition of transformations must be weakly space efficient since actual compilers consist of many phases. To make the composition weakly space efficient we should restrict program transformations so that the expansion of the size of a program is limited by some polynomial.

Definition 3.6 We say that a program transformation $M \rightsquigarrow M'$ is polynomial size safe if $|M'| \leq f(|M|)$ for all programs M where $f(x)$ is a polynomial of x.

This is a natural restriction because actual compilers already avoid exponential blowup of code size. We can now construct a weakly space efficient compiler by composing weakly space efficient and polynomial size safe transformations.

Theorem 3.7 *Let $\rightsquigarrow_1$ and $\rightsquigarrow_2$ be program transformations from L_1 to L_2 and from L_2 to L_3 respectively. If $\rightsquigarrow_1$ is weakly space efficient and polynomial size safe, and $\rightsquigarrow_2$ is weakly space efficient, then their composition $\rightsquigarrow_1 \circ \rightsquigarrow_2$ is a weakly space efficient transformation from L_1 to L_3.*

Proof. Let $M \rightsquigarrow_1 \circ \rightsquigarrow_2 N$. Then P exists such that $M \rightsquigarrow_1 P$ and $P \rightsquigarrow_2 N$. By weak space efficiency we have:

$$\begin{aligned} space_2(P) &\leq f_1^1(|M|)space_1(M) + f_2^1(|M|) \\ space_3(N) &\leq f_1^2(|P|)space_2(P) + f_2^2(|P|) \end{aligned}$$

By polynomial size safety we have $|P| \leq g(|M|)$. Then:

$$space_3(N) \leq f_1^2(g(|M|))(f_1^1(|M|)space_1(M) + f_2^1(|M|)) + f_2^2(g(|M|))$$

Here, $f_1^2(g(x))f_1^1(x)$ and $f_1^2(g(x))f_2^1(x) + f_2^2(g(x))$ are clearly polynomials of x. □

This proof clarifies why we adopted a polynomial instead of a linear function in the definition of weakly efficient transformation. Even if two transformations are bounded by linear functions of the size of a program, their composition is not necessarily bounded by some linear function.

4 Weak equivalence on stack space

In this section we consider two space semantics profiling stack space for a simple call-by-value functional language. One semantics models evaluation by an interpreter and the other models evaluation based on compilation. Both semantics properly model tail calls. We show that although they allocate different numbers of stack frames during evaluation, the semantics are weakly space equivalent. Furthermore, it is shown that A-normalization preserves stack space for the second semantics.

$$E \vdash c \downarrow_1 c \quad E \vdash x \downarrow_1 E(x) \quad E \vdash \lambda x.M \downarrow_1 \langle \mathsf{cl}\, E, x, M \rangle$$

$$\frac{E \vdash M_1 \downarrow_l \langle \mathsf{cl}\, E', x, M \rangle \quad E \vdash M_2 \downarrow_m v_2 \quad E'[v_2/x] \vdash M \downarrow_n v}{E \vdash M_1 M_2 \downarrow_{\mathsf{max}(l+1, m+1, n)} v}$$

Fig. 1. Operational semantics profiling stack size (interpreter-based)

4.1 Equivalence of two semantics profiling stack space

We consider the following untyped call-by-value λ-calculus with a constant c:

$$M ::= x \mid c \mid \lambda x.M \mid M_1 M_2$$

We define two space semantics $space_1^\lambda(M)$ and $space_2^\lambda(M)$ by deductive systems. For the definition of the deductive systems, we first define values: a value v is either a constant c or a closure $\langle \mathsf{cl}\, E, x, M \rangle$ consisting of an environment E mapping variables to values, a variable and an expression.

$$v ::= c \mid \langle \mathsf{cl}\, E, x, M \rangle$$

The space semantics $space_1^\lambda(M)$ models evaluation by an interpreter and is defined by the deductive system given in Figure 1.

$$space_1^\lambda(M) = n \text{ iff } \emptyset \vdash M \downarrow_n v$$

The space semantics $space_2^\lambda(M)$ models stack space required for execution based on compilation and is defined by the deductive system given in Figure 2.

$$space_2^\lambda(M) = n \text{ iff } \emptyset \vdash_2^t M \downarrow_n v$$

The deductive system is defined mutually inductively by the following two judgments: $E \vdash_2^t M \downarrow_n v$ models execution at tail call positions and $E \vdash_2^n M \downarrow_n v$ models execution at non-tail call positions. The application at a tail call position does not allocate a new stack frame. In the figure, we write $E \vdash_2 M \downarrow_n v$ for $E \vdash_2^t M \downarrow_n v$ and $E \vdash_2^n M \downarrow_n v$.

We have shown that the two semantics $space_1^\lambda(M)$ and $space_2^\lambda(M)$ are weakly equivalent.

Theorem 4.1 *If $\emptyset \vdash M \downarrow_i v$ and $\emptyset \vdash_2^t M \downarrow_{i'} v$, then $i' + 1 \leq i \leq |M| \cdot (i' + 1)$.*

To prove this theorem we must generalize the claim so that non-empty environments can be treated. To treat non-empty environments we define the

$$E \vdash_2 c \downarrow_0 c \quad E \vdash_2 x \downarrow_0 E(x) \quad E \vdash_2 \lambda x.M \downarrow_0 \langle \mathsf{cl}\ E, x, M\rangle$$

$$\frac{E \vdash_2^n M_1 \downarrow_l \langle \mathsf{cl}\ E', x, M\rangle \quad E \vdash_2^n M_2 \downarrow_m v_2 \quad E'[v_2/x] \vdash_2^t M \downarrow_n v}{E \vdash_2^n M_1 M_2 \downarrow_{\max(l,m,n+1)} v}$$

$$\frac{E \vdash_2^n M_1 \downarrow_l \langle \mathsf{cl}\ E', x, M\rangle \quad E \vdash_2^n M_2 \downarrow_m v_2 \quad E'[v_2/x] \vdash_2^t M \downarrow_n v}{E \vdash_2^t M_1 M_2 \downarrow_{\max(l,m,n)} v}$$

Fig. 2. Operational semantics profiling stack size (compiler-based)

size of a value and an environment as follows:

$$\begin{aligned} |c|_v &= 0 \\ |\langle \mathsf{cl}\ E, \lambda x.M\rangle|_v &= \max(|E|_v, |M|_v) \\ |E|_v &= e \max\{|E(x)|_v \mid x \in Dom(E)\} \end{aligned}$$

Then the theorem is generalized to the following lemma. This lemma is proved by induction on the derivation of evaluation.

Lemma 4.2 *Let K be a constant such that $|M| \leq K$ and $|E|_v \leq K$.*

(i) *If $E \vdash M \downarrow_i v$ and $E \vdash_2^t M \downarrow_{i'} v$, then $i' + 1 \leq i \leq K \cdot (i' + 1)$.*

(ii) *If $E \vdash M \downarrow_i v$ and $E \vdash_2^n M \downarrow_{i'} v$, then $i' \leq i \leq K \cdot i' + |M|$.*

4.2 Preservation of stack space by A-normalization

In this section we show that A-normalization preserves stack space given by $space_2^\lambda(M)$ and thus $space_2^\lambda(M)$ actually models execution based on compilation. This also shows that A-normalization is weakly space efficient with respect to $space_1^\lambda(M)$.

We define the syntax of the language of A-normal forms as follows:

$$\begin{aligned} &\textit{Values} && V ::= x \mid \lambda x.M \\ &\textit{Expressions} && M ::= V \mid V_1 V_2 \mid \texttt{let}\ x = V_1 V_2\ \texttt{in}\ M \end{aligned}$$

The application $V_1 V_2$ represents tail calls and the application in $\texttt{let}\ x = V_1 V_2\ \texttt{in}\ M$ represents non-tail calls.

The semantics of this language is naturally given by the C_aEK Machine defined in Figure 3 [5]. In this operational semantics continuation clearly

$$\begin{aligned} \mathit{State}\ S &= \langle M, E, K\rangle \\ \mathit{Continuation}\ K &= \mathsf{stop} \mid \langle \mathsf{ar}\ x, M, E, K\rangle \end{aligned}$$

Transition Rules:

$$\begin{aligned} \langle v, E, \langle \mathsf{ar}\ x, M, E', K'\rangle\rangle &\mapsto \langle M, E'[\gamma(v,E)/x], K'\rangle \\ \langle \mathtt{let}\ x = V_1V_2\ \mathtt{in}\ M, E, K\rangle &\mapsto \langle M', E'[V_2/x], \langle \mathsf{ar}\ x, M, E, K'\rangle\rangle \\ &\qquad \mathit{where}\ \gamma(V_1, E) = \langle \mathsf{cl}\ x, M', E'\rangle \\ \langle V_1V_2, E, K\rangle &\mapsto \langle M', E'[V_2/x], K\rangle \\ &\qquad \mathit{where}\ \gamma(V_1, E) = \langle \mathsf{cl}\ x, M', E'\rangle \end{aligned}$$

$$\gamma(V, E) = \begin{cases} E(x) & \text{if } V \equiv x \\ \langle \mathsf{cl}\ x, M, E\rangle & \text{if } V \equiv \lambda x.M \end{cases}$$

Fig. 3. The C_aEK Abstract Machine

corresponds to stack. The size of continuation is naturally defined as follows:

$$\begin{aligned} \mathit{size}(\mathsf{stop}) &= 0 \\ \mathit{size}(\langle \mathsf{ar}\ x, M, E, K\rangle) &= \mathit{size}(K) + 1 \end{aligned}$$

In this definition we ignore the size of each frame because it is bounded by the size of the program and we discuss weak space efficiency in this paper. To discuss space efficiency it is natural to count the number of free variables of M.

The stack space of state $\langle M, E, K\rangle$ is defined by $\mathit{size}(K)$. Then we define the space semantics of this language as follows: $\mathit{space}_A(M) = n$ if $\langle M, \emptyset, \mathsf{stop}\rangle \mapsto^* \langle V, E'', \mathsf{stop}\rangle$ and n is the maximum size of the states in the transition.

A-normalization can be defined as one pass translation [5,4]. In the following definition, we use a two-level lambda calculus where $\overline{\lambda}$ and $\overline{@}$ are meta-level abstraction and application. $||M||_{AK}$ translates expressions at non-tail call positions and $||M||'_A$ translates expressions at tail call positions. The entire program is translated by $||M||'_A$.

$$\begin{aligned}|x|_A &= x\\ |\lambda x.M|_A &= \lambda x.||M||'_A\end{aligned}$$

$$\begin{aligned}||V||_A\kappa &= k\overline{@}|V|_A\\ ||M_1M_2||_A\kappa &= ||M_1||_A(\overline{\lambda}x_1.||M_2||_A(\overline{\lambda}x_2.\mathtt{let}\ z = x_1x_2\ \mathtt{in}\ \kappa\overline{@}z))\\ ||v||'_A &= |v|_A\\ ||M_1M_2||'_A &= ||M_1||_A(\overline{\lambda}x_1.||M_2||_A(\overline{\lambda}x_2.x_1x_2))\end{aligned}$$

Then it is shown that the stack space required for execution is preserved by A-normalization.

Theorem 4.3 *If $space_2^\lambda(M) = n$, then $space_A(||M||'_A) = n$.*

This theorem shows that $space_2^\lambda(M)$ models the stack space required for execution based on compilation. Furthermore, since $space_2^\lambda(M)$ is weakly equivalent to $space_1^\lambda(M)$, it is enough to consider $space_1^\lambda(M)$ even when we consider weak efficiency of program transformations with respect to execution based on compilation.

5 Local transformations

In this section we discuss the connection between our new criterion and the properties of local program transformations. We show that some class of local transformations induces weakly space efficient transformations.

Based on the classification of local transformations by Gustavsson and Sands [7] we define two classes of local transformations.

Definition 5.1 Let R be a relation on terms of a programming language.

(i) We say that R is a strong improvement relation if we have:

$$space(\mathbb{C}[M]) = n \implies space(\mathbb{C}[N]) \leq n$$

for all $(M, N) \in R$ and all context $\mathbb{C}[\cdot]$ producing a whole program for M and N.

(ii) We say that R is a weak improvement relation if there exists some linear function f such that the following holds for all $(M, N) \in R$ and all context $\mathbb{C}[\cdot]$ producing a whole program for M and N.

$$space(\mathbb{C}[M]) = n \implies space(\mathbb{C}[N]) \leq f(n)$$

We should remark that there is one subtle difference in our definition of weak improvement from that of Gustavsson and Sands. They defined a single weak improvement relation as follows. $M \underset{\approx}{\rhd} N$ if some linear function f exists such that for all contexts $\mathbb{C}[\cdot]$ the following holds:

$$space(\mathbb{C}[M]) = n \implies space(\mathbb{C}[N]) \leq f(n)$$

The relation $\underset{\approx}{\rhd}$ is the union of all the weak improvement relations. However, this relation $\underset{\approx}{\rhd}$ itself is not a weak improvement relation in our sense.

To discuss the connection between these properties and the properties of global transformations, we first define the induced global transformation $M \rightsquigarrow_R N$ as follows: $M \rightsquigarrow_R N$ if some $\mathbb{C}[\cdot]$, M' and N' exist such that $M = \mathbb{C}[M']$, $N = \mathbb{C}[N']$, and $(M', N') \in R$. Then we immediately obtain the following theorem.

Theorem 5.2 *If R is a weak or strong improvement relation, $\rightsquigarrow_R$ is space efficient.*

On the other hand, the relation $\underset{\approx}{\rhd}$ does not induce a space efficient transformation. This is because there is no single linear function f such that $space(\mathbb{C}[N]) \leq f(space(\mathbb{C}[M]))$ for all $M \underset{\approx}{\rhd} N$.

The theorem above is still not enough to use a local transformation in a compiler. In a compiler we usually apply local transformations n times in one phase of a compiler where n is proportional to the size of a program. Even for such composition, a strong improvement relation induces a space efficient transformation.

Theorem 5.3 *If R is a strong improvement relation, $\rightsquigarrow_R^*$ is space efficient.*

On the other hand, a weak improvement relation does not necessarily induce a weakly space efficient transformation. Consider the following sequence of transformations where R is a weak improvement relation with $space(\mathbb{C}[N]) \leq k space(\mathbb{C}[M])$ for all $(M, N) \in R$.

$$M_0 \rightsquigarrow_R M_1 \rightsquigarrow_R M_2 \rightsquigarrow_R M_3 \rightsquigarrow_R \ldots \rightsquigarrow_R M_n$$

The space requirement of M_n can be calculated as follows:

$$space(M_2) \leq k space(M_1)$$

$$space(M_3) \leq k space(M_2) \leq k^2 space(M_1)$$

$$space(M_n) \leq k^n space(M_1)$$

Then it is clear that there is no single linear function f such that:

$$space(N) \leq f(space(M))$$

for $M \rightsquigarrow^*_R N$. Even if we restrict the number of repetitions to $|M_0|$, k^n is not a polynomial of $|M_0|$. Thus, it is not even weakly space efficient.

We therefore must consider stricter conditions on local transformations. In the following definition a local transformation is permitted to add only a constant amount of extra space.

Definition 5.4 We say that R is a semi-strong improvement relation if some constant k exists such that:

$$space(\mathbb{C}[M]) = n \implies space(\mathbb{C}[N]) \leq n + k$$

for all $(M, N) \in R$ and all context $\mathbb{C}[\cdot]$ producing a whole program for M and N.

It can be shown that this class of local transformations induces weakly space efficient transformations if the number of applications of the transformation is limited by the size of a source program. We write $M \mapsto_R N$ if $M \rightsquigarrow^n_R N$ where $n \leq |M|$.

Theorem 5.5 *If R is a semi-strong improvement relation, $M \mapsto_R N$ is weakly efficient.*

Although this theorem relates a semi-strong improvement relation to weakly efficient transformations, semi-strong improvement relations seem too restrictive. There are many useful transformations R that are not semi-strong improvement relations, but $\mapsto_R$ seem weakly space efficient. The following transformation is an example.

$$\lambda x.\texttt{let}\ y = M\ \texttt{in}\ N \Rightarrow \texttt{let}\ y = M\ \texttt{in}\ \lambda x.N$$

We have not shown formally that this transformation is weakly space efficient. For such proof we think that we require further study on the connections between global transformations and local transformations.

6 Discussion and future work

We have shown weak efficiency only for stack space for two semantics of a simple functional language. It will not be very difficult to deal with execution time or heap space. For example, the proof that the CPS transformation is space efficient [8] can easily be modified to show that the CPS transformation is weakly space efficient with respect to a simpler space semantics of the source language that ignores the sizes of closures and stack frames.

We have shown no examples of local program transformations that are weak improvement relations or semi-strong improvement relations. We are planning to show that various optimizations formalized as local program transformations have these kinds of properties. In this area, Gustavsson and Sands have developed a theory of space improvement relations for call-by-need programming languages and have shown that several local transformations are weak improvements [7]. Their work will be also applicable to call-by-value languages.

We think that the framework we have developed in this paper requires further refinement. For example, although intuitively clear, it is not proved that space safety, weak space efficiency and space efficiency ensure that the space complexity of programs is preserved. Bakewell and Runciman discussed these kinds of issues more formally in their study on the comparison of space usage of lazy evaluators [2]. They modeled lazy evaluators by graph rewriting systems. This kind of uniform formalization of semantics may help develop a theory of safe program transformations.

There is an implementation strategy of ML that is not space efficient, but is space safe. That is the implementation strategy that uses types as parameters at runtime [10,13]. This is because the extra work and space necessary for type parameters cannot be bounded by any constant. Furthermore, this implementation strategy is not even weakly space efficient, because the types appearing in the typing derivation of a program may have a size exponential to the size of the program. However, if we take the sum of the size of a program and the maximum size of types appearing in typing derivation of the program as the size of the program, this implementation strategy can be considered weakly space efficient. By choosing the definition of the size of a program in this way we can control the class of transformations that can be used in compilers of the language.

Acknowledgement

This work is partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Encouragement of Young Scientists of Japan No. 11780216, 1999. We would like to thank anonymous reviewers for their many helpful comments and suggestions.

References

[1] Appel, A. W., "Compiling with Continuation," Cambridge University Press, 1992.

[2] Bakewell, A. and C. Runciman, *A model for comparing the space usage of lazy evaluators*, in: *2nd International Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, 2000.

[3] Blelloch, G. E. and J. Greiner, *A provably time and space efficient implementation of NESL*, in: *Proc. of ACM SIGPLAN International Conference on Functional Programming*, 1996, pp. 213–225.

[4] Danvy, O. and A. Filinski, *Representing control: a study of the CPS transformation*, Mathematical Structures in Computer Science **2** (1992), pp. 361 – 391.

[5] Flanagan, C., A. Sabry, B. F. Duba and M. Felleisen, *The essence of compiling with continuations*, in: *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1993, pp. 237–247.

[6] Greiner, J. and G. E. Blelloch, *A provably time-efficient parallel implementation of full speculation*, in: *Proc. of ACM Symposium on Principles of Programming Languages*, 1996, pp. 309 – 321.

[7] Gustavsson, J. and D. Sands, *A foundation for space-safe transformations of call-by-need programs*, in: *Proc. of the Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS99)*, ENTCS **26**, 1999.

[8] Minamide, Y., *A space-profiling semantics of call-by-value lambda calculus and the CPS transformation*, in: *Proc. of the Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS99)*, ENTCS **26**, 1999.

[9] Minamide, Y. and J. Garrigue, *On the runtime complexity of type-directed unboxing*, in: *Proc. of ACM SIGPLAN International Conference on Functional Programming*, 1998, pp. 1–12.

[10] Ohori, A. and N. Yoshida, *Type inference with rank 1 polymorphism for type-directed compilation of ML*, in: *Proc. of ACM SIGPLAN International Conference on Functional Programming*, 1999, pp. 160– 171.

[11] Santos, A. L., "Compilation by Transformation in Non-strict Functional Languages," Ph.D. thesis, Department of Computing Science, University of Glasgow (1995).

[12] Shao, Z., *Flexible representation analysis*, in: *Proc. of ACM SIGPLAN International Conference on Functional Programming*, 1997, pp. 85 – 98.

[13] Tarditi, D., G. Morrisett, P. Cheng, C. Stone, R. Harper and P. Lee, *TIL: A type-directed optimizing compiler for ML*, in: *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1996, pp. 181–192.

Minimised Geometric Buchberger Algorithm for Integer Programming

Qiang Li Yi-ke Guo[*] Tetsuo Ida[**] John Darlington[*]

Research Center
PFU Limited, Japan
[*]Department of Computing
Imperial College, U.K.
[**]Institute of Information Sciences and Electronics
University of Tsukuba, Japan

Abstract

Recently, various algebraic integer programming (IP) solvers have been proposed based on the theory of Gröbner bases. The main difficulty of these solvers is the size of the Gröbner bases generated. In algorithms proposed so far, large Gröbner bases are generated by either introducing additional variables or by considering the generic IP problem $IP_{A,C}$. Some improvements have been proposed such as Hosten and Sturmfels' method (GRIN) designed to avoid additional variables and Thomas' truncated Gröbner basis method which computes the reduced Gröbner basis for a specific IP problem $IP_{A,C}(b)$ (rather than its generalisation $IP_{A,C}$). In this paper we propose a new algebraic algorithm for solving IP problems. The new algorithm, called *Minimised Geometric Buchberger Algorithm*, combines Hosten and Sturmfels' GRIN and Thomas' truncated Gröbner basis method to compute the fundamental segments of an IP problem $IP_{A,C}$ directly in its original space and also the truncated Gröbner basis for a specific IP problem $IP_{A,C}(b)$. We have carried out experiments to compare this algorithm with others such as the geometric Buchberger algorithm, the truncated geometric Buchberger algorithm and the algorithm in GRIN. These experiments show that the new algorithm offers significant performance improvement.

1 Introduction

In this paper, we consider the following IP problem:

$$IP_{A,C}(b) = \min\{Cx : Ax = b, x \in \mathbb{N}^n\}$$

where C is an objective vector in $\mathbb{R}^n$, A is an $m \times n$ matrix of integers, and b is a vector in $\mathbb{Z}^m$. We use $IP_{A,C}$ to denote a generic IP problem where b is not taken into account.

Recently, the tools of commutative algebra and algebraic geometry have brought new insights to IP via the theory of Gröbner bases [3]. The key idea is to encode an IP problem into a special ideal associated with the constraint matrix A and the cost (objective) function Cx. An important property of such an encoding is that its Gröbner bases correspond directly to the test sets of the IP problem. Thus, by employing an algebraic package such as MACAULAY [6] or MAPLE [1], the test sets of the IP problem can be directly computed. Using a proper test set (such as the minimal test set which corresponds directly to the reduced Gröbner basis of the encoded ideal), the optimal value of the cost function can be computed by constructing a monotonic path from the initial non-optimal solution of the problem to the optimal solution. Thus, IP problems can be solved in a similar fashion to the simplex method for linear programming without using intensive heuristic searching algorithms.

There are two strategies for encoding an IP problem into a special ideal.

- Indirect encoding: encoding by adding extra variables.
- Direct encoding: encoding without adding extra variables.

The first strategy was originally given by Conti and Traverso [8]. The scheme involves two encoding mechanisms: encoding the cost function of $IP_{A,C}$ into a linear order and encoding the coefficient matrix into a polynomial ideal. With this translation, IP problems are transformed into solving the subalgebra membership problems (See section 2.1 in detail).

In [12], Thomas proposed a geometric interpretation of Conti-Traverso method. The key idea of Thomas' *Geometric Buchberger Algorithm* (GBA) is to relate the Gröbner bases of the encoded polynomial ideal of an IP problem $IP_{A,C}$ to the notion of test sets for $IP_{A,C}$. Each binomial is now directly interpreted as directed line segments, i.e. vectors, in a lattice of all feasible solutions of $IP_{A,C}$. The Buchberger algorithm is then directly applied to a directed graph, where nodes of the graph are lattice points corresponding to feasible solutions of $IP_{A,C}$ and the edges at the beginning correspond to the input basis of the binomial ideal I. Finding the reduced Gröbner basis amounts to rebuilding the graph such that the edges correspond to the members of the reduced Gröbner basis of I, which can be geometrically understood as a test set of the IP problem. Thus, by this graph, an optimal solution of $IP_{A,C}$ can be found along the directed path in the graph from a feasible solution. Thomas' work provides not only a succinct understanding of an algebraic IP solver but also a practical computational procedure for its implementation. In particular, this "generate and test" approach provides great inherent parallelism. In [9], we presented a parallel implementation of GBA on a Fujitsu AP1000+. The experiment showed that the algebraic approach towards IP provides a very promising mechanism. It also showed that the new method can be improved in various ways.

In the above strategy, the first problem is that the strategy is applied to an extended IP (EIP) with additional variables (y), of the form:

$$\min\{My + Cx\}$$

subject to $Iy+Ax = b$ and $(y, x) \in \mathbb{Z}^{m+n}$. I is the $m \times m$ identity matrix and $M \in \mathbb{R}^m$ is a vector whose components have large magnitude (it is assumed, without loss of generality, that all entries in A, C and b are nonnegative integers). In practice the additional variables will lead to a considerable increase in the space and time requirements of the algorithms considered.

The second problem is that the test set generated by both algorithms are generic in the sense that it is only determined by A and C for an IP problem $IP_{A,C}$. Thus, the search space for computing the reduced Gröbner basis for such a generalised problem is quite large. In [13], Thomas proposed the "Truncated Göbner basis" method by fixing b to reduce the cardinality of the reduced Gröbner basis, but the size of Gröbner basis computed by the algorithm is still not optimal since the basis is for the EIP w.r.t $IP_{A,C}(b)$, not for $IP_{A,C}(b)$ itself. So, many vectors in the reduced Gröbner basis are needed to move from an initial solution of EIP to an initial solution of IP.

The second strategy was given by Hosten and Sturmfels. In [14], Hosten and Sturmfels proposed an algorithm in which a set of fundamental segments of $IP_{A,C}$ can be computed without going through EIP. This algorithm starts with a basis for the lattice $ker(A)$ and then proceeds to refine this to a set of fundamental segments for $IP_{A,C}$. But the basis constructed by the algorithm is not a truncated basis since the vector b is not taken into account. Thus, the efficiency is still a problem when the method is applied to large scale IP problems due to the complexity of the Buchberger algorithm.

In this paper we propose a new algebraic algorithm for solving integer programming. The new algorithm, called the *Minimised Geometric Buch-*

berger Algorithm (MGBA), combines Hosten and Sturmfels' method GRIN and Thomas' truncated Gröbner basis method to compute the fundamental segments of an IP problem $IP_{A,C}$ directly in its original space and also the truncated Gröbner basis for the fixed b.

This paper is organized as follows. In section 2 and section 3, we give a brief sketch of the approaches of strategy 1 and strategy 2, respectively. Section 4 introduces the idea of the truncated Gröbner basis method. In section 5, we present the new Buchberger algorithm to compute a test set for IP. We also show some computational results to compare MGBA with other algorithms such as the algorithm in GRIN, the geometric Buchberger algorithm and the truncated geometric Buchberger algorithm in section 6. Finally, we draw a conclusion in section 7.

2 Strategy 1: indirect encoding

In this strategy, we first translate IP into the extended IP (EIP) by introducing additional variables y:

$$\text{EIP(b)} = \min\{My + Cx : \; Iy + Ax = b, \; (y, x) \in \mathbb{N}^{m+n}\}$$

where I is the identity matrix, $M \in \mathbb{N}^m$ is a vector whose components have large magnitude, A is an $m \times n$ matrix of non-negative integers, and b is a vector of non-negative integers. We use $\text{EIP}_{A,C}$ to denote the whole family of these integer programs, with fixed A and C, but varying right-hand side b.

From EIP(b), we can see all of the programs in it are feasible: they have the obvious solution $x = 0$, $y = b$. An optimal solution will satisfy $y = 0$, $x = x_o$ if the $IP_{A,C}(b)$ is feasible, because the components of M are sufficiently large comparing to C. If $IP_{A,C}(b)$ is infeasible, then EIP(b) has an optimal solution with $y > 0$. The value of M will not affect the computation of x and y if M is greater than C. So, in actual computation, we can select any integers for M which are much greater than the maximum in C, for instance we assign M=100 which is much greater than the maximum in C 3 in example 2.1 and example 2.2.

There are two approaches as follows to solve the EIP(b).

2.1 Algebraic approach

The algebraic approach was proposed by Conti and Traverso [8]. The scheme involves two encoding mechanisms:

1. Encoding the cost function $(M\ C)$ into a linear order on $\mathbb{Z}^{m+n}$. This can be done by choosing an arbitrary term order, such as lexicographic order $\prec_o$ and use it as a "tie breaker" on the points that have the same value under function $(M\ C)$; That is, we define $\prec_{M,C}$ to encode $(M\ C)$ as a linear order on $\mathbb{Z}^{m+n}$:

$$x_1 \prec_{M,C} x_2 \iff \begin{cases} (M\ C)x_1 < (M\ C)x_2 \\ (M\ C)x_1 = (M\ C)x_2 \quad x_1 \prec_o x_2 \end{cases}$$

2. Encoding $(I\ A)$ into a polynomial ideal:

$$I = \langle y^{Ae_j} - x_j : j = 1, \ldots, n \rangle$$

where e_j is the jth unit vector on $\mathbb{Z}^n$.

Actually, Ae_j is a projection of jth column of A. So,

$$I = \langle y^{a^1} - x_1, \ldots, y^{a^n} - x_n \rangle$$

where $a^1, \ldots, a^n$ are the columns of A.

Let $\phi:\ k[x_1, ..., x_n] \longrightarrow k[y_1, ..., y_n]$ be the image defined by

$$x_j \longmapsto y_1^{a_{1j}} \cdots y_m^{a_{mj}}$$

So, with this translation, IP problems are transformed into solving the subalgebra membership problem for determining whether y^b is in the image of ϕ.

Let $\mathcal{G}$ be the reduced Gröbner basis of I with respect to $\prec_{M,C}$. According to Shannon and Sweedler subalgebra membership theorem [15], $g \in k[y_1, ..., y_m]$ is in the image of ϕ iff the remainder $r_{\mathcal{G}}(g)$ of g on division by $\mathcal{G}$ is in $k[x_1, ..., x_n]$. Thus, first we compute the reduced Gröbner basis of I. Then we divide y^b by $\mathcal{G}$. If $r_{\mathcal{G}}(y^b) \in k[x_1, ..., x_n]$ then the remainder is a monomial whose exponent vector is the optimal solution to the problem $IP_{A,C}(b)$, Otherwise $IP_{A,C}(b)$ has no feasible solution.

Example 2.1

$$\text{IP} : \min\{x_1 + x_2 : \ \ 3x_1 + 2x_2 = 6,\ (x_1, x_2 \in \mathbb{N})\}$$

Firstly, we translate it into extended IP:

$$\text{EIP} : \min\{100y + x_1 + x_2 : \ \ y + 3x_1 + 2x_2 = 6,\ (y, x_1, x_2 \in \mathbb{N})\}$$

Encoding $(I\ A)$ into the polynomial ideal:

$$\mathrm{I} = \langle y^{(3\ 2)(1\ 0)^T} - x_1, y^{(3\ 2)(0\ 1)^T} - x_2 \rangle = \langle y^3 - x_1, y^2 - x_2 \rangle$$

Computing the reduced Gröbner basis:

$$\mathcal{G} = \{y^2 - x_2, yx_2 - x_1, yx_1 - x_2^2, x_2^3 - x_1^2\}$$

Computing the optimal solution of $IP_{A,C}(b)$:

$$r_{\mathcal{G}}(y^6) = x_1^2$$

So, the optimal solution is: $x_1 = 2, x_2 = 0$.

2.2 Geometric approach

The Geometric Buchberger Algorithm (GBA), proposed by Thomas [12], is based on a geometric interpretation of above algebraic approach. The key idea is to relate the Gröbner basis of encoded polynomial ideal to the notion of test set of IP. A test set for an IP problem $IP_{A,C}$ is a set of vectors in $\mathbb{Z}^n$ such that for each non-optimal solution u to a problem in this family, there is at least one element g in this set such that $u - g$ has an improved cost value as compared to u.

Definition 2.1 A test set of $IP_{A,C}$, $\mathcal{G} \subseteq ker(A) \cap \mathbb{N}^n$ such that for every feasible solution u of $IP_{A,C}(b)$ either u is the optimal solution of $IP_{A,C}(b)$ or there exists $g \in \mathcal{G}$ such that $u - g$ is also the solution and $Cu > C(u - g)$.

Thus, a test set for $IP_{A,C}$ provides an obvious algorithm to find the optimal solution of a feasible problem $IP_{A,C}(b)$. That is, starting an initial feasible solution u, a better solution can be found by moving from u a unit step along -g in the test set until the optimal is reached.

By the theorem in [12], the unique minimal test set of EIP is the reduced Gröbner basis $\mathcal{G}_{M,C}$ of the polynomial ideal $I = \langle y^{Ae_j} - x_j : j = 1, ..., n \rangle$. Thus, the optimal solution of $IP_{A,C}(b)$ can be therefore computed by using $\mathcal{G}_{M,C}$ to improve the obvious feasible solution $(b, 0)$ to optimum $(0, v)$ of EIP(b). Here, we are really dealing with this algorithm operating on lattice vectors.

Segment vector : Geometric Polynomial Each binomial in the ideal can be interpreted as a directed line segment, i.e. a vector by reading off its exponents. For example, we translate $x_1^2x_3x_5^3 - x_2^3x_4x_6^2$ directly into the vector $[(2,0,1,0,3,0),(0,3,0,1,0,2)]$. For a vector $d=[\alpha,\beta]$, $\alpha,\beta\in\mathbb{N}^{m+n}$, the tail $d^t=\alpha$ of the vector is more expensive than the head $d^h=\beta$ according to the order $\prec_{M,C}$ defined as follow:

$$\beta \prec_{M,C} \alpha \iff \begin{cases} \beta(M\ C)^T < \alpha(M\ C)^T \\ \beta(M\ C)^T = \alpha(M\ C)^T \quad \beta \prec_o \alpha \end{cases}$$

The generating set of the ideal I, which is called *fundamental segment*, can be easily constructed by translating each binomial $y_1^{a_{1j}},\ldots,y_m^{a_{mj}} - x_j$ of I into a vector d_j.

S-Vector : Geometric S-Polynomial The geometric correspondence of S-Polynomial is called S-vector of two segment vectors d_1, d_2, which is computed as the difference of the heads of two vectors yielded by translated d_1, d_2 into a fiber on which their tails meet. Here a fiber is defined as a set of feasible solutions to an IP problem. Specially, given a right-hand side vector b, the set of feasible solutions to $IP_{A,C}(b)$ is identified as b-fiber (refer to [12] for the detailed definition). The computation of S-vector is shown in Fig.2.1. Here the fiber is 3-fiber.

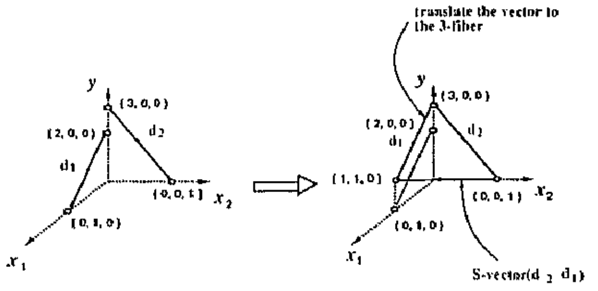


Figure 2.1: Computing S-Vector

Geometric Reduction: A vector f is reduced by a set of vectors $\{f_i\}$. If the tail of f can be reduced by f_i ($f_i^t \leq f^t$), then translate f_i so that f_i^t meets f^t and replace f by $\bar{f}$ joining f^h and f_i^h of translated f_i directed from the expensive to cheap end; if the head of f can be reduced by f_i ($f_i^t \leq f^h$), then translate f_i so that f_i^t meets f^h and replace f by $\bar{f}$ joining f^t and f_i^h of translated f_i directed from the expensive to cheap end. Repeat this procedure until there exists no such f_i. Actually, the geometric interpretation of reducing a vector f by a set of vectors $\{f_i\}$ is constructing a path from the tail of f to the head of $\bar{f}$, as in Fig.2.2.

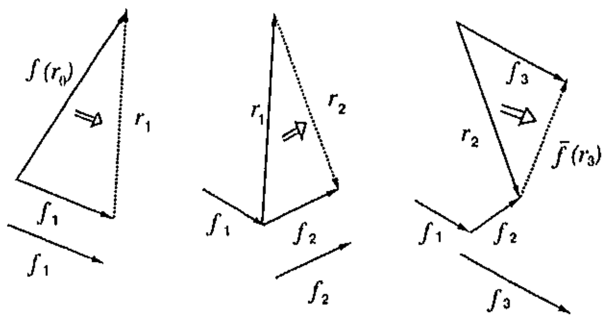


Figure 2.2: Geometric Reduction

Algorithm 2.1 Geometric Buchberger Algorithm:
The algorithm computes the reduced Gröbner basis of the ideal I with the term order $\prec_{M,C}$.

First Step: Construct a Gröbner basis

INPUT $F = \{d_1, ..., d_n\}$, the fundamental segments of EIP(b) directed according to $\prec_{M,C}$

SET $\mathcal{G}_{old} := \emptyset$, $\mathcal{G} := F$

REPEAT While $\mathcal{G}_{old} \neq \mathcal{G}$, repeat the following steps

$\mathcal{G}_{old} := \mathcal{G}$

(S-vector) construct the pair $g := d_i - d_j$

(reduction) reduce the vector g by the vectors in $\mathcal{G}_{old}$. If $\bar{g} \neq 0$, set $\mathcal{G} := \mathcal{G} \cup \{\bar{g}\}$.

Second Step: Construct a minimal Gröbner basis

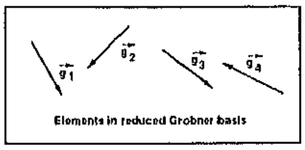
REPEAT If for some $g \in \mathcal{G}$ the tail g^t can be reduced by some $g' \in \mathcal{G} \backslash \{g\}$, then delete g from $\mathcal{G}$.

Third Step: Construct the reduced Gröbner basis

REPEAT If for some $g \in \mathcal{G}$ the head g^h can be reduced by some $g' \in \mathcal{G}\backslash\{g\}$, then replace g by the corresponding reduced vector: $\mathcal{G} := \mathcal{G}\backslash\{g\} \cup \bar{g}$.

OUTPUT $\mathcal{G}_{red} := \mathcal{G}$.

Example 2.2 (The same as Example 2.1)
EIP : $\min\{100y + x_1 + x_2 : \ y + 3x_1 + 2x_2 = 6, \ (y, x_1, x_2 \in \mathbb{N})\}$
Computing fundamental segments by interpreting binomials of I geometrically.
$\mathrm{I} = \langle y^3 - x_1, y^2 - x_2 \rangle = \langle y^3x_1^0x_2^0 - y^0x_1^1x_2^0, y^2x_1^0x_2^0 - y^0x_1^0x_2^1 \rangle$
So, the fundamental segments are:
$d_1 = [(3,0,0),(0,1,0)], d_2 = [(2,0,0),(0,0,1)]$
Computing reduced Gröbner basis $\mathcal{G}_{M,C}$:
$g_1 = [(2,0,0),(0,0,1)], \ \ g_2 = [(1,0,1),(0,1,0)]$,
$g_3 = [(1,1,0),(0,0,2)], \ \ g_4 = [(0,0,3),(0,2,0)]$.
Deriving the optimal solution of $IP_{A,C}(b)$ from the feasible solution (6,0,0) by using $\mathcal{G}_{M,C}$, we obtain the optimal solution: $x_1 = 2, x_2 = 0$, as in Fig.2.3.



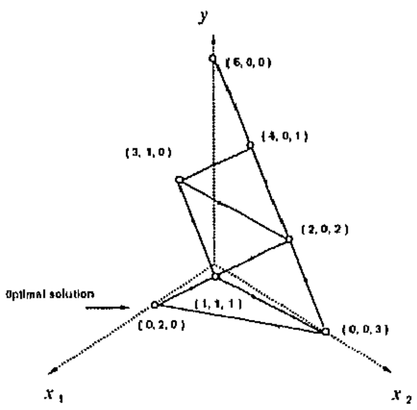


Figure 2.3: Example 2.2

3 Strategy 2: direct encoding

A typical approach of this strategy is the GRIN method. GRIN (GRöbner basis for INteger programming) is an experimental software system developed by Serkan Hosten and Bernd Sturmfels for computing the Gröbner basis of a toric ideal, in particular, for solving an IP problem using Gröbner bases. The algorithm in GRIN introduces a new method for computing the reduced Gröbner basis of the toric ideal which operates entirely in $k[x_1,...,x_n]$ rather than in the auxiliary polynomial ring $k[y_1,...,y_m,x_1,...,x_n]$. In GRIN, two algorithms are implemented. Here we discuss only one.

First we give a definition of the toric ideal.

Definition 3.1 The toric ideal I_A is a binomial ideal constructed from matrix A:

$$I_A = \langle x^u - x^v : u, v \in N^n, u - v \in ker(A)\rangle$$

The algorithm in GRIN works in three stages: stage 1 encodes an IP problem $IP_{A,C}$ into a subideal of the toric ideal I_A; stage 2 computes the toric ideal I_A from the subideal; stage 3 computes the reduced Gröbner basis of I_A with respect to cost function Cx.

Stages 1 and 3 are easy. We can encode A into a subideal of the toric ideal I_A by finding an arbitrary lattice basis for $ker(A)$ with some methods such as Hermit normal form algorithm [7] or Smith normal form algorithm [2]. After we get the toric ideal I_A, we can use Buchberger algorithm to compute the reduced Gröbner basis for I_A. Here we focus on stage 2 computing the toric ideal I_A from a subideal.

Definition 3.2 If f is a polynomial in $k[x_1,...,x_n]$ and $J \subset k[x_1,...,x_n]$ is an ideal, then the following two subsets of $k[x_1,...,x_n]$ are again ideals:

$$(J : f) = \{g \in k[x_1,...,x_n] : fg \in J\},$$

$$(J : f^\infty) = \{g \in k[x_1,...,x_n] : f^r g \in J \ \ for \ \ some \ \ r \in \mathbb{N}\}$$

A basic formula involving ideal quotients is $(I : fg) = ((I : f) : g)$. A general method for computing Gröbner bases of the ideals from generators of J can be found in [5]. If J is a homogeneous ideal and f is one of the variables, say, $f = x_n$, then the algorithm for computing the Gröbner basis of the ideal from J is provided by the following lemma in [4].

First we give a definition of the graded reverse lexicographic order.

Definition 3.3 (Graded Reverse Lexicographic Order $\succ_{grevlex}$)
Let $\alpha, \beta \in \mathbb{N}^n$, we say $\alpha \succ_{grevlex} \beta$ if

$|\alpha| = \sum_{i=1}^n \alpha_i > |\beta| = \sum_{i=1}^n \beta_i$, or $|\alpha| = |\beta|$ and $\alpha \succ_{revlex} \beta$
where $\succ_{revlex}$ is the reverse lexicographic order [5].

We call this order graded reverse lexicographic order.

Lemma 3.1 Fix the graded reverse lexicographic order induced by $x_1 > \ldots > x_n$, and let $\mathcal{G}$ be the reduced Gröbner basis of a homogeneous ideal $J \subset k[x_1, \ldots, x_n]$. Then the set

$$\mathcal{G}' = \{f \in \mathcal{G} : x_n \textit{ does not divide } f\} \cup \{f/x_n : f \in \mathcal{G} \textit{ and } x_n \textit{ divides } f\}$$

is a Gröbner basis of $(J : x_n)$. A Gröbner basis of $(J : f^\infty)$ is obtained by dividing each element $f \in \mathcal{G}$ by the highest power of x_n that divides f.

The term order used in Lemma 3.1 makes sense whenever the ideal J is homogeneous with respect to some positive grading $deg(x_i) = d_i > 0$. By iterating the Gröbner basis computation n times with respect to different graded reverse lexicographic orders, that is, by applying Lemma 3.1 one variable at a time, one can compute the ideal quotient

$$(J : (x_1 x_2 \cdots x_n)^\infty) = ((\cdots(J : x_1{}^\infty) : x_2{}^\infty)\cdots) : x_n{}^\infty)$$

So, if we find the relationship between the toric ideal I_A and ideal quotient, we can compute I_A and the reduced Gröbner basis of I_A. Let $B \subset ker(A)$, we associate a subideal of I_A:

$$J_B := \langle x^{v^+} - x^{v^-} : v \in B \rangle$$

where $v = v^+ - v^-$ is the usual decomposition into positive and negative part.

We have the following lemma whose proof is in [4].

Lemma 3.2 A subset B spans the lattice $ker(A)$ if and only if

$$(J_B : (x_1 \cdots x_n)^\infty) = I_A$$

From Lemma 3.1 and Lemma 3.3, we can prove the following proposition.

Proposition 3.1 Let $J_0 = \langle x^{v^+} - x^{v^-} : v \in B \rangle$ and $J_i = (J_{i-1} : x_i{}^\infty)$ $(i = 1, ..n)$ with the graded reverse lexicographic order by making x_i the reverse lexicographically cheapest variable. Then J_n is the toric ideal I_A.

The lemmas and proposition stated above give the following algorithm which computes a Gröbner basis of a toric ideal.

Algorithm 3.1 Algorithm 1 in GRIN

1. Find any lattice basis B for ker(A).
2. (Optional) Replace B by a reduced lattice basis B_{red}.
3. Let $J_0 := \langle x^{u^+} - x^{u^-} : u \in B_{red} \rangle$.
4. For $i = 1, ..., n$: Compute $J_i := (J_{i-1} : x_i{}^{\infty})$ using Lemma 3.1, that is, by making x_i the reverse lexicographically cheapest variable.
5. Compute the reduced Gröbner basis of $J_n = I_A$ for the desired term order. If the term order is obtained from an objective function Cx, then the computed reduced Gröbner basis is the minimal test set of $IP_{A,C}$.

Example 3.1 Let $d = 4$, $n = 8$ and consider the matrix A

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 & 0 & 1 & 4 & 5 \\ 2 & 3 & 4 & 1 & 1 & 4 & 5 & 0 \\ 3 & 4 & 1 & 2 & 4 & 5 & 0 & 1 \\ 4 & 1 & 2 & 3 & 5 & 0 & 1 & 4 \end{pmatrix}$$

STEP 1 and STEP 2 :

Compute basis for the lattice $ker(A)$. In this case we get the reduced basis B_{red} as follows.

$$B_{red} = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & -1 & 0 & -1 \\ 0 & 1 & 0 & -3 & 0 & 0 & 0 & 2 \\ 1 & 0 & 1 & 0 & -1 & 0 & -1 & 0 \\ 2 & 0 & -2 & 0 & -1 & 0 & 1 & 0 \end{pmatrix}$$

Here the basis for $ker(A)$ is expressed as a matrix whose every row is in $ker(A)$. In the new algorithm *Minimised Geometric Buchberger Algorithm* in section 5, we will give another expression of a basis for $ker(A)$ which is better than this one.

STEP 3:

By splitting the vectors of B_{red} into positive and negative parts, we get the binomial ideal J_0 associated with B_{red} :

$$J_0 = \langle x_2x_4 - x_6x_8,\ x_2x_8^2 - x_4^3,\ x_1x_3 - x_5x_7,\ x_1^2x_7 - x_3^2x_5\rangle$$

STEP 4:

In this step, we need to make eight Gröbner basis computations with respect to certain graded reverse lexicographic orders, starting with J_0. After each Gröbner basis computation we need to divide out certain variables. What we get after these eight Gröbner bases computations is a generating set of I_A.

Entering the loop in this step, we first compute the reduced Gröbner basis for J_0 with respect to the graded reverse lexicographic order that makes x_1 the cheapest variable. Here is $x_1 < x_2 < x_3 < x_4 < x_5 < x_6 < x_7 < x_8$. The result is

$G_0 = \{x_3^3x_1 - x_7^2x_1^2,\ x_4^3 - x_8^2x_2,\ x_5x_3^2 - x_7x_1^2,\ x_7x_5 - x_3x_1,\ x_8x_6 - x_4x_2\}$
Next we divide each binomial in G_0 by x_1 whenever possible. So for example $x_3^3x_1 - x_7^2x_1^2$ when divided by x_1 gives $x_3^3 - x_7^2x_1$ and none of the other can be divided by x_1(the cheapest variable in the above order).

Then we get a new set J_1 which consists of all the binomials in G_0 divided by x_1 whenever possible.

$J_1 = \langle x_3^3 - x_7^2x_1,\ x_4^3 - x_8^2x_2,\ x_5x_3^2 - x_7x_1^2,\ x_7x_5 - x_3x_1,\ x_8x_6 - x_4x_2\rangle$

Now we compute the reduced Gröbner basis G_1 for J_1 by using the graded reverse lexicographic order that makes x_2 the cheapest variable. For example we use the order $x_1 > x_8 > x_7 > x_6 > x_5 > x_4 > x_3 > x_2$. The result is

$G_1 = \{x_4^3 - x_8^2x_2,\ x_7x_5 - x_1x_3, x_8x_6 - x_4x_2,\ x_1x_7^2 - x_3^3,$
$x_1^2x_7 - x_5x_3^2,\ x_1^3x_3 - x_5^2x_3^2\}$
Dividing each binomial in G_1 by x_2 whenever possible, we get J_2:

$J_2 = \langle x_4^3 - x_8^2x_2,\ x_7x_5 - x_1x_3, x_8x_6 - x_4x_2,\ x_1x_7^2 - x_3^3,$
$x_1^2x_7 - x_5x_3^2,\ x_1^3x_3 - x_5^2x_3^2\rangle$
Then we can repeat the process, each time computing the reduced Gröbner basis G_i for J_i by using the graded reverse lexicographic order that makes x_{i+1} the cheapest variable and then dividing each binomial in G_i by x_{i+1} to get J_{i+1}. Finally we get J_8, that is the generating set of the toric ideal I_A.

$J_8 = \langle x_2^4 - x_6^3x_8,\ x_3^3 - x_7^2x_1,\ x_4x_2 - x_6x_8,\ x_4^3 - x_2x_8^2,$
$x_5x_3^2 - x_7x_1^2,\ x_5^2x_3 - x_1^3,\ x_6x_4^2 - x_2^2x_8,\ x_6^2x_4 - x_2^3,\ x_7x_5 - x_3x_1\rangle$

STEP 5:

Now, we can use J_8 as a generating set to compute the reduced Gröbner basis of I_A with a fixed term order. Here, the reduced Gröbner basis of I_A with respect to the lexicographic term order given by $x_1 > x_2 > x_3 > x_4 > x_5 > x_6 > x_7 > x_8$ equals

$$\begin{aligned}\mathcal{G} = \{&x_1^3 - x_3x_5^2,\ x_1^2x_7 - x_3^2x_5,\ x_1x_3 - x_5x_7,\ x_1x_7^2 - x_3^3,\\ &x_2^3 - x_4x_6^2,\ x_2^2x_8 - x_4^2x_6,\ x_2x_4 - x_6x_8,\ x_2x_8^2 - x_4^3,\\ &x_3^4 - x_5x_7^3,\ x_4^4 - x_6x_8^3\}\end{aligned}$$

4 Truncated Gröbner bases

The computation of the entire reduced Gröbner basis associated with the family of programs $IP_{A,C}$, is often expensive or infeasible. In practice, we are often interested in solving $IP_{A,C}(b)$ for a fixed right hand side vector b, which typically requires only a subset of the entire Gröbner basis. In [13], Thomas proposed a truncated Buchberger algorithm called b-Buchberger algorithm for toric ideals that finds a sufficient test set for $IP_{A,C}(b)$. This set is a proper subset of the reduced Gröbner basis of I_A, with respect to C. So, by the algorithm, we can produce a minimal test set for $IP_{A,C}$ whose right hand side vector is smaller than or equal to b in a specific sense, which greatly improves the computation.

Let $C_{\mathbb{N}}(A) = \{\sum_{i=1}^{n} m_i a_i : m_i \in \mathbb{N}\}$, where a_i is the ith column of the matrix A $(i = 1, ..., n)$. Then $C_{\mathbb{N}}(A)$ is a monoid and the $IP_{A,C}(b)$ is feasible if and only if b lies in $C_{\mathbb{N}}(A)$. We have the following lemma:

Lemma 4.1 The toric ideal $I_A = \bigoplus_{\beta \in C_{\mathbb{N}}(A)} I_A(\beta)$ where $I_A(\beta)$ is the vector space spanned by the binomials $\{x^u - x^v : Au = Av = \beta, u, v \in \mathbb{N}^n\}$.

Let M denote the set of all monomials in $k[x] = k[x_1, ..., x_n]$ where k is a field. The monoids M and $\mathbb{N}^n$ are isomorphic via the usual identification of a monomial x^u with its exponent vector. Under this identification, the monoid homomorphism π_A induces a multivariate grading of M and hence $k[x]$, where π_A-degree of x^u is denoted by $\pi_A(x^u) = \pi_A(u) = Au \in C_{\mathbb{N}}(A)$. Let M$(f)$ denote the monomials in a polynomial $f \in k[x]$.

Definition 4.1 A polynomial $0 \neq f \in k[x]$ is said to be π_A-homogeneous if $\pi_A(s) = \pi_A(t)$ for all monomials $s, t \in M(f)$. The π_A-degree of a homogeneous polynomial f, denoted $\pi_A(f)$, equals the π_A-degree of any monomial in M(f).

With above Lemma 4.1 and Definition 4.1, we have the following lemma:

Lemma 4.2 The toric ideal I_A is homogeneous with respect to the grading induced by π_A.

Associated with the monoid $C_{\mathbb{N}}(A)$ there is a "natural" partial order $\succeq$ such that for $b_1, b_2 \in C_{\mathbb{N}}(A)$, $b_1 \succeq b_2$ if and only if $b_1 - b_2 \in C_{\mathbb{N}}(A)$. Notice that when $C_{\mathbb{N}}(A) = \mathbb{N}^m$, the partial order $\succeq$ coincides with the componentwise partial order $\geq$.

Based on the above lemmas, we give b-Buchberger algorithm as follows. Let $\mathrm{NF}_{\{G,>_c\}}(g)$ denote the normal form of a binomial g, modulo a set of binomials G with term order $>_c$ and $S\text{-}bin(g_1, g_2)$ denote the S-binomial of two binomials g_1 and g_2.

Algorithm 4.1 b-Buchberger algorithm for toric ideals:

Input: A finite homogeneous binomial basis F of I_A and the term order $>_c$.

Output: A truncated (with respect to b) Gröbner basis of I_A.

```
$i = -1$
$G_0 = F$
Repeat
    $i = i + 1$
    $G_{i+1} = G_i \cup (\{\mathrm{NF}_{\{G_i,>_c\}}(S\text{-}bin(g_1, g_2)) : g_1, g_2 \in G_i, \pi_A(S\text{-}bin(g_1, g_2)) \preceq b\} \backslash \{0\})$
until $G_{i+1} = G_i$
Reduce $G_{i+1}$ modulo the leading monomials of its elements.
```

The algorithm 4.1 considers an S-binomial $g = x^u - x^v$ for reduction if and only if $\pi_A(g) = Au = Av \preceq b$. This amounts to checking feasibility if the system $\{x \in \mathbb{N}^n : Ax = b - Au\}$ which is as hard as solving the original IP problem $IP_{A,C}(b)$. Therefore, in order to implement the algorithm in practice, Thomas proposed two relaxations of the above check. Consider the S-binomial $g = x^u - x^v \in I_A$ for reduction if:

- $b - Au \in C(A)$ where $C(A) = \{Ax : x \in \mathbb{R}^n_+\}$, i.e., check feasibility of the linear programming relaxation of the original check.
- $b - Au \in C(A) \cap ZA$ where $ZA = \{Az : z \in \mathbb{Z}^n\}$. This is a relaxation of the original check since in general, $C_{\mathbb{N}}(A)$ is strictly contained in $C(A) \cap ZA$.

In our implementation, we use the second check by introducing the Hermite normal form, see the *Minimised Buchberger Geometric Algorithm* in section 5. When $C_{\mathbb{N}}(A) = \mathbb{N}^m$, we just use $b - Au \geq 0$, as the following example.

Example 4.1 (continue with Example 3.1)
Consider Example 3.1 in section 3. Suppose we are interested only in right hand side vector $b = (n_1, n_2, n_3, n_4)$ which satisfies $n_1 \leq 8$ and $n_2 \leq 8$. According to the condition, the truncated Gröbner basis equals $\{x_1^3 - x_3x_5^2,\ x_1x_3 - x_5x_7,\ x_2x_4 - x_6x_8\}$. Because here $C_{\mathbb{N}}(A) = \mathbb{N}^4$, the degree Au of these three binomials are (3,6,9,12), (4,6,4,6) and (6,4,6,4), which satisfy $b - Au \geq 0$, while for the other seven binomials, $b - Au < 0$. Thus, they lie outside of $C_{\mathbb{N}}(A)$.

5 The Minimised Geometric Buchberger Algorithm

Combining the GRIN method, the truncated Gröbner bases and geometric Buchberger algorithm together, we propose a new Buchberger algorithm called *Minimised Geometric Buchberger Algorithm* (MGBA) for IP problems. The idea behind the new algorithm is that for a special IP problem $IP_{A,C}(b)$ with fixed b, its minimal test set corresponds to a truncated reduced Gröbner basis of the toric ideal I_A. We encode $IP_{A,C}(b)$ into a subideal of I_A first, and then compute I_A using GRIN method, finally compute the truncated reduced Gröbner basis of I_A with b-Buchberger algorithm. We formulate the algorithm in the original space of $IP_{A,C}$ without introducing any additional variables and interpret all steps of the algorithm geometrically. This truncated reduced Gröbner basis, i.e., the minimal test set for $IP_{A,C}(b)$ that we obtained is a subset of the test set for $IP_{A,C}$. So, with the new algorithm, we can achieve considerable improvements in the efficiency and applicability of the Gröbner basis technique for IP.

Before giving the description of the algorithm, we need to introduce the Hermite normal form [7] from which we compute the lattice basis of $ker(A)$.

Definition 5.1 Let H be an $m \times m$ nonsingular integer matrix and $h_{ij} \in H$ for $i = 1, ..., m$ and $j = 1, ..., m$. H is said to be in *Hermite normal form* if:

(a) $h_{ij} = 0$ for $i < j$,

(b) $h_{ii} > 0$ for $i = 1, ..., m$, and

(c) $h_{ij} \leq 0$ and $|h_{ij}| < h_{ii}$ for $i > j$.

Definition 5.2 Let R be a nonsingular integer matrix. Then R is called unimodular if R has determinant ± 1.

Theorem 5.1 If A is an $m \times n$ integer matrix with rank$(A)=m$, then there exists an $n \times n$ unimodular matrix R such that:

(a) $AR = (H, 0)$ and H is in *Hermite normal form*, and

(b) $H^{-1}A$ is an integer matrix.

$(H, 0)$ is called the *Hermite normal form* of A. In [7], is shown a polynomial-time algorithm, called Hermite Normal Form Algorithm for finding R and H which serves as a constructive proof of Theorem 5.1. It also can be shown that H is unique. We have the following theorem in [7].

Theorem 5.2 Let $S = \{x \in \mathbb{Z}^n : Ax = b\}$ and let H and $R = (R_1, R_2)$ be as in Theorem 5.1, with R_1 an $n \times m$ matrix and R_2 an $n \times (n-m)$ matrix.

(a) $S \neq \emptyset$ if and only if $H^{-1}b \in \mathbb{Z}^m$.

(b) If $S \neq \emptyset$, every solution of S is of the form

$$x = R_1 H^{-1} b + R_2 z, \quad z \in \mathbb{Z}^{n-m}$$

From Theorem 5.2, we have a computation of a basis for $ker(A)$ as stated in the following theorem:

Theorem 5.3 Let B be a basis for $ker(A)$ and let H and $R = (R_1, R_2)$ be as in Theorem 5.2. Then $B = \{r_i : r_i$ is the ith column of R_2 and $i = 1, .., n-m\}$.

Proof: Suppose $x \in ker(A)$. Then $Ax = 0$. From Theorem 5.2, we have

$$x = R_1 H^{-1} 0 + R_2 w_2 = R_2 w_2$$

where w_2 is an arbitrary $(n-m)$-vector of integers.
Thus,

$$x = \sum_{i=1}^{n-m} \nu_i r_i$$

where r_i is the ith column of R_2 and $\nu_i \in \mathbb{Z}$, $i = 1, ..., n-m$.
Therefore, $B = \{r_i : r_i$ is the ith column of R_2 and $i = 1, .., n-m\}$. □

So, by the Hermite Normal Form Algorithm [7], we can compute H and $R = (R_1, R_2)$ for a matrix A, as well as a basis for $ker(A)$.

Now we are ready to describe our new algorithm MGBA. In this algorithm, the segment vector is slightly different from one in the geometric Buchberger algorithm. For a vector $d = [\alpha, \beta]$ in our algorithm, $\alpha, \beta \in \mathbb{N}^n$ and α is more expensive than β according to the term order $\prec_c$ defined as follow:

$$\beta \prec_c \alpha \iff \begin{cases} \beta C^T < \alpha C^T \\ \beta C^T = \alpha C^T \quad \text{and} \quad \beta \prec_o \alpha \end{cases}$$

where $\prec_c$ is encoded from the objective function Cx of $IP_{A,C}$ and $\prec_o$ is the lexicographic order. The fundamental segments in MGBA are constructed by interpreting the binomials of toric ideal $I_A = \langle x^{u^+} - x^{u^-} : u \in ker(A) \rangle$ geometrically.

Algorithm 5.1 Minimised Buchberger Geometric Algorithm

1. Compute lattice basis B for ker(A)

(1.1) Use the Hermite normal form algorithm in [7] to compute H and $R = (R_1, R_2)$.

(1.2) Compute the basis B:
$B = \{r_i : r_i \text{ is the } i\text{th column of } R_2 \text{ and } i = 1, .., n - m\}$

2. Reduce B into reduced lattice basis B_{red}

Here we use *Reduced Basis Algorithm* in [7] to compute the reduced lattice basis B_{red}. The purpose of this step is to make smaller some big numbers in B so that we can speed up the following computation by using the simplified (reduced) basis.

3. Compute toric ideal I_A

(3.1) Compute a subideal of I_A based on B_{red}

$$J_0 := \langle x^{u^+} - x^{u^-} : u \in B_{red} \rangle$$

Then interpret each binomial in J_0 as a vector by reading off its exponents. Here we can directly translate an element of B_{red} into a vector of J_0. For example, let $u \in B_{red}$ and $u = (1, 2, -1, -2)$, then the translated vector $v = [(1, 2, 0, 0), (0, 0, 1, 2)]$.

(3.2) For $i = 1, 2, ..., n$: Compute $J_i := (J_{i-1} : x_i^{\infty})$ geometrically by making x_i the reverse lexicographically cheapest variable. Here each J_i is in geometric term, i.e. its elements are all vectors. So, we use *Geometric Buchberger Algorithm* (Algorithm 2.1) to compute the reduced Gröbner basis for each J_i, $i = 1, ..., n$.

4. **Compute the truncated Gröbner basis $\mathcal{G}_{\prec_c}(b)$ of I_A with order $\prec_c$**

Input: generating set J_n of toric ideal I_A and term order $\prec_c$
Output: truncated reduced Gröber basis $\mathcal{G}_{\prec_c}(b)$

(4.1) Construct a Gröbner basis
In the first step of Algorithm 2.1 (*Geometric Buchberger algorithm*) we add a related check of the truncated Gröbner basis into the computation of S-vector :

$$b - Au \in C(A) \cap ZA \text{ } where \text{ } ZA = \{Az : z \in \mathbb{Z}^n\}$$

we check whether there exists a feasible solution for $S = \{x \in \mathbb{Z}^n : Ax = b - Au\}$ by (a) of Theorem 5.2 stated as follow:

$$\text{S is not empty if and only if } H^{-1}(b - Au) \in \mathbb{Z}^m$$

(4.2) Construct a minimal Gröbner basis
This step is the same as the second step of Algorithm 2.1.

(4.3) Construct the reduced Gröbner basis
This step is the same as the third step of Algorithm 2.1.

We illustrate the whole procedure of the above algorithm by the following example.

Example 5.1 We consider the following IP problem $IP_{A,C}(b)$:

minimise $x_1+8x_2 + 8x_3 + 16x_4 + 2x_5 + 2x_6 + 2x_7 + 2x_8$ subject to

$$\begin{aligned}
&x_1 + 2x_2 + 3x_3 + 4x_4 + x_6 + 4x_7 + 5x_8 = 7,\\
&2x_1 + 3x_2 + 4x_3 + x_4 + x_5 + 4x_6 + 5x_7 = 7,\\
&3x_1 + 4x_2 + x_3 + 2x_4 + 4x_5 + 5x_6 + x_8 = 13,\\
&5x_1 + 2x_2 + 3x_3 + 4x_4 + 6x_5 + x_6 + 2x_7 + 5x_8 = 17.
\end{aligned}$$

We have the coefficient matrix A, vector C and b as follows.

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 & 0 & 1 & 4 & 5 \\ 2 & 3 & 4 & 1 & 1 & 4 & 5 & 0 \\ 3 & 4 & 1 & 2 & 4 & 5 & 0 & 1 \\ 5 & 2 & 3 & 4 & 6 & 1 & 2 & 5 \end{pmatrix}$$

$$C = (1, 8, 8, 16, 2, 2, 2, 2) \quad b = (7, 7, 13, 17)$$

STEP 1.1: We use the Hermite normal form algorithm to compute H and R for A and obtain a basis B for $ker(A)$ as follows.

$$H = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ -11 & -12 & -2 & 22 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$R = \begin{pmatrix} -3 & -3 & -2 & 7 & -3 & 0 & -4 & 0 \\ 2 & 1 & 1 & -3 & 0 & -3 & 0 & -4 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 2 & 0 & 3 & 0 \\ 0 & 1 & 0 & -1 & 0 & 2 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Then,

$$R_2 = \begin{pmatrix} -3 & 0 & -4 & 0 \\ 0 & -3 & 0 & -4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 2 & 0 & 3 & 0 \\ 0 & 2 & 0 & 3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

From R_2 we can get the basis B for $ker(A)$:

$B = \{$(-3,0,1,0,2,0,0,0),(0,-3,0,1,0,2,0,0),(-4,0,0,0,3,0,1,0),(0,-4,0,0,0,3,0,1)$\}$

STEP 2: We compute the reduced basis B_{red} for $ker(A)$ and get:

B_{red} = {(-1,0,-1,0,1,0,1,0),(0,-1,0,-1,0,1,0,1),(-2,0,2,0,1,0,-1,0),(0,-2,0,2,0,1,0,-1)}

STEP 3.1: We interpret each element of B_{red} as a vector. For example, (-1,0,-1,0,1,0,1,0) is translated to the vector $[(0,0,0,0,1,0,1,0),(1,0,1,0,0,0,0,0)]$. So, we obtain a subideal of I_A as follow.

$$\begin{aligned}J_0 = \langle &[(0,0,0,0,1,0,1,0),(1,0,1,0,0,0,0,0)],[(0,0,0,0,0,1,0,1),(0,1,0,1,0,0,0,0)],\\ &[(0,0,2,0,1,0,0,0),(2,0,0,0,0,0,1,0)],[(0,0,0,2,0,1,0,0),(0,2,0,0,0,0,0,1)]\rangle\end{aligned}$$

STEP 3.2: We compute the toric ideal I_A. First we use Algorithm 2.1 to compute the reduced Gröbner basis for J_0 with respect to graded reverse lexicographic order that makes x_1 the cheapest variable. Here is $x_1 < x_2 < x_3 < x_4 < x_5 < x_6 < x_7 < x_8$. The result is

$$\begin{aligned}G_0 = \{&[(0,0,0,0,1,0,1,0),(1,0,1,0,0,0,0,0)],[(0,0,0,0,0,1,0,1),(0,1,0,1,0,0,0,0)],\\ &[(0,0,2,0,1,0,0,0),(2,0,0,0,0,0,1,0)],[(0,0,0,2,0,1,0,0),(0,2,0,0,0,0,0,1)],\\ &[(1,0,3,0,0,0,0,0),(2,0,0,0,0,0,2,0)],[(0,1,0,3,0,0,0,0),(0,2,0,0,0,0,0,2)]\}\end{aligned}$$

Next we divide each vector in G_0 by x_1 whenever possible, removing the common factor x_1 from two monomials. For example [(1,0,3,0,0,0,0,0), (2,0,0,0,0,0,2,0)] when divided by x_1 gives $[(0,0,3,0,0,0,0,0),(1,0,0,0,0,0,2,0)]$ and none of the other can be divided by x_1 (the cheapest variable in the above order).

Then we get a new set J_1 which consists of all the vectors in G_0 divided by x_1 whenever possible.

$$\begin{aligned}J_1 = \langle &[(0,0,0,0,1,0,1,0),(1,0,1,0,0,0,0,0)],[(0,0,0,0,0,1,0,1),(0,1,0,1,0,0,0,0)],\\ &[(0,0,2,0,1,0,0,0),(2,0,0,0,0,0,1,0)],[(0,0,0,2,0,1,0,0),(0,2,0,0,0,0,0,1)],\\ &[(0,0,3,0,0,0,0,0),(1,0,0,0,0,0,2,0)],[(0,1,0,3,0,0,0,0),(0,2,0,0,0,0,0,2)]\rangle\end{aligned}$$

Now we compute reduced Gröbner basis G_1 for J_1 by using the graded reverse lexicographic order that makes x_2 the cheapest variable. The order is $x_1 > x_8 > x_7 > x_6 > x_5 > x_4 > x_3 > x_2$. The result is

$$\begin{aligned}G_1 = \{&[(0,0,0,0,1,0,1,0),(1,0,1,0,0,0,0,0)],[(0,0,0,0,0,1,0,1),(0,1,0,1,0,0,0,0)],\\ &[(2,0,0,0,0,0,1,0),(0,0,2,0,1,0,0,0)],[(0,0,0,2,0,1,0,0),(0,2,0,0,0,0,0,1)],\\ &[(1,0,0,0,0,0,2,0),(0,0,3,0,0,0,0,0)],[(0,1,0,3,0,0,0,0),(0,2,0,0,0,0,0,2)],\\ &[(3,0,1,0,0,0,0,0),(0,0,2,0,2,0,0,0)]\}\end{aligned}$$

Dividing each binomial in G_1 by x_2 whenever possible, we get J_2:

$$\begin{aligned}J_2 = \langle &[(0,0,0,0,1,0,1,0),(1,0,1,0,0,0,0,0)],[(0,0,0,0,0,1,0,1),(0,1,0,1,0,0,0,0)],\\ &[(2,0,0,0,0,0,1,0),(0,0,2,0,1,0,0,0)],[(0,0,0,2,0,1,0,0),(0,2,0,0,0,0,0,1)],\\ &[(1,0,0,0,0,0,2,0),(0,0,3,0,0,0,0,0)],[(0,0,0,3,0,0,0,0),(0,1,0,0,0,0,0,2)],\\ &[(3,0,1,0,0,0,0,0),(0,0,2,0,2,0,0,0)]\rangle\end{aligned}$$

Then we can repeat the process, each time computing reduced Gröbner basis G_i for J_i by using graded reverse lexicographic order that makes x_{i+1} the cheapest variable and then dividing each vector in G_i by x_{i+1} to get J_{i+1}. Finally we get J_8, that is the the toric ideal I_A.

$$\begin{aligned} J_8 = \langle &[(0,0,0,0,1,0,1,0),(1,0,1,0,0,0,0,0)],\ [(0,1,0,1,0,0,0,0),(0,0,0,0,0,1,0,1)],\\ &[(0,0,2,0,1,0,0,0),(2,0,0,0,0,0,1,0)],\ [(0,0,0,2,0,1,0,0),(0,2,0,0,0,0,0,1)],\\ &[(0,0,3,0,0,0,0,0),(1,0,0,0,0,0,2,0)],\ [(0,0,0,3,0,0,0,0),(0,1,0,0,0,0,0,2)],\\ &[(0,0,1,0,2,0,0,0),(3,0,0,0,0,0,0,0)],\ [(0,0,0,1,0,2,0,0),(0,3,0,0,0,0,0,0)],\\ &[(0,4,0,0,0,0,0,0),(0,0,0,0,0,3,0,1)]\rangle \end{aligned}$$

STEP 4: In this step, we can use J_8 as the fundamental segments to compute truncated reduced Gröbner basis of I_A with fixed right hand side b and cost function Cx. The test set for $IP_{A,C}(b)$, i.e., the truncated reduced Gröbner basis is:

$$\begin{aligned} \mathcal{G}_{>_c}(b) = \{&[(1,0,1,0,0,0,0,0),(0,0,0,0,1,0,1,0)],\ [(0,1,0,1,0,0,0,0),(0,0,0,0,0,1,0,1)],\\ &[(0,0,2,0,1,0,0,0),(2,0,0,0,0,0,1,0)],\ [(0,0,0,2,0,1,0,0),(0,2,0,0,0,0,0,1)],\\ &[(0,0,3,0,0,0,0,0),(1,0,0,0,0,0,2,0)],\ [(0,0,0,3,0,0,0,0),(0,1,0,0,0,0,0,2)],\\ &[(0,0,1,0,2,0,0,0),(3,0,0,0,0,0,0,0)],\ [(0,3,0,0,0,0,0,0),(0,0,0,1,0,2,0,0)]\} \end{aligned}$$

For any feasible solution of the problem $IP_{A,C}(b)$, we can derive an optimal solution by using the above test set to reduce this feasible solution. For example, we have a feasible solution:

$$x_1 = 1,\ x_2 = 1,\ x_3 = 0,\ x_4 = 1,\ x_5 = 1,\ x_6 = 0,\ x_7 = 0,\ x_8 = 0$$

Then we can get an optimal solution:

$$x_1 = 1,\ x_2 = 0,\ x_3 = 0,\ x_4 = 0,\ x_5 = 1,\ x_6 = 1,\ x_7 = 0,\ x_8 = 1$$

Theorem 5.4 The algorithm MGBA terminates after a finite number of steps and its output is the unique minimal test set for $IP_{A,C}(b)$.

Proof: Finiteness of the algorithm is clear since the Hermite normal form algorithm, the Buchberger algorithm and b-Buchberger algorithm all terminate in finitely many steps.

By Proposition 3.1, we obtain the toric ideal I_A in step 3. Because the generating set of the toric ideal I_A is a set of fundamental segments for $IP_{A,C}$, the geometric Buchberger algorithm and b-Buchberger algorithm guarantee that we can obtain the truncated reduced Gröbner basis $\mathcal{G}_{\prec_c}(b)$ for $IP_{A,C}(b)$ with the term order $\prec_c$ in step 4. Now, we study $\mathcal{G}_{\prec_c}(b)$ from a completely geometric point of view. With $\mathcal{G}_{\prec_c}(b)$, we can build a connected, directed graph for only one fiber (b-fiber) of $IP_{A,C}(b)$. The nodes of the graph are all the lattice points in the fiber and the edges are the translations of elements in $\mathcal{G}_{\prec_c}(b)$ by nonnegative integral vectors. By Theorem 2.1.8. in [12], the graph has a unique sink at the unique optimum in this fiber. In this graph, there exists a directed path from every nonoptimal point to the unique optimum. So, the reduced Gröbner basis $\mathcal{G}_{\prec_c}(b)$ is a test set for $IP_{A,C}(b)$. By Corollary 2.1.10. in [12], we can prove $\mathcal{G}_{\prec_c}(b)$ is the unique minimal test set for $IP_{A,C}(b)$, depending on A, $\prec_c$ and b. □

6 Implementation, Experiment and Comparison

We have implemented the Minimised Geometric Buchberger Algorithm in the language C and developed a solver, called MGBS (Minimised Geometric Buchberger Solver) for IP on a Sun Ultra Enterprise 3000. MGBS works in two stages: the first stage is to compute a test set (reduced Gröbner basis) $\mathcal{G}_{\prec_c}(b)$ for $IP_{A,C}(b)$ based on MGBA, the second is to find a feasible solution and derive the optimal solution for $IP_{A,C}(b)$ by using $\mathcal{G}_{\prec_c}(b)$ to reduce the feasible solution. MGBS is connected via MathLink and CGI to the modelling IP system TIP [10]. When MGBS is called in Web page with an IP model (an objective function and a set of constraints), MGBS will solve the IP model and send the result (an optimal solution or a message "no feasible solution") to the Web page via CGI.

The main difficulty with MGBS is the computation of Gröbner bases. MGBS uses conventional Gröbner basis techniques to speed up this computation whenever this is suitable. For example, we make effective use of criteria to cut down the number of S-pairs, which is a bottleneck during the computations. However, the criteria which proved to be inefficient for the binomial case, such as Gebauer's B-criterion [11], are "switched off". It is a common strategy to keep the set of binomials throughout the entire Gröbner basis computation as reduced as possible. We implemented this idea by doing global reductions periodically (whenever new elements of the size of a fixed percentage of the current basis are created) as opposed to doing it every time a new binomial is added. Another important strategy is the extraction of common monomial factors in every newly created S-pair. This extraction is justified by the fact that the toric ideal I_A is a prime ideal not containing any common monomials. The above idea proved to be very effective, leading to reductions as much as 40-50% in execution time.

We have implemented the Geometric Buchberger Algorithm (GBA), the Truncated Geometric Buchberger Algorithm (TGBA) and the algorithm in GRIN, simply called GRIN by language C respectively on the Sun Ultra Enterprise 3000. We have carried out experiments by running MGBS on randomly generated matrices A of various sizes (ranging from 3×7 to 8×16) with nonnegative entries in a range between 0 and 20. We generate random right hand sides b to compute truncated Gröbner basis $\mathcal{G}_{\prec_c}(b)$. For each test instance (A,C,b) three comparisons are made with GBA, TGBA and GRIN.

Table 1: Experimental Result

Problems	Entries	MGBA		GRIN		TGBA		GBA	
		Size	Time	Size	Time	Size	Time	Size	Time
A3x7.1	0-20	17	0.34	31	0.55	212	68.84	245	420.90
A3x7.2	0-20	20	2.29	69	4.64	236	78.68	345	1129.26
A3x7.3	0-20	26	3.88	72	5.42	237	79.88	723	4746.13
A4x8.1	0-20	21	4.30	95	40.29	823	8476.86	3390	35118.24
A4x8.2	0-20	25	17.72	103	116.40	847	9913.19	3831	39426.20
A4x8.3	0-20	33	92.37	124	217.30	949	12118.89	4814	42042.16
A5x10.1	0-4	55	65.72	85	70.57	1143	826.27	1766	11887.24
A5x10.2	0-4	57	65.89	92	73.18	1171	896.76	1890	11995.97
A5x10.3	0-4	65	66.42	102	73.38	1284	930.23	2014	12207.70
A6x12.1	0-3	100	112.29	181	256.54	1300	1569.63	2353	18600.83
A6x12.2	0-3	153	189.87	418	1348.34	3304	3671.52	5026	24189.12
A6x12.3	0-3	267	358.39	709	3119.38	7872	8593.21	9590	35870.38
A8x16.1	0-1	11	6.76	20	7.46	48	13.89	63	15.67
A8x16.2	0-1	18	210.78	26	215.72	136	267.36	702	6385.31
A8x16.3	0-1	19	212.60	30	215.77	167	313.45	928	8250.13

The result of the comparisons is summarised in Table 1. The first column of the table indicates IP problems with their coefficient matrix A. For example, A3x7.1 represents No.1 IP problem with 3×7 integer matrix A. The range of the entries used in the problems are given in the second column. The third and fourth columns give the size of the truncated reduced Gröbner basis and the execution time for computing it for each problem with MGBA.

The fifth and sixth columns give two kinds of data (size and execution time) for GRIN. The seventh and eighth columns are for TGBA. The last two columns are for GBA. The timings are in CPU seconds on the Sun Ultra Enterprise 3000.

From Table 1, we can see the reduced Gröbner basis in GBA is the biggest one among the all algorithms because of the introduction of additional variables. Also the execution time is longest. For TGBA, because the algorithm computes the reduced Gröbner basis by fixing b, we can see the size of the reduced Gröbner basis in TGBA is less than that in GBA. But it is greater than that in GRIN and MGBA, because TGBA still introduces the additional variables to compute the reduced Gröbner basis. The size of the reduced Gröbner basis generated by MGBA is much less than those in the other three algorithms and also the performance in MGBA is the best.

7 Conclusions

We have proposed a new algorithm called Minimised Geometric Buchberger Algorithm for integer programming. It combines the GRIN method and the truncated Gröbner bases method to compute a generating set of the Gröbner bases in the original space and then refine it into a minimal test set i.e. a truncated reduced Gröbner basis of $IP_{A,C}(b)$ with fixed right hand side. Our preliminary experiments indicate that the algorithm is much faster than others such as the geometric Buchberger algorithm, the truncated geometric Buchberger algorithm and the algorithm in GRIN.

At present, the application of our algorithm MGBA is confined to small and middle size of IP problems. The prototype MGBS is not competitive in computing speed to popular commercial software such as CPLEX. However, we clearly see a new direction of research in solving IP problem using the theory of Gröbner bases. We would like to emphasize the importance of the symbolic methods of applying Gröbner basis technique for solving IP problems. Especially MGBA becomes more attractive when we tackle stochastic IP problems in [16]. The property of Gröbner basis corresponding directly to the test set of the IP problem does seem particularly useful for solving the classes of stochastic problems where some or all variables are integer valued which the general numerical methods can not handle. In this context, computing the Gröbner basis is still the main difficulty. So, with our algorithm for the test set of IP, we can provide an efficient method for solving the stochastic IP problems in [16]. Future research will be parallelisation of our algorithm and the application of the algorithm to stochastic IP problems.

References

[1] A.Heck, *Introduction to Maple, a computer algebra system.* Springer-Verlag, 1993.

[2] A. Schrijver, *Theory of Linear and Integer Programming* . John Wiley & Sons Ltd, 1986.

[3] B. Buchberger, *Gröbner bases: an algorithm method in polynomial ideal theory.* in Multidimensional Systems Theory (N.K.Bose ed.), Reidel, Dordrecht, pp. 184-232, 1985.

[4] B. Sturmfels, *Gröbner Bases and Convex Polytopes.* American Mathematical Society, volume 8. 1996.

[5] D. Cox, J. Little and D. O'Shea, *Ideals, varieties and algorithms.* Springer, New York, 1992.

[6] Daniel R. Grayson and Michael E. Stillman, *Macaulay 2, a software system for research in algebraic geometry.* Available at http://www.math.uiuc.edu/Macaulay2.

[7] G. L. Nemhauser and L. A. Wolsey, *Integer and combinatorial optimization.* Wiley-Interscience Series in Discrete Mathematics and Optimization, New York, 1988.

[8] P. Conti and C.Traverso, *Buchberger algorithm and integer programming.* Proceedings AAECC-9, New Orleans, pp. 130–139, LNCS 539 Springer-Verlag.

[9] Q.Li, Y.K.Guo and T.Ida, *A parallel algebraic approach towards integer programming.* Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, Washington,D.C.,U.S.A. , pp. 59-64, 1997.

[10] Q.Li, Y.K.Guo and T.Ida, *Modelling integer programming with logic: Language and implementation.* IEICE Trans. Fundamentals of Electronics, Communications and Computer Sciences, to appear.

[11] R. Gebauer and H. M. Möller, *On an installation of Buchberger's algorithm.* Journal of Symbolic Computation 6:275–286, 1988.

[12] R. R. Thomas, *A geometric Buchberger algorithm for integer programming.* Mathematics of Operations Research 20:864–884, 1995.

[13] R. R. Thomas and R. Weismantel, *Truncated Gröbner bases for integer programming.* Applicable Algebra in Engineering, Communication and Computing 8:241–257, 1997.

[14] S. Hosten and B. Sturmfels, *Grin: An implementation of Gröbner bases for integer programming.* In Balas, E., Clausen, J., editors, Integer Programming and Combinatorial Optimization, pp. 207–276, LNCS 920 Springer-Verlag.

[15] W. W. Adams and P. Loustaunau, *An Introduction to Gröbner bases.* American Mathematical Society, volume 3, 1994.

[16] S. R. Tayur, R. R. Thomas and N. R. Natrj, *An algebraic geometry algorithm for scheduling in presence of setups and correlated demands.* Mathematical Programming 69:369-401, 1995.

Higher-order Lazy Narrowing Calculus: a Solver for Higher-order Equations

Tetsuo Ida[1], Mircea Marin[1], and Taro Suzuki[2]

[1] Institute of Information Sciences and Electronics
University of Tsukuba, Tsukuba 305-8573, Japan
{mmarin,ida}@score.is.tsukuba.ac.jp
[2] Department of Computer Software
University of Aizu, Aizu Wakamatsu 965-8580, Japan
taro@u-aizu.ac.jp

Abstract. This paper introduces a higher-order lazy narrowing calculus (HOLN for short) that solves higher-order equations over the domain of simply typed λ-terms. HOLN is an extension and refinement of Prehofer's higher-order narrowing calculus LN using the techniques developed in the refinement of a first-order lazy narrowing calculus LNC. HOLN is defined to deal with both unoriented and oriented equations. It keeps track of the variables which are to be bound to normalized answers. We discuss the operating principle of HOLN, its main properties, i.e. soundness and completeness, and its further refinements. The solving capability of HOLN is illustrated with an example of program calculation.

1 Introduction

Proving, solving and computing are the essence of mathematicians' activities [2]. Correspondingly, modern programmers' role can be thought of as automating proving, solving and computing by defining specifications called programs. Traditionally, computing is the main concern of many programmers, and relatively smaller emphasis has been placed on the other two aspects of our activities. As computer science has become matured and demand for clarity and rigor is ever increasing as information technologies penetrate into our daily life, more and more programmers become concerned with proving and solving.

In this paper, we are concerned with the solving aspect of programming and present a solver which is a computation model for a programming language built upon the notion of equational solving. Let us start with functional programming. In functional programming we are interested in specifying a set $\mathcal{R}$ of rewrite rules as a program and then compute the normal forms of a term t, if the normal form exists. Formally, the problem statement is to prove the following formula:

$$\exists s. t \rightarrow^*_{\mathcal{R}} s \text{ and } s \text{ is a normal form.}$$

Usually, $\mathcal{R}$ is assumed to be confluent, and hence s is unique.

Proving the above statement is easy since all we have to do is to rewrite the term t repeatedly by applying the rewrite rules in $\mathcal{R}$ until it no longer gets rewritten. So the main concern here is not how to prove the statement, but how to rewrite the term t efficiently to its normal form if the normal form exists. The problem is generalized as follows.
Let t and t' be terms that may contain multiple free occurrences of a variable X. Prove $\exists X.t \leftrightarrow^*_{\mathcal{R}} t'$ such that X is a normal form.

Proving an existentially quantified formula by presenting a value that instantiates X is called *solving*. In particular, when an equality is defined as the reflexive, transitive and symmetric closure of $\rightarrow_{\mathcal{R}}$ as above, we call this *equational solving* (with respect to $\mathcal{R}$). Solving an equation is significantly difficult since (i) rewriting is not uni-directional, and (ii) we have to find a value for X before we perform rewriting. Indeed, various specialized methods have been developed for solving equations defined over specific domains, e.g. Gaussian elimination for solving a system of linear equations defined over reals.

In this paper we are primarily interested in solving equations over purely syntactic domains consisting of terms of simply typed λ-calculus. It is a domain of choice when we are reasoning about programs. Therefore the main theme of our paper is to show a method for solving high-order equations.

In the first-order setting where the domain is the Herbrand universe, methods for solving equations called paramodulation and narrowing are known. Narrowing is an E-unification procedure (E for Equational theory), and hence it can be naturally specified as a set of inference rules extending the rule-based specification of the unification algorithm [7]. The inference rules are used to recursively transform an equation into (hopefully) simpler equations.

There are pioneering works on extending narrowing to the higher-order case. A first systematic study of higher-order narrowing appeared in Prehofer's thesis [10]. It presents a higher-order lazy narrowing calculus that can be implemented relatively easily. It also has been shown that higher-order lazy narrowing is highly nondeterministic. Whereas various refinements have been developed to reduce the search space for solutions of first-order narrowing, the situation is much more complicated and difficult in the higher-order case.

With these observations in mind, we will first present a higher-order lazy narrowing calculus to be called HOLN in a general setting in order to expose its essential ingredients, as well as to enable further systematic refinements.

The rest of this paper is organized as follows. In Sect. 2 we introduce our main notions and notations. In Sect. 3 we define our main calculus HOLN and outline its properties. In Sect. 4 we describe the refinements of HOLN towards more deterministic computation. In Sect. 6 we illustrate by an example the solving capabilities of HOLN. Finally, in Sect. 7 we draw some conclusions and outline directions of further research.

2 Preliminaries

We use a slightly modified framework of simply typed λ-terms proposed in [10]. The main ingredients of our framework are:

- the set of all types T generated by a fixed set of *base types* and the function space constructor $\rightarrow$.
- an algebra $\mathcal{T}(\mathcal{F}, \mathcal{V})$ of simply typed λ-terms generated from a set $\mathcal{F}$ of T-typed constants and a set $\mathcal{V}$ of T-typed variables. We denote terms by the letters s, t, l, r, u possibly with a subscript. Instead of $\lambda x_1 \ldots \lambda x_n.s$ we write $\lambda\overline{x_n}.s$, where the x_i are assumed to be distinct. Similarly, instead of $(\ldots(s\ t_1)\ldots)\ t_n$ we write $s(\overline{t_n})$. The subscript n will be omitted when irrelevant. The set of free variables in a term t is denoted by $vars(t)$.
- a fully extended pattern rewrite system (EPRS for short) $\mathcal{R}$, which is a finite set of pairs $l \rightarrow r$ such that
  - l and r are λ-terms of the same base type,
  - $vars(r) \subseteq vars(l)$,
  - l is of the form $f(\overline{l_n})$, where $f \in \mathcal{F}$ and $l_1, \ldots, l_n$ are *fully extended patterns*. A *pattern* is a term such that all its free variable occurrences have distinct bound variables as arguments. A *fully extended pattern* is a pattern such that all its free variable occurrences take as arguments all the variables that are λ-abstracted above its position.

  Given an EPRS $\mathcal{R}$, we regard $\mathcal{F}$ as a disjoint union $\mathcal{F}_d \uplus \mathcal{F}_c$, where $\mathcal{F}_d = \{f \in \mathcal{F} \mid \exists(f(\overline{l_n}) \rightarrow r) \in \mathcal{R}\}$ is the set of *defined symbols*, and $\mathcal{F}_c = \mathcal{F} \setminus \mathcal{F}_d$ is the set of *constructors*.
- *equations* $e, e_1, e_2, \ldots$, which are pairs of terms of the same type. We distinguish *oriented* equations denoted by $s \rhd t$ and *unoriented* equations denoted by $s \approx t$. A *equational goal* (*goal* for short) is a pair $E|_W$ where E is a sequence of equations $e_1, \ldots, e_n$, abbreviated $\overline{e_n}$, and W is a set of free variables. The elements of W are called the *solution variables* of the given goal.

Unoriented equations are the usual equations used in everyday mathematics, and oriented equations were introduced in our formulation of narrowing to mark equations generated in the process of solving equations. Although the latter can be confined intermediate, we rather give them a first-class status by allowing oriented equations in an initial goal. Having both oriented and unoriented equations as syntactically distinct objects of study gives us more freedom for writing equational programs and also facilitates the understanding of the solving process.

We regard a goal as a pair consisting of a sequence of equations which denotes the existential closure of their logical conjunction, and a set of variables that we want to have bound to $\mathcal{R}$-normalized solutions. The reasons for this notion of goal are (i) that we are only interested in computing $\mathcal{R}$-normalized solutions, and (ii) that it allows us to keep track of the free variables that have to be instantiated to $\mathcal{R}$-normalized terms, as we will see later.

We use the following naming conventions: X, Y, Z, H, possibly primed or with a subscript, denote free variables; x, y, z, possibly primed or with a subscript, denote bound variables; and v denotes a constant or a bound variable. A sequence of syntactic objects $ob_1, \ldots, ob_n$ where $n \geq 0$ is abbreviated $\overline{ob_n}$.

We identify any λ-term t with its so called *long $\beta\eta$-normal form* defined by:

$$t\updownarrow_\beta^\eta := (t\downarrow_\beta)\uparrow^\eta,$$

where $t\downarrow_\beta$ denotes the *β-normal form* of t, and $t\uparrow^\eta$ the *η-expanded normal form* of t. The transformation of t to $t\updownarrow_\beta^\eta$ is assumed to be implicit. With this convention, every λ-term t can be uniquely written as $\lambda\overline{x_n}.a(\overline{s_m})$ where a is either a constant, bound variable, or free variable. The symbol a is called the *head* of t and is denoted by head(t). A term t is *flex* if head$(t) \in vars(t)$, and *rigid* otherwise. To simplify the notation, we will often relax the convention mentioned above and represent terms by their η-normal form.

An EPRS $\mathcal{R}$ induces a rewrite relation $\rightarrow_\mathcal{R}$ as usual. In each step of rewriting we employ an $\overline{x}$-lifted rewrite rule instead of a rewrite rule [10]. From now on we assume that $\mathcal{R}$ is an EPRS.

The *size* $|t|$ of a term t is the number of symbols occurring in t, not counting λ-binders. In the case of an equation e, its size $|e|$ is the sum of the sizes of the terms of both sides. For a sequence $\overline{e_n}$ of equations, its size is $\Sigma_{i=1}^n|e_i|$.

A substitution is a mapping $\gamma : \mathcal{V} \rightarrow \mathcal{T}(\mathcal{F}, \mathcal{V})$ such that its domain $Dom(\gamma)$ is finite, where $Dom(\gamma)$ is the set $\{X \in \mathcal{V} \mid \gamma(X) \neq X\}$. When $Dom(\gamma) = \{X_1, \ldots, X_n\}$, we may write γ as $\{X_1 \mapsto \gamma(X_1), \ldots, X_n \mapsto \gamma(X_n)\}$. The *empty substitution* ϵ is the substitution with empty domain. The homomorphic extension of a substitution is defined as usual, and we abuse the notation and use γ for its homomorphic extension. We denote by $t\gamma$ the image of a term t via the homomorphic extension of a substitution γ.

A substitution γ is *$\mathcal{R}$-normalized* iff $\gamma(X)$ is an $\rightarrow_\mathcal{R}$-normal form for all $X \in Dom(\gamma)$. Given a finite set of free variables V, we define the *restriction of γ to V* by $\gamma\restriction_V(X) = \gamma(X)$ if $X \in V$, and $\gamma\restriction_V(X) = X$ if $X \in \mathcal{V} \setminus V$. We define the relation $\gamma_1 \leq \gamma_2$ $[V]$ as $\exists\theta, \forall X \in V \,.\, X\gamma_2 = X\gamma_1\theta$.

A substitution γ is a *solution of an equation* e, notation $\mathcal{R} \vdash e\gamma$, if there exists a rewrite derivation R of the form (i) $s\gamma \rightarrow_\mathcal{R}^* t\gamma$, if $e = s \triangleright t$, and (ii) $s\gamma \leftrightarrow_\mathcal{R}^* t\gamma$, if $e = s \approx t$. Such an R is called a *rewrite proof that γ is a solution of* e. γ is a *solution of a goal* $\overline{e_n}|_W$, notation $\gamma \in Sol_\mathcal{R}(\overline{e_n}|_W)$, if $\gamma\restriction_W$ is an $\mathcal{R}$-normalized substitution and $\mathcal{R} \vdash e_k\gamma$ for all $1 \leq k \leq n$. Given $\gamma \in Sol_\mathcal{R}(E|_W)$, a *rewrite proof of* $\gamma \in Sol_\mathcal{R}(E|_W)$ is a mapping ρ which maps every equation e of E to a rewrite proof that γ is a solution of e. We denote by $|R|$ the length of a rewrite derivation R.

3 The Calculus HOLN

Now we are ready to formulate our problem.

Narrowing problem. *Given an EPRS $\mathcal{R}$ and a goal $E|_W$, find a set $Ans_\mathcal{R}(E|_W) \subseteq Sol_\mathcal{R}(E|_W)$ such that for any solution γ of $E|_W$ there exists $\theta \in Ans_\mathcal{R}(E|_W)$ and $\theta \leq \gamma$ $[vars(E)]$.*

HOLN (Higher-Order Lazy Narrowing calculus) is designed to give an answer to this narrowing problem.

3.1 Inference rules of HOLN

HOLN consists of three groups of inference rules: *preunification rules, narrowing rules,* and *removal rules of flex equations.* The inference rules are relations of the form

$$(E_1, e, E_2) \upharpoonright_W \Rightarrow_{\alpha,\theta} (E_1\theta, E, E_2\theta) \upharpoonright_{W'}$$

where α is the label of the inference rule, e is the selected equation, θ is the substitution computed in this inference step, $W' = \bigcup_{X \in W} vars(X\theta)$, and E is a sequence of equations called the *descendants* of e. We adopt the following notational conventions: $s \trianglerighteq t$ stands for $s \approx t$ or $t \approx s$ or $s \rhd t$; and $s \approxeq t$ stands for either $s \approx t$, $t \approx s$, $s \rhd t$ or $t \rhd s$. We assume that the usage of the symbols $\trianglerighteq$ and $\approxeq$ in both sides of an inference rule preserves the orientation of equations. Whenever used, $H, H_1, H_2, \ldots$ are assumed to be fresh variables.

Preunification rules

[i] Imitation.
If $g \in \mathcal{F}$ then

$$(E_1, \lambda\overline{x}.X(\overline{s_m}) \approxeq \lambda\overline{x}.g(\overline{t_n}), E_2)\upharpoonright_W \Rightarrow_{[i],\theta} (E_1, \overline{\lambda\overline{x}.H_n(\overline{s_m}) \approxeq \lambda\overline{x}.t_n}, E_2)\theta\upharpoonright_{W'}$$

where $\theta = \{X \mapsto \lambda\overline{y_m}.g(\overline{H_n(\overline{y_m})})\}$.

[p] Projection.
If $\lambda\overline{x}.t$ is rigid then

$$(E_1, \lambda\overline{x}.X(\overline{s_m}) \approxeq \lambda\overline{x}.t, E_2)\upharpoonright_W \Rightarrow_{[p],\theta} (E_1, \lambda\overline{x}.X(\overline{s_m}) \approxeq \lambda\overline{x}.t, E_2)\theta\upharpoonright_{W'}$$

where $\theta = \{X \mapsto \lambda\overline{y_m}.y_i(\overline{H_n(\overline{y_m})})\}$

[d] Decomposition.

$$(E_1, \lambda\overline{x}.v(\overline{s_n}) \trianglerighteq \lambda\overline{x}.v(\overline{t_n}), E_2)\upharpoonright_W \Rightarrow_{[d],\epsilon} (E_1, \overline{\lambda\overline{x}.s_n \trianglerighteq \lambda\overline{x}.t_n}, E_2) \upharpoonright_{W'}$$

Lazy narrowing rules

[on] Outermost narrowing at nonvariable position.
If $f(\overline{l_n}) \to r$ is an $\overline{x}$-lifted rewrite rule of $\mathcal{R}$ then

$$(E_1, \lambda\overline{x}.f(\overline{s_n}) \trianglerighteq \lambda\overline{x}.t, E_2) \upharpoonright_W \Rightarrow_{[on],\epsilon} (E_1, \overline{\lambda\overline{x}.s_n \rhd \lambda\overline{x}.l_n}, \lambda\overline{x}.r \trianglerighteq \lambda\overline{x}.t, E_2) \upharpoonright_{W'}$$

[ov] Outermost narrowing at variable position.

If $f(\overline{l_n}) \to r$ is an $\overline{x}$-lifted rewrite rule of $\mathcal{R}$ and $\begin{cases} \lambda\overline{x}.X(\overline{s_m}) \text{ is not a pattern} \\ \text{or} \\ X \notin W \end{cases}$

then

$$\begin{aligned}(E_1, \lambda\overline{x}.X(\overline{s_m}) \trianglerighteq \lambda\overline{x}.t, E_2) \upharpoonright_W \Rightarrow_{[ov],\theta} (&E_1\theta, \overline{\lambda\overline{x}.H_n(\overline{s_m\theta}) \rhd \lambda\overline{x}.l_n}, \\ &\lambda\overline{x}.r \trianglerighteq \lambda\overline{x}.t\theta, E_2\theta) \upharpoonright_{W'}\end{aligned}$$

where $\theta = \{X \mapsto \lambda\overline{y_m}.f(\overline{H_n(\overline{y_m})})\}$.

Removal rules

A *flex equation* is an equation both sides of which are flex terms.

[t] Trivial equations.

$$(E_1, \lambda\overline{x}.X(\overline{s}) \approx \lambda\overline{x}.X(\overline{s}), E_2)\lfloor_W \Rightarrow_{[t],\epsilon} (E_1, E_2)\lfloor_W$$

[fs] Flex-same.
If $\lambda\overline{x}.X(\overline{y_n})$ and $\lambda\overline{x}.X(\overline{y'_n})$ are patterns, and $X \in W$ then

$$(E_1, \lambda\overline{x}.X(\overline{y_n}) \trianglerighteq \lambda\overline{x}.X(\overline{y'_n}), E_2)\lfloor_W \Rightarrow_{[fs],\theta} (E_1, E_2)\theta\lfloor_{W'}$$

where $\theta = \{X \mapsto \lambda\overline{y_n}.H(\overline{z})\}$ with $\{\overline{z}\} = \{y_i \mid y_i = y'_i, 1 \le i \le n\}$.

[fd] Flex-different.
If $\lambda\overline{x}.X(\overline{y})$ and $\lambda\overline{x}.Y(\overline{y'})$ are patterns, and $\begin{cases} X \in W \text{ and } Y \in W & \text{if } \trianglerighteq \text{ is } \approx \\ X \in W & \text{if } \trianglerighteq \text{ is } \rhd \end{cases}$
then

$$(E_1, \lambda\overline{x}.X(\overline{y}) \trianglerighteq \lambda\overline{x}.Y(\overline{y'}), E_2)\lfloor_W \Rightarrow_{[fd],\theta} (E_1, E_2)\theta\lfloor_{W'}$$

where $\theta = \{X \mapsto \lambda\overline{y}.H(\overline{z}), Y \mapsto \lambda\overline{y'}.H(\overline{z})\}$ with $\{\overline{z}\} = \{\overline{y}\} \cap \{\overline{y'}\}$.

3.2 Main Property

An HOLN-*refutation* is a sequence of HOLN-steps

$$E\lfloor_W = E_0\lfloor_{W_0} \Rightarrow_{\alpha_1,\theta_1} E_1\lfloor_{W_1} \Rightarrow_{\alpha_2,\theta_2} \cdots \Rightarrow_{\alpha_n,\theta_n} E_n\lfloor_{W_n}$$

such that there is no HOLN-step starting with the goal $E_n\lfloor_{W_n}$. We abbreviate this sequence by $E_0\lfloor_{W_0} \Rightarrow^*_\theta E_n\lfloor_{W_n}$ where $\theta = \theta_1 \ldots \theta_n$. The *set of partial answers* computed with HOLN for a goal $E\lfloor_W$ is

$$PreAns^{HOLN}_{\mathcal{R}}(E\lfloor_W) = \{\langle\theta, E'\lfloor_{W'}\rangle \mid \exists \text{ HOLN-refutation } E\lfloor_W \Rightarrow^*_\theta E'\lfloor_{W'}\},$$

and the *set of answers* of HOLN is

$$Ans^{HOLN}_{\mathcal{R}}(E\lfloor_W) = \{\theta\gamma'\lfloor_{vars(E)} \mid \langle\theta, E'\lfloor_{W'}\rangle \in PreAns^{HOLN}_{\mathcal{R}}(E\lfloor_W) \text{ and } \gamma' \in Sol_{\mathcal{R}}(E'\lfloor_{W'})\}$$

HOLN is designed not to solve all equations: most of the flex equations are not transformed by HOLN-refutations. The reasons for not solving all flex equations are (i) that a flex equation always has a solution, and (ii) that in general there is no minimal complete set of unifiers for a flex equation. Therefore, the result of an HOLN-refutation is a pair $\langle\theta, E'\lfloor_{W'}\rangle$ where E' is a sequence of unsolvable flex equations. This design decision is similar to the one which underlies Huèt's higher-order preunification procedure, where flex equations are kept unsolved [3].

HOLN enjoys the following properties:

soundness: $Ans^{HOLN}_{\mathcal{R}}(E\lfloor_W) \subseteq Sol_{\mathcal{R}}(E\lfloor_W)$

completeness: for any $\gamma \in Sol_{\mathcal{R}}(E\lfloor_W)$ there exists $\theta \in Ans_{\mathcal{R}}^{HOLN}(E\lfloor_W)$ such that $\theta \leq \gamma$ $[vars(E)]$

Soundness follows easily from an inductive proof by case distinction on the inference rule used in the first step of the HOLN-refutation.

The main ideas of our completeness proof of HOLN are:

(a) We define the set $\mathtt{Cfg}$ of tuples of the form $\langle E\lfloor_W, \gamma, \rho\rangle$ with ρ a rewrite proof of $\gamma \in Sol_{\mathcal{R}}(E\lfloor_W)$. Such tuples are called *configurations.*

(b) We identify a well founded ordering $\succ \subseteq \mathtt{Cfg} \times \mathtt{Cfg}$ such that whenever $\langle E\lfloor_W, \gamma, \rho\rangle \in \mathtt{Cfg}$ and $e \in E$ can be selected in an HOLN-step, then there exists a pair $\langle \pi, \langle E'\lfloor_{W'}, \gamma', \rho'\rangle\rangle$ with π an HOLN-step of the form $E\lfloor_W \Rightarrow_\theta E'\lfloor_{W'}$, $\langle E\lfloor_W, \gamma, \rho\rangle \succ \langle E'\lfloor_{W'}, \gamma', \rho'\rangle$, and $\gamma = \theta\gamma'$ $[vars(E)]$.

We can define such $\succeq$ for HOLN as the lexicographic combination of the orderings $\succeq_A, \succeq_B, \succeq_C$, where:

- $\langle E\lfloor_W, \gamma, \rho\rangle \succeq_A \langle E'\lfloor_{W'}, \gamma', \rho'\rangle$ iff $\Sigma_{e\in E}|\rho(e\gamma)| \geq \Sigma_{e'\in E'}|\rho'(e'\gamma')|$,
- $\langle E\lfloor_W, \gamma, \rho\rangle \succeq_B \langle E'\lfloor_{W'}, \gamma', \rho'\rangle$ iff $\{|X\gamma| \mid X \in Dom(\gamma)\} \geq_{mul} \{|X'\gamma'| \mid X' \in Dom(\gamma')\}$,
- $\langle E\lfloor_W, \gamma, \rho\rangle \succeq_C \langle E'\lfloor_{W'}, \gamma', \rho'\rangle$ iff $|E\gamma| \geq_{mul} |E'\gamma'|$

The restriction $\succ$ of $\succeq$ is obviously well-founded, and its existence implies the completeness of HOLN.

HOLN can be regarded as an extension of the first-order lazy narrowing calculus LNC [8, 9] to higher-order one in the framework of EPRSs. This framework was first used by Prehofer [10] in the design of his higher-order lazy narrowing calculus LN for solving goals consisting of directed equations. HOLN can also be viewed as an extension of LN using the techniques developed in the refinements of LNC.

There are three sources of nondeterminism in computations with HOLN-derivations: the choice of the equation in the current goal, the choice of the inference rule of HOLN, and the choice of the rewrite rule of $\mathcal{R}$ when narrowing steps are performed. The completeness proof outlined above reveals a stronger result: HOLN is strongly complete, i.e., completeness is independent of the choice of the equation in the current goal.

In the sequel we will investigate the possibility to reduce the nondeterminism of computations with HOLN-derivations by reducing the choices of inference rules applicable to a given goal.

4 Refinements of HOLN

The main source of nondeterminism with HOLN-derivations is due to the many choices of solving an equation between a rigid term and a flex term. We call such an equation a *flex/rigid equation.* For example, to solve an equation of the form $\lambda\overline{x}.X(\overline{s_n}) \rhd \lambda\overline{x}.t$ where $\lambda\overline{x}.t$ is a rigid term, we have to consider all possible applications of rules [ov], [p], [i] (if head$(t) \notin \{\overline{x}\}$) and [on] (if head$(t) \in \mathcal{F}_d$.).

Also, the application of rule [ov] is a source of high nondeterminism, as long as we have large freedom to choose the defined symbols whose inference rules are employed in performing the [ov]-step.

In the sequel we describe two refinements of HOLN towards more deterministic versions, by restricting the narrowing problem to particular classes of EPRSs.

4.1 HOLN$_1$: Refinement for Left-Linear EPRSs

The restriction of programs to left-linear TRSs is widely accepted in the declarative programming community. It is well known that for left-linear confluent TRSs the standardization theorem holds [11]. This result allows to avoid the application of the outermost narrowing at nonvariable position to certain parameter-passing descendants in the case of the first-order lazy narrowing calculus LNC, without losing completeness [9]. As a consequence, the search space of LNC is reduced when the given TRS is left-linear and confluent.

In this subsection we show that a similar refinement is possible for confluent LEPRSs. This result is based on the fact that the standardization theorem holds for confluent LEPRSs as well. In the sequel we assume that $\mathcal{R}$ is an LEPRS.

To explain our result, we need some more definitions. A *parameter-passing equation* of a goal $E'|_{W'}$ in an HOLN-derivation $\Pi : E|_W \Rightarrow_\theta^* E'|_{W'}$ is either

(a) an equation $\lambda\overline{x}.s_k \rhd \lambda\overline{x}.l_k$ $(1 \leq k \leq n)$ if the last step of Π is of the form:

$$(E_1, \lambda\overline{x}.f(s_n) \trianglerighteq \lambda\overline{x}.t, E_2)|_W \Rightarrow_{[\text{on}],\epsilon} (E_1, \overline{\lambda\overline{x}.s_n \rhd \lambda\overline{x}.l_n}, \lambda\overline{x}.r \trianglerighteq \lambda\overline{x}.t, E_2)|_{W'}$$

(b) an equation $\lambda\overline{x}.H_k(\overline{s_m\theta}) \rhd \lambda\overline{x}.l_k$ $(1 \leq k \leq n)$ if the last step of Π is of the form:

$$(E_1, \lambda\overline{x}.X(\overline{s_m}) \trianglerighteq \lambda\overline{x}.t, E_2)|_W \Rightarrow_{[\text{ov}],\theta} (E_1\theta, \overline{\lambda\overline{x}.H_n(\overline{s_m\theta}) \rhd \lambda\overline{x}.l_n}, \lambda\overline{x}.r \trianglerighteq \lambda\overline{x}.t\theta, E_2\theta)|_{W'}.$$

A *parameter-passing descendant* of a goal $E'|_{W'}$ in an HOLN-derivation Π : $E|_W \Rightarrow_\theta^* E'|_{W'}$ is either a parameter-passing equation or a descendant of a parameter-passing equation. Note that parameter-passing descendants are always oriented equations. To distinguish them from the other oriented equations, we will write $s \blacktriangleright t$ instead of $s \rhd t$.

Positions in λ-terms are sequences of natural numbers which define the *path* to a subterm of a λ-term. We denote by ϵ the empty sequence, by $i{\cdot}p$ the sequence p appended to an element i, and by $p + p'$ the concatenation of sequences p and p'. A position p is above a position p', notation $p < p'$, if there exists $q \neq \epsilon$ such that $p' = p + q$. The *subterm* of s at position p, written $s|_p$, is defined as

- $s|_\epsilon = s$, $\quad v(\overline{t_n})|_{i\cdot p} = t_i|_p$ if $1 \leq i \leq n$, $\quad (\lambda\overline{x_m}.t)|_{1\cdot p} = (\lambda x_2 \ldots x_m.t)|_p$,
- undefined otherwise.

The set of positions of a term t is denoted by $Pos(t)$. p is a *pattern position* of a term t, notation $p \in Pat(t)$, if $p \in Pos(t)$ and $\mathrm{head}(t|_q) \notin vars(t)$ for all $q < p$.

A rewrite proof ρ of $\gamma \in Sol_{\mathcal{R}}(E|_W)$ is *outside-in* if the following conditions are satisfied for all equations e of E:

(a) $\rho(e\gamma)$ is an outside-in reduction derivation, that is, if $\rho(e\gamma)$ rewrites at positions $p_1, \dots, p_n$ with the $\overline{x}$-lifted rewrite rules $l_1 \to r_1, \dots, l_n \to r_n$ respectively, then the following condition is satisfied for all $1 \leq i \leq n-1$: if there exists j with $i < j$ such that $p_i = p_j + q$ then $q \in Pat(l_j)$ for the least such j.

(b) If $e = s \blacktriangleright t \in E$ and $\rho(e\gamma)$ has a rewrite step at position $1{\cdot}p$ such that no later rewrite steps take place above position $1{\cdot}p$ then $p \in Pat(t)$.

The following theorem can be proved using the the standardization theorem for confluent LEPRSs [12]:

Theorem 1 *Let $\mathcal{R}$ be a confluent LEPRS and $\gamma \in Sol_{\mathcal{R}}(E|_W)$. Then there exists an outside-in rewrite proof of $\gamma \in Sol_{\mathcal{R}}(E|_W)$.*

Theorem1 with its constructive proof states that for any rewrite proof γ we can construct an outside-in rewrite proof. Recall the proof sketch of completeness of HOLN. Even if we consider only configurations of the form $\langle E|_W, \gamma, \rho\rangle$ with ρ an outside-in rewrite proof of $\gamma \in Sol_{\mathcal{R}}(E|_W)$, the proof of strong completeness of HOLN remains valid when $\mathcal{R}$ is restricted to a confluent LEPRS. This implies that the HOLN-refutations considered in the proof do not contain [on]-steps applied to equations of the form

$$\lambda\overline{x}.f(\overline{s_n}) \blacktriangleright \lambda\overline{x}.X(\overline{y}) \quad \text{where } f \in \mathcal{F}_d. \tag{1}$$

Now we design HOLN_1 as follows.
HOLN_1 is the same as HOLN except that the inference rule [on] is not applied to the (selected) equation of form (1).

For HOLN_1, we have the following main result.

Main result: HOLN_1 is sound and strong complete for confluent LEPRSs [6].

4.2 HOLN$_2$: Refinement for Constructor LEPRSs

This refinement is inspired by a similar refinement of LNC with leftmost equation selection strategy for left-linear constructor TRSs [8]. It addresses the possibility to avoid the generation of parameter-passing descendants of the form $s \blacktriangleright t$ with $t \notin \mathcal{T}(\mathcal{F}_c, \mathcal{V})$. The effect of this behavior is that the nondeterminism between the inference rules [on] and [d] disappears for parameter-passing descendants.

In the first-order case, it is shown that LNC with leftmost equation selection strategy $\mathcal{S}_{\mathrm{left}}$ does not generate parameter-passing descendants $s \blacktriangleright t$ with $t \notin \mathcal{T}(\mathcal{F}_c, \mathcal{V})$. Unfortunately, this property is lost in the higher-order case mainly because the leftmost equation may be a flex equation to which no inference rule is applicable. Therefore, $\mathcal{S}_{\mathrm{left}}$ can not be adopted. To restore this property, we need to modify HOLN and to introduce a new equation selection strategy.

We define a new calculus HOLN$_2$ as the calculus consisting of all the inference rules of HOLN$_1$ and of the inference rule [c] defined as follows.

[c] Constructor propagation.
If $\exists s \blacktriangleright \lambda\overline{x_n}.X(\overline{y_n}) \in E_1$ and $s' = \lambda\overline{y_n}.s(\overline{x_n})$ then

$$(E_1, \lambda\overline{x}.X(\overline{t_n}) \trianglerighteq \lambda\overline{x}.u, E_2)\lfloor_W \Rightarrow_{[c],\epsilon} (E_1, \lambda\overline{x}.s'(\overline{t_n}) \trianglerighteq \lambda\overline{x}.u, E_2)\lfloor_{W'}. \qquad (2)$$

We give to [c] the highest priority.

Note that the application of [c] replaces the outermost occurrence of X in the selected equation by $\lambda\overline{y_n}.s(\overline{x_n})$.

We define a strategy $\mathcal{S}_c$ as follows. Let e be a selected equation of the form

$$\lambda\overline{x}.X(\overline{t_n}) \trianglerighteq \lambda\overline{x}.u$$

in a goal $(E_1, e, E_2)\lfloor_W$. An HOLN$_2$-step

$$(E_1, e, E_2)\lfloor_W \Rightarrow_{\alpha,\theta} (E_1, E, E_2)\theta\lfloor_{W'} \qquad (3)$$

respects strategy $\mathcal{S}_c$ if the inference step (3) is enabled only when all the parameter-passing descendants $s \blacktriangleright t$ in E_1 have t as a flex term.

We can easily prove the following lemma [6].

Lemma 1 *Let $\mathcal{R}$ be a confluent constructor LEPRS and Π be an* HOLN$_2$*-derivation that respects strategy $\mathcal{S}_c$. All the equations $s \blacktriangleright t$ in Π satisfy the property $t \in \mathcal{T}(\mathcal{F}_c, \mathcal{V})$.*

We have the following result for HOLN$_2$.

Main result: HOLN$_2$ with strategy $\mathcal{S}_c$ is sound and complete for confluent constructor LEPRSs.

Soundness follows from the fact that both rule [c] and the inference rules of HOLN are sound when strategy $\mathcal{S}_c$ is obeyed. The completeness proof works along the same lines as the completeness proof of HOLN$_1$, but the definition of the ordering between configurations is much more involved than the definition of $\succ$ [6].

5 Extensions of the Computational Model of HOLN

Many applications from the area of scientific computing require capabilities for solving constraints such as systems of linear equations, polynomial equations, or differential equations. Since HOLN can solve equations only over the domain of simply typed λ-terms, we investigated the possibility to extend HOLN to solve equations over some specific constraint domains equipped with well known solving methods. The results of our investigation are incorporated in our system CFLP (Constraint Functional Logic Programming system) [5]. CFLP is a distributed constraint solving system implemented in *Mathematica*, which extends the solving power of HOLN with methods to solve:

- systems of linear equations and equations with invertible functions,
- systems of multivariate polynomials, using Buchberger algorithm,
- systems of differential and partial differential equations.

The computational capabilities of CFLP go beyond the guarantee of our completeness results. This naturally points to the area of further research, namely the study of completeness of HOLN combined with complete external solvers.

6 Application

We will explain by an example how CFLP employs HOLN to compute solutions of problems formalized in higher-order equational programs.

Program Calculation. This example was briefly discussed in [5] to give a flavor to the capability of CFLP. Here we describe how HOLN works to compute efficient functional programs from less efficient but easily understandable ones. Such derivations are typical computations of higher-order equational programming.

We pose a question in a form of a goal that involves a higher-order variable. Then HOLN operates on the goal and transforms it successively into subgoals that are in turn solved. The computation is completed when HOLN finds no more subgoals to solve. HOLN delivers a substitution in which the higher-order variable is bound to the desired program.

Consider the problem of writing a program to check whether a list of numbers is steep. We say a list is *steep* if each element is greater than or equal to the average of the elements that follow it. By default, the empty list is steep.

With CFLP, such a test can be done by the function `steep` defined via the program `Prog` given below:

$$\begin{array}{ll}
\texttt{Prog} = & \{\texttt{steep[}\{\}\texttt{]}\to\texttt{True},\\
& \texttt{steep[[}\underline{a} \mid \underline{x}\texttt{]]}\to (a * \texttt{len}[x] \geq \texttt{sum}[x]) \wedge \texttt{steep}[x],\\
& \texttt{sum[}\{\}\texttt{]}\to 0,\ \texttt{sum[[}\underline{x} \mid \underline{y}\texttt{]]} \to x + \texttt{sum[}y\texttt{]},\\
& \texttt{len[}\{\}\texttt{]}\to 0,\ \texttt{len[[}\underline{x} \mid \underline{y}\texttt{]]} \to 1 + \texttt{len[}y\texttt{]},\\
& \texttt{tupling[}\underline{x}\texttt{]}\to\texttt{c3[sum[}x\texttt{],len[}x\texttt{],steep[}x\texttt{]]}\};
\end{array}$$

where

- the underlined symbols denote free variables,
- $\{\}$ denotes the empty list and $[H \mid T]$ denotes a list with head H and tail T,
- `c3` is a data constructor defined by

  ```
  TypeConstructor[Tuple = c3[Float,Float,Float]];
  ```

  This command defines the type constructor `Tuple` with associated data constructor `c3` of type `Float` $\times$ `Float` $\times$ `Float` $\to$ `Tuple`, and the corresponding data selectors sel·c3·1, sel·c3·2, sel·c3·3. CFLP assumes that, for a given constructor c_n, the following axioms hold for any $x, x_1, \ldots, x_n$ and $1 \leq k \leq n$:

$$c_n(\texttt{sel}\cdot c_n\cdot 1(x), \ldots, \texttt{sel}\cdot c_n\cdot n(x)) = x, \quad \texttt{sel}\cdot c_n\cdot k(c_n(x_1, \ldots, x_n)) = x_k.$$

CFLP implements partially these axioms via the following additional inference rule:

[fl] Flattening.
If $\lambda\overline{x}.t$ is rigid then

$$(E_1, \underline{\lambda\overline{x}.X(\overline{s_m}, c(\overline{t_n}), \overline{u_p})} \approx \lambda\overline{x}.t, E_2)|_W \Rightarrow_{[\mathrm{fl}],\theta}$$
$$(E_1\theta, \overline{\lambda\overline{x}.t_n\theta \rhd \lambda\overline{x}.H_n(\overline{x})}, \lambda\overline{x}.H(\overline{s_m\theta}, \overline{H_n(\overline{x})}, \overline{u_p\theta}) \approx \lambda\overline{x}.t\theta, E_2\theta)|_{W'}$$

where $\theta = \{X \mapsto \lambda\overline{x_m}, y, \overline{z_p}.H(\overline{x_m}, \overline{\texttt{sel}\cdot c_n\cdot n(y)}, \overline{z_p})\}$.

In addition, CFLP replaces automatically by t all the terms of the form

$$c_n(\texttt{sel}\cdot c_n\cdot 1(t), \ldots, \texttt{sel}\cdot c_n\cdot n(t)).$$

Prog is modular and easy to understand, but it is rather inefficient because the computation of steep for a given list has quadratic complexity. It is desirable to have a means to automatically compute the efficient version of the function steep defined above. Such a computation can be described via the so called fusion calculational rule shown below:

$$\frac{f(e) = e' \quad f(\texttt{foldr}(g, [n \mid ns])) = h(n, f(ns))}{f(\texttt{foldr}(g, e, ns)) = \texttt{foldr}(h, e', ns)} \tag{4}$$

where foldr is the usual fold function on lists.

In (4), the expression $f(\texttt{foldr}(g, e, ns))$ describes the inefficient computation, and $\texttt{foldr}(h, e', ns)$ is its efficient version. In our particular case, the inefficient computation of $\texttt{steep}([n \mid ns])$ is described by $\texttt{sel}\cdot\texttt{c3}\cdot 3(\texttt{tupling}([n \mid ns]))$. To find its efficient version, we employ rule (4) with $f = \texttt{tupling}$ and $g = \texttt{Cons}$ to the inefficient computation $\texttt{tupling}([n \mid ns])$ and compute an appropriate answer for the higher-order variable H to describe its efficient version $\texttt{H}(n, \texttt{tupling}(ns))$:

```
TSolve[
    λ[{n, ns}, tupling[[n | ns]] ≈ λ[{n, ns}, H[n, c3[sum[ns], len[ns], steep[ns]]]]],
    {h}, {},
    DefinedSymbol → {
     steep : TyList[Float] → Bool, sum : TyList[Float] → Float,
     len : TyList[Float] → Float, tupling : TyList[Float] → Tuple},
    EnableSelectors → True,
    Rules → Prog];

Type checking program ...
Type checking goal ...
{H → λ[{x$1865, x$1866},
```

$$\begin{aligned}&\texttt{c3}[x\$1865 + \texttt{sel}\cdot\texttt{c3}\cdot 1[x\$1866],\\ &\ 1 + \texttt{sel}\cdot\texttt{c3}\cdot 2[x\$1866],\\ &\ (x\$1865\ \texttt{sel}\cdot\texttt{c3}\cdot 2[x\$1866] \geq \texttt{sel}\cdot\texttt{c3}\cdot 1[x\$1866]) \wedge \texttt{sel}\cdot\texttt{c3}\cdot 3[x\$1866]]]\}\end{aligned}$$

The TSolve call of CFLP expects three arguments: the list of equations to be solved, the list of variables for which we want to compute normalized values, and the list of other variables. The computation performed during the execution of TSolve call can be controlled via the following options:

- Rules: specifies the LEPRS,
- DefinedSymbol: specifies the list of possibly typed-annotated defined symbols,
- EnableSelectors: specifies whether to enable or disable the usage of data selectors in the solving process.

In this case, the goal submitted to the underlying calculus of CFLP is

$$\lambda n, ns.\texttt{tupling}([n \mid ns]) \approx \lambda n, ns.\texttt{H}(n, \texttt{c3}(\texttt{sum}(ns), \texttt{len}(ns), \texttt{steep}(ns)))\lfloor_{\{\texttt{H}\}}.$$

To compute the binding for H, CFLP performs the following derivation:

$$\begin{aligned}
&\lambda n, ns.\texttt{tupling}([n \mid ns]) \approx \lambda n, ns.\texttt{H}(n, \texttt{c3}(\texttt{sum}(ns), \texttt{len}(ns), \texttt{steep}(ns)))\lfloor_{\{\texttt{H}\}}\\
&\Downarrow_{[\mathrm{on}]}\\
&(\lambda n, ns.[n \mid ns] \blacktriangleright \lambda n, ns.X(n, ns),\\
&\ \lambda n, ns.\texttt{c3}(\texttt{sum}(X(n, ns)), \texttt{len}(X(n, ns)), \texttt{steep}(X(n, ns))) \approx\\
&\quad \lambda n, ns.\texttt{H}(n, \texttt{c3}(\texttt{sum}(ns), \texttt{len}(ns), \texttt{steep}(ns))))\lfloor_{\{\texttt{H}\}}\\
&\Downarrow^{*}_{\{X \mapsto \lambda n, ns.[n|ns]\}}\\
&\lambda n, ns.\texttt{c3}(\texttt{sum}([n \mid ns]), \texttt{len}([n \mid ns]), \texttt{steep}([n \mid ns])) \approx\\
&\quad \lambda n, ns.\texttt{H}(n, \texttt{c3}(\texttt{sum}(ns), \texttt{len}(ns), \texttt{steep}(ns)))\lfloor_{\{\texttt{H}\}}\\
&\Downarrow_{[\mathrm{ff}], \{\texttt{H} \mapsto \lambda x, y.H_1(x, \texttt{sel}\cdot\texttt{c3}\cdot 1(y), \texttt{sel}\cdot\texttt{c3}\cdot 2(y), \texttt{sel}\cdot\texttt{c3}\cdot 3(y))\}}\\
&(\lambda n, ns.\texttt{sum}(ns) \rhd \lambda n, ns.X_1(n, ns),\\
&\ \lambda n, ns.\texttt{len}(ns) \rhd \lambda n, ns.X_2(n, ns),\\
&\ \lambda n, ns.\texttt{steep}(ns) \rhd \lambda n, ns.X_3(n, ns),\\
&\lambda n, ns.\texttt{c3}(\texttt{sum}([n \mid ns]), \texttt{len}([n \mid ns]), \texttt{steep}([n \mid ns])) \approx\\
&\lambda n, ns.H_1(n, X_1(n, ns), X_2(n, ns), X_3(n, ns)))\lfloor_{\{H_1\}}\\
&\Downarrow^{*}_{\{X_1 \mapsto \lambda n, ns.\texttt{sum}(ns), X_2 \mapsto \lambda n, ns.\texttt{len}(ns), X_3 \mapsto \lambda n, ns.\texttt{steep}(ns)\}}\\
&\lambda n, ns.\texttt{c3}(\texttt{sum}([n \mid ns]), \texttt{len}([n \mid ns]), \texttt{steep}([n \mid ns])) \approx\\
&\quad \lambda n, ns.H_1(n, \texttt{sum}(ns), \texttt{len}(ns), \texttt{steep}(ns))\lfloor_{\{H_1\}}\\
&\Downarrow_{[\mathrm{i}], \{H_1 \mapsto \lambda\overline{x_4}.\texttt{c3}(H_2(\overline{x_4}), H_3(\overline{x_4}), H_4(\overline{x_4}))\}}\\
&(\lambda n, ns.\texttt{sum}([n \mid ns]) \approx \lambda n, ns.H_2(n, \texttt{sum}(ns), \texttt{len}(ns), \texttt{steep}(ns)),\\
&\ \lambda n, ns.\texttt{len}([n \mid ns]) \approx \lambda n, ns.H_3(n, \texttt{sum}(ns), \texttt{len}(ns), \texttt{steep}(ns)),\\
&\ \lambda n, ns.\texttt{steep}([n \mid ns]) \approx \lambda n, ns.H_4(n, \texttt{sum}(ns), \texttt{len}(ns), \texttt{steep}(ns)))\lfloor_{\{H_2, H_3, H_4\}}\\
&\Downarrow^{*}\\
&G = (\lambda n, ns.n + \texttt{sum}(ns) \approx \lambda n, ns.H_2(n, \texttt{sum}(ns), \texttt{len}(ns), \texttt{steep}(ns)),\\
&\qquad \lambda n, ns.1 + \texttt{len}(ns) \approx \lambda n, ns.H_3(n, \texttt{sum}(ns), \texttt{len}(ns), \texttt{steep}(ns)),\\
&\qquad \lambda n, ns.(n * \texttt{len}(ns) \geq \texttt{sum}(ns)) \wedge \texttt{steep}(ns))\\
&\qquad\quad \approx \lambda n, ns.H_4(n, \texttt{sum}(ns), \texttt{len}(ns), \texttt{steep}(ns)))\lfloor_{\{H_2, H_3, H_4\}}
\end{aligned}$$

Finally, CFLP solves the goal G produced by the derivation depicted above by employing the inference rules [i], [p], [d], [fs], and [fd] of HOLN to compute the unifier $\{H_2 \mapsto \lambda\overline{x_4}.x_1 + x_2, H_3 \mapsto \lambda\overline{x_4}.1 + x_3, H_4 \mapsto \lambda\overline{x_4}.((x_1 * x_3 \geq x_2) \wedge x_4)\}$ of the equational part of G.

In this way CFLP computes the answer

$$\begin{aligned}\{H \mapsto \lambda n, ns.\mathrm{c3}(\ & n + \mathtt{sel\cdot c3\cdot 1}(ns),\\ & 1 + \mathtt{sel\cdot c3\cdot 2}(ns),\\ & n * \mathtt{sel\cdot c3\cdot 2}(ns) \geq \mathtt{sel\cdot c3\cdot 1}(ns) \wedge \mathtt{sel\cdot c3\cdot 3}(ns))\}\end{aligned}$$

which corresponds to the *Mathematica* representation of the answer produced by CFLP.

7 Conclusions and Future Work

We have presented a new lazy narrowing calculus HOLN for EPRS designed to compute solutions which are normalized with respect to a given set of variables, and then have presented two refinements to reduce its nondeterminism. Those refinements result in two calculi which are sound and complete.

The results presented in this paper owe largely to a new formalism in which we treat a goal as a pair consisting of a sequence of equations and a set of variables for which we want to compute normalized answers. This formulation of narrowing has the following advantages:

- it clarifies problems and locates points for optimization during the refutation process of goals,
- it simplifies the soundness and completeness proofs of the calculi,
- it simplifies and systematizes the implementation of the lazy narrowing calculus as a computational model of a higher-order functional logic programming system.

All the calculi given in this paper have been implemented as part of our distributed constraint functional logic system CFLP[1, 4, 5].

An interesting direction of research is to extend HOLN to conditional EPRSs. A program specification using conditions is much more expressive because it allows the user to impose equational conditions under which rewrite steps are allowed. Such an extension is quite straightforward to design, but it introduces many complexities for proving completeness.

Acknowledgements. The work reported in this paper has been supported in part by the Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Scientific Research (B) 12480066, 2000-2002. Mircea Marin has been supported by JSPS postdoc fellowship 00096, 2000-2001.

References

1. `http://www.score.is.tsukuba.ac.jp/reports/cflp/system/`.
2. B. Buchberger. Proving, Solving, Computing. A Language Environment Based on Mathematica. Technical Report 97-20, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, June 1997.
3. G. Huèt. *Résolution d'équations dans les languages d'ordre 1,2,...ω*. PhD thesis, University Paris-7, 1976.
4. M. Marin, T. Ida, and W. Schreiner. CFLP: a Mathematica Implementation of a Distributed Constraint Solving System. In *Third International Mathematical Symposium (IMS'99)*, Hagenberg, Austria, August 23-25 1999.
5. M. Marin, T. Ida, and T. Suzuki. Cooperative Constraint Functional Logic Programming. In T. Katayama, T. Tamai, and N. Yonezaki, editors, *International Symposium on Principles of Software Evolution (ISPSE 2000)*, pages 223–230, November 1-2 2000.
6. M. Marin, T. Suzuki, and T. Ida. Refinements of lazy narrowing for left-linear fully extened pattern rewrite systems. Technical Report ISE-TR-01-180, Institute of Information Sciences and Electronics, University of Tsukuba, Japan, 2001. To appear.
7. A. Martelli and U. Montanari. An Efficient Unification Algorithm. In *ACM Transactions on Programming Languages and Systems*, volume 4, pages 258–282, 1982.
8. A. Middeldorp and S. Okui. A deterministic lazy narrowing calculus. *Journal of Symbolic Computation*, 25(6):733–757, 1998.
9. A. Middeldorp, S. Okui, and T. Ida. Lazy narrowing: Strong completeness and eager variable elimination. *Theoretical Computer Science*, 167(1,2):95–130, 1996.
10. C. Prehofer. *Solving Higher-Order Equations. From Logic to Programming*. Foundations of Computing. Birkhäuser Boston, 1998.
11. T. Suzuki. Standardization theorem revisited. In *Proceedings of the Fifth International Conference on Algebraic and Logic Programming*, volume 1139 of *LNCS*, pages 122–134, Aachen, Germany, 1996.
12. V. van Oostrom. Personal communication.

An Open Environment for Cooperative Equational Solving

Tetsuo Ida, Mircea Marin and Norio Kobayashi
Institute of Information Sciences and Electronics
1-1-1 Tennoudai, Tsukuba 305-8573, Japan

ida, mmarin, nori @score.is.tsukuba.ac.jp

ABSTRACT

We describe a system called CFLP which aims at the integration of the best features of functional logic programming (FLP), cooperative constraint solving (CCS), and distributed computing. FLP provides support for defining one's own abstractions over a constraint domain in an easy and comfortable way, whereas CCS is employed to solve systems of mixed constraints by iterating specialized constraint solving methods in accordance with a well defined strategy. The system is a distributed implementation of a cooperative constraint functional logic programming scheme that combines higher-order lazy narrowing with cooperative constraint solving. The model takes advantage of the existence of several constraint solving resources located in a distributed environment (e.g., a network of computers), which communicate asynchronously via message passing. To increase the openness of the system, we are redesigning CFLP based on CORBA. We discuss some design and implementation issues of the system.

1. INTRODUCTION

Many important problems from engineering and sciences can be reduced to solving systems of equations over various constraint domains. It is generally accepted that a general-purpose solver can not solve efficiently such problems. A promising alternative is to focus on the design of a cooperative constraint solver. The main idea is to integrate in a coherent way the capabilities of various specialized constraint solvers and produce a system capable of solving systems of equations that none of the individual solvers can handle alone. In this view, it is desirable to create an *open* system, whose expressive power (i.e., the language of constraints) and solving capabilities (i.e., the constraint methods solving used by the cooperation) can be improved by appropriate addition of new constraint solvers. The design of such a system is challenging, at least because of the following reasons:

1. The system should allow to define and interpret one's own abstractions (i.e., user-defined symbols). Higher-order functional logic programming is a very powerful mechanism for defining such abstractions, but the design of efficient operational models is a difficult task [7].
2. Termination is a fundamental property of a constraint solver which can be easily lost during solver integration. This problem becomes even more challenging when designing an open equational environment.

Our goal is to design and implement a system capable of solving problems of the following type:

Given (a) a domain $\mathcal{A}$ and a signature $\mathcal{F}_e$ of *external* operators defined over $\mathcal{A}$, together with a collection of associated constraint solvers,

(b) a set $\mathcal{F}_c$ of *data constructors*,

(c) a program $\mathcal{R}$ over $\mathcal{A} \cup \mathcal{T}$ that introduces a set $\mathcal{F}_d$ of *user-defined function symbols*, where $\mathcal{T}$ is a syntactic domain formed by the formation rules of programs and

(d) a sequence G of equations that may contain a set of variables,

find the solutions of G in the equational theory defined by $\mathcal{A}$ and $\mathcal{R}$.

$\mathcal{R}$ and G are defined by the user, and are respectively called *user program*, and the *goal* of the program. The program $\mathcal{R}$ is a conditional pattern rewrite system, a higher-order rewrite system defined over the simply-type λ terms modulo β and long η [9]. The *solution* of G is a substitution θ for the variables in G such that the formula $G\theta$ holds in the theory defined $\mathcal{R}$. [6]

2. SYSTEM ARCHITECTURE

We have designed and implemented a distributed software system called CFLP [6] whose computational model integrates lazy narrowing for conditional pattern rewrite systems with cooperative constraint solving (CCS). We employ lazy narrowing to solve systems of equations containing symbols defined by conditional pattern rewrite rules [7, 8], and CCS to support the integration of various external constraint solvers so as to produce a generic constraint solver.

The architecture of our system is shown in Fig. 1. The system consists of:

1. an *interpreter* which solves goals by a mechanism which combines lazy narrowing steps with steps that collect constraints into a constraint store and dispatch them to the cooperative constraint solving system, and

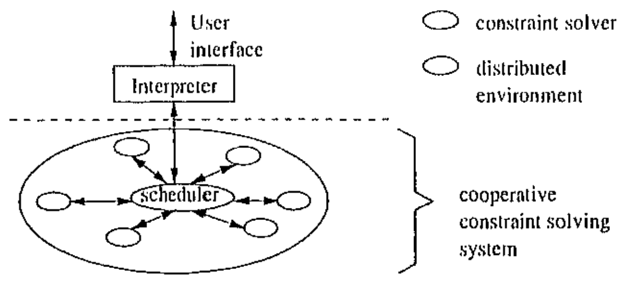


Figure 1: Architecture of CFLP

2. a distributed implementation of a cooperative constraint solver, consisting of
   (a) a *scheduler* which receives the constraints sent by the interpreter and dispatches them to specialized solvers in accordance with a given cooperation strategy, and
   (b) *specialized solvers*, which are resources located in a distributed environment and can be shared by many users.

Our choice for a distributed implementation of the cooperative constraint solver of CFLP was determined by the following observations. First, constraint solving resources are expensive; therefore they are best maintained at a developing server site and shared in a distributed environment. Secondly, our computing environment is becoming more and more net-centric; therefore instead of copying software at user side, we will in the future receive services of constraint solvers (i.e. obtain solutions) provided by constraint solvers via network.

3. MAIN FEATURES

Currently, CFLP has the following features:

- All system components are processes that communicate asynchronously over MathLink connections [10].
- The computation of the CCS is driven by a built-in cooperation strategy that has been proposed for cooperative constraint logic programming [5].
- The constraint solvers integrated in CFLP are built on top of the solving capabilities of Mathematica. We have defined and implemented solvers for
  - systems of linear equations,
  - equations with invertible functions,
  - systems of multivariate polynomial equations, and
  - differential equations.
- To improve the efficiency of the computation of the interpreter, we have integrated in CFLP all the deterministic refinements of lazy narrowing proposed by us so far [7, 8]. The user can choose the underlying calculus of the interpreter.
- The user can adjust the cooperative constraint solver of CFLP by specifying the number and location of the specialized solvers.

4. APPLICATIONS

We describe with two examples the solving capabilities of CFLP.

4.1 Electric Circuit Modeling

The first example shows how electric circuit layouts can be computed with CFLP. This example illustrates the expressive power of the FLP style programming extended with

- higher-order constructs such as λ-abstractions and function variables, and
- constraint solving capabilities for differential equations, linear equations, and systems of polynomial equations.

We first define a function spec which describes the behavior in time t of an electrical component as a function of the current $i[t]$ and voltage $v[t]$ in the circuit. The CFLP rules of spec correspond to a recursive definition, where the base case describes the behavior of elementary circuits such as resistors, capacitors, and inductors, and the inductive case describes the behavior of serial and parallel connections of electrical components.

The CFLP program is given below. We do not explain the underlying electronic laws since it should be easy to read them off from the program.

```
In[1]:= Prog = {
          spec[res[r], λ[{t}, v[t]], λ[{t}, i[t]]] →
            True ⇐ {λ[{t}, v[t]] ≈ λ[{t}, r i[t]]},
          spec[ind[l], λ[{t}, v[t]], λ[{t}, i[t]]] →
            True ⇐ λ[{t}, v[t]] ≈ λ[{t}, l i'[t]],
          spec[cap[c], λ[{t}, v[t]], λ[{t}, i[t]]] →
            True ⇐ λ[{t}, i[t]] ≈ λ[{t}, c v'[t]],
          spec[serial[{}], λ[{t}, 0], i] → True,
          spec[serial[[comp|T]], v, i] → True ⇐
            {spec[comp, v1, i] ≈ True,
              spec[serial[T], v2, i] ≈ True,
              λ[{t}, v[t]] ≈ λ[{t}, v1[t] + v2[t]]},
          spec[parallel[{}], v, λ[{t}, 0]] → True,
          spec[parallel[[comp|T]], v, i] → True ⇐
            {spec[comp, v, i1] ≈ True,
              spec[parallel[T], v, i2] ≈ True,
              λ[{t}, i[t]] ≈ λ[{t}, i1[t] + i2[t]]}};
```

The universally quantified variables are underlined. Note the usage of the list construct in the recursive specification of serial and parallel connections of electrical components. The CFLP system recognizes the following list specifications:

- $\{t_1, \ldots, t_n\}$: list consisting of components $t_1, \ldots, t_n$,
- $[\ h \mid tl\ ]$: CFLP list with head h and tail tl in Prolog-style notation.

Consider the problem of finding the behavior in time of the current in a RLC circuit with R = 2, L = 1 and C = 1/2 (see Fig. 2), under the restrictions that the voltage is 50V and the current was initially set to 0.

In CFLP this problem is reduced to solving the system of equations:

```
In[2]:= Problem =
          {spec[serial[{res[2], ind[1], cap[1/2]}],
          λ[{t}, 50], i] ≈ True,
            i[0] ≈ 0};
```

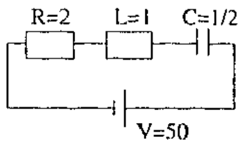


Figure 2: RLC circuit

The variable which we want to compute is *i*, and it is overlined. Note the usage of the expression λ[{ },] for specifying that the voltage is constant (50 V) in time.

To solve this problem, we call:

```
In[3]:= TSolve[Problem,
            Constructor → {V, res, ind, cap,
                           serial, parallel},
            Rules → Prog]
Add rule :∀{r,i,v}spec[res[r], v, i] → True ⇐
  λ[{t}, v[t]] ≈ λ[{t}, r i[t]]
Add rule :∀{l,i,v}spec[ind[l], v, i] → True ⇐
  λ[{t}, v[t]] ≈ λ[{t}, l i'[t]]
Add rule :∀{c,i,v}spec[cap[c], v, i] → True ⇐
  λ[{t}, i[t]] ≈ λ[{t}, c v'[t]]
Add rule :∀{i}spec[serial[{}], λ[{t}, 0], i] → True
Add rule :∀{comp,T,v1,v2,i,v}
    spec[serial[[comp|T]], v, i] → True ⇐
      spec[comp, v1, i] ≈ True,
      spec[serial[T], v2, i] ≈ True,
      λ[{t}, v[t]] ≈ λ[{t}, v1[t] + v2[t]]
Add rule :∀{v}spec[parallel[{}], v, λ[{t}, 0]] → True
Add rule :∀{comp,i,i1,i2,T,v}
    spec[parallel[[comp|T]], v, i] → True ⇐
      spec[comp, v, i1] ≈ True,
      spec[parallel[T], v, i2] ≈ True,
      λ[{t}, i[t]] ≈ λ[{t}, i1[t] + i2[t]]
{i → λ[{t}, -c1$2 e^-t Sin[t]]
Out[3]= {{i → λ[{t}, -c1$2 e^-t Sin[t]]}}
```

This call is similar to the `Solve` call of *Mathematica*, but `TSolve` has the specific options

- `Rules`: provides the functional logic program (i.e., conditional rewrite system) used during solving the goal,
- `Constructor`: specifies the data constructors used in the specification of the program and of the goal.

The system computes the parametric solution

$$\{i \to \lambda t.- \quad \$ \quad e^{-t}\sin(t)\}$$

which is represented in *Mathematica* by

$$\{i \to \lambda[\{t\}, - \quad \$ \quad e^{-t}\ \sin[t]]\}.$$

4.2 Program Calculation

This example illustrates the capabilities of a computing environment which can perform full higher-order pattern unification and higher-order term rewriting.

Consider the problem of writing a program to check whether a list of numbers is *steep*, i.e. if every element of the list is greater than or equal to the average of the elements that follow it. A CFLP program that does this is:

```
In[4]:= Prog = {
            steep[{}] → True,
            steep[[a|x]] → And[a * len[x] ≥ sum[x],
                                  steep[x]],
            sum[{}] → 0, sum[[x|y]] → x + sum[y],
            len[{}] → 0, len[[x|y]] → 1 + len[y]};
```

Prog is modular and easy to understand, but it is inefficient (quadratic complexity). It is desirable to compute the efficient version steepOpt of steep. For doing this, we employ the *fusion calculational rule*:

$$\frac{f[e] = e' \quad f[g[a, \quad]] = h[a, f[\quad]]}{f[\mathrm{foldr}[g, e, \quad]] = \mathrm{foldr}[h, e', \quad]}$$

where foldr is defined recursively by

$$\mathrm{foldr}[g, e, \{\}] = e,\ \mathrm{foldr}[g, e, [n|ns]] = g[n, \mathrm{foldr}[g, e, ns]].$$

The inefficient computation is f[foldr[g,e,ns]], and its efficient version is foldr[h,e',ns]. Finding the efficient version amounts to finding h such that f[g[a,ns]]=h[a,f[ns]].

To use the fusion calculational rule, we first formalize the the computation of steep as f[foldr[g,e,ns]]. It is easy to see that

$$\mathrm{steep}[n]=\mathrm{sel\text{-}c3\text{-}1}[n]= \mathrm{sel\text{-}c3\text{-}1}[f[\mathrm{foldr}[g, e, n]]],$$

where f[n]=c3[steep[n],sum[n],len[n]], g=Cons and e={ }. Here, sel-c3-1 denotes the first data selector associated of the data constructor c3. This implies that the following relation holds for any valid arguments *x*, *y*, *z*:

$$\mathrm{sel\text{-}c3\text{-}1}[\mathrm{c3}[x, y, z]] = x.$$

If we succeed to find h, then

$$\mathrm{steepOpt}[n] = \mathrm{sel\text{-}c3\text{-}1}[f[\mathrm{foldr}[h, e', n]]]$$

where e'=f[e]=c3[True,0,0].

To compute h, we solve the equation f[g[a,ns]]=h[a,f[ns]], i.e. the goal

```
In[5]:= Problem = λ[{n, ns}, f[[n|ns]]] ≈
            λ[{n, ns},
             h̄[n, c3[steep[ns], sum[ns], len[ns]]]];

In[6]:= Constructor[Triple = c3[Bool, Float, Float]];

In[7]:= TSolve[Problem,
          Constructor →
           {Plus, Power, Times, GreaterEqual, And},
          Rules → Prog∪
           {f[ns] → c3[steep[ns], sum[ns], len[ns]]}]
Add rule :∀{ns}f[ns] → c3[steep[ns], sum[ns], len[ns]]
Add rule :∀{x,y}len[[x|y]] → 1 + len[y]
Add rule :len[{}] → 0
Add rule :∀{a,x}steep[[a|x]] → alen[x] ≥ sum[x]&&steep[x]
Add rule :steep[{}] → True
Add rule :∀{x,y}sum[[x|y]] → x + sum[y]
Add rule :sum[{}] → 0
{h → λ[{x$23, x$24},
    c3[x$23 sel·c3·3[x$24] ≥ sel·c3·2[x$24]&&
        sel·c3·1[x$24], x$23 + sel·c3·2[x$24],
      1 + sel·c3·3[x$24]]]}
Out[7]=  {{h → λ[{x$23, x$24}, c3[
             x$23 sel·c3·3[x$24] ≥ sel·c3·2[x$24]&&
              sel·c3·1[x$24],
             x$23 + sel·c3·2[x$24],
             1 + sel·c3·3[x$24]]]}}
```

Note that CFLP handles the operators + (Plus), Power, * (Times), and ≥ (GreaterEqual) as external operators, that is, operators used in constraint specifications. As a consequence, the equations built with these operators are handled by default as constraints and sent to be solved by the constraint solver. In this example we override the default specification of these operators by declaring them as mere constructors in the `Constructor` option of the `TSolve` call. As a consequence, CFLP gives up using constraint solvers for solving equations built with the operators mentioned above, and enables the interpreter to solve them.

4.3 Other Applications

We have investigated several other problems from physics and found that they can be solved with CFLP by reducing them to systems of constraints that can be handled by a cooperation between the specialized solvers of CFLP.

5. CORBA-BASED OPEN ENVIRONMENT

We are currently working on redesigning the scheduler to accept different cooperation strategies provided by the user.

The work includes the replacement of the modules that rely on MathLink by more versatile protocols. We use CORBA [1] in the new implementation because it is the standardized software technology for distributed computing and it fits well with the objective of our open environment for equational solving. CORBA facilitates

(a) to get uniform access to various applications (such as specialized solvers) written in different languages, via object modeling, when they have a common interface specified in OMG IDL,

(b) to locate transparently solving resources in distributed environments, and

(c) to plug them in whenever their capabilities are needed to solve a problem.

As CORBA provides the framework for cooperative constraint solvers and low-level protocols for communication among objects, our task is to define higher-level protocols for communicating among distributed solvers.

5.1 MAXCOR

We have designed MAXCOR (MAth eXchange for CORBA), a framework for exchanging mathematical expressions for CORBA. MAXCOR contain three ingredients:

1. the language for mathematical formulas to be used in conjunction with OMG IDL,
2. the common interface definition in OMG IDL for all objects in MAXCOR,
3. CORBA object wrappers for binding MAXCOR and existing applications.

As for point 1., we adopted MathML [3], the subset of XML. In a simply-minded implementation, MathML documents could be passed as arguments of string type in OMG IDL. However, this would require translation of object-to-string and vice versa on both client and server sides. Instead for efficiency reasons, we employ XML DOM (Document Object Model) and DOM-to-valuetype mapping [2]. XML DOM is standardized by W3C and its implementation is available as an API. Furthermore, DOM-to-valuetype mapping is being specified by OMG and is a part of XML-to-valuetype mapping. DOM-to-valuetype mapping is entirely defined in OMG IDL. Thus our design is platform-independent.

As for point 2., we show below a simplified version of the MAXCOR module declaration. Here the server is a solver and the client is the scheduler that calls the solver.

```
#include"value_dom.idl"

module MAXCOR {
  struct threadID {
    sequence<octet> passwd;
      // password for this thread
    long number;
      // thread number identifying the thread
      // in which the solver runs
  };
  exception mException{
    string reason;
  };

  interface MAXCORObject {
    threadID init(in string parameter)
      raises(mException);
    void deinit(
      in threadID id, in string parameter)
      raises(mException);
    Document execute(
      in threadID id, in string command,
      in Document expr) raises(mException);
    Document serviceSpec(in string command)
      raises(mException);
    string usage(in string parameter);
  };
};
```

In the above program, `value_dom.idl` is the declaration file of XML-to-valuetype mapping and `Document` is the valuetype of DOM object. Type definition of `threadID` is self-explanatory. Operations `init`, `deinit` and `execute` do the real work. Operation `init` creates new threads for running the server, initializes the running environment and establishes the session with its client. Operation `deinit` dismantles the server process when the server completes the service. Operation `execute` invokes the program of the solver. The arguments of `execute` are a string of command name, and an expression which will be evaluated by the server.

Operations `serviceSpec` and `usage` are auxiliary and provide information to the client. Operation `usage` returns the help message related to the service of the solver. Operation `serviceSpec` in our present version returns specification of the argument of the solver.

5.2 Implementation

We have implemented the following APIs: DOM to valuetype mapping in C++ and Java, CORBA object wrappers for Mathematica in C++ and Java and for CPLEX [4] in C++. CPLEX is an application package for integer programming and its wrapped API is used as one of the solvers for CFLP. The solvers that the present version of CFLP are using are all written in Mathematica with the interface complied with MathLink. This interface will be replaced by CORBA object wrappers. We will use MAXCOR for communication between a scheduler and solvers in CFLP. Currently we are working for the full deployment of solvers wrapped as CORBA objects that cooperate with the scheduler of CFLP.

6. CONCLUSION

We introduced an open environment for equational solving. The core of the environment is CFLP system which supports equational reasoning over variety of domains. We showed main features of the system by giving examples that exploits the style of equational reasoning. We further discussed a new implementation of the open environment which is under way based on CORBA technology.

7. ACKNOWLEDGMENTS

This work is supported in part by the Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Scientific Research (B) 12480066, 2000 and Mircea Marin is supported by JSPS postdoc fellowship.

8. REFERENCES

[1] `http://www.corba.org.`

[2] `http://www.omg.org/techprocess/meetings/schedule/XML_Value_RFP.html.`

[3] `http://www.w3.org/Math/.`

[4] Using the CPLEX Callable Library. CPLEX Optimization, Inc., USA, 1995.

[5] H. Hong. CLP(CF): Constraint Logic Programming over Complex Functions. Technical Report 94-09, RISC-Linz, Castle of Hagenberg, Austria, 1992.

[6] M. Marin. *Functional Logic Programming with Distributed Constraint Solving*. PhD thesis, Institute RISC-Linz, Johannes Kepler University, Castle of Hagenberg, Austria, 2000.

[7] M. Marin, T. Ida, and T. Suzuki. On Reducing the Search Space of Higher-Order Lazy Narrowing. In A. Middeldorp and T. Sato, editors, *FLOPS'99*, volume 1722 of *LNCS*, pages 225–240. Springer-Verlag, 1999.

[8] M. Marin, T. Ida, and T. Suzuki. Lazy Narrowing Calculi in Perspective. In *Proceedings of the 9th International Workshop on Functional and Logic Programming*, pages 238–252, Benisassim, Spain, 2000.

[9] T. Nipkow. Functional unification of higher-order patterns. In *Proceedings of 8th IEEE Symposium on Logic in Computer Science*, pages 64–74, 1993.

[10] S. Wolfram. *The Mathematica Book*. Fourth Edition, Wolfram Media Inc. Champaign, Illinios, USA, and Cambridge University Press, 1999.

An Open Environment for Cooperative Equational Solving

Tetsuo Ida, Mircea Marin and Norio Kobayashi
Institute of Information Sciences and Electronics
1-1-1 Tennoudai, Tsukuba 305-8573, Japan
{ida, mmarin, nori}@score.is.tsukuba.ac.jp

ABSTRACT

We describe a system called CFLP which aims at the integration of the best features of functional logic programming (FLP), cooperative constraint solving (CCS), and distributed computing. FLP provides support for defining one's own abstractions over a constraint domain in an easy and comfortable way, whereas CCS is employed to solve systems of mixed constraints by iterating specialized constraint solving methods in accordance with a well defined strategy. The system is a distributed implementation of a cooperative constraint functional logic programming scheme that combines higher-order lazy narrowing with cooperative constraint solving. The model takes advantage of the existence of several constraint solving resources located in a distributed environment (e.g., a network of computers), which communicate asynchronously via message passing. To increase the openness of the system, we are redesigning CFLP based on CORBA. We discuss some design and implementation issues of the system.

1. INTRODUCTION

Many important problems from engineering and sciences can be reduced to solving systems of equations over various constraint domains. It is generally accepted that a general-purpose solver can not solve efficiently such problems. A promising alternative is to focus on the design of a cooperative constraint solver. The main idea is to integrate in a coherent way the capabilities of various specialized constraint solvers and produce a system capable of solving systems of equations that none of the individual solvers can handle alone. In this view, it is desirable to create an *open* system, whose expressive power (i.e., the language of constraints) and solving capabilities (i.e., the constraint methods solving used by the cooperation) can be improved by appropriate addition of new constraint solvers. The design of such a system is challenging, at least because of the following reasons:

1. The system should allow to define and interpret one's own abstractions (i.e., user-defined symbols). Higher-order functional logic programming is a very powerful mechanism for defining such abstractions, but the design of efficient operational models is a difficult task [7].
2. Termination is a fundamental property of a constraint solver which can be easily lost during solver integration. This problem becomes even more challenging when designing an open equational environment.

Our goal is to design and implement a system capable of solving problems of the following type:

Given (a) a domain $\mathcal{A}$ and a signature $\mathcal{F}_e$ of *external* operators defined over $\mathcal{A}$, together with a collection of associated constraint solvers,
(b) a set $\mathcal{F}_c$ of *data constructors*,
(c) a program $\mathcal{R}$ over $\mathcal{A} \cup \mathcal{T}$ that introduces a set $\mathcal{F}_d$ of *user-defined function symbols*, where $\mathcal{T}$ is a syntactic domain formed by the formation rules of programs and
(d) a sequence G of equations that may contain a set of variables,

find the solutions of G in the equational theory defined by $\mathcal{A}$ and $\mathcal{R}$.

$\mathcal{R}$ and G are defined by the user, and are respectively called *user program*, and the *goal* of the program. The program $\mathcal{R}$ is a conditional pattern rewrite system, a higher-order rewrite system defined over the simply-type λ terms modulo β and long η [9]. The *solution* of G is a substitution θ for the variables in G such that the formula $G\theta$ holds in the theory defined $\mathcal{R}$. [6]

2. SYSTEM ARCHITECTURE

We have designed and implemented a distributed software system called CFLP [6] whose computational model integrates lazy narrowing for conditional pattern rewrite systems with cooperative constraint solving (CCS). We employ lazy narrowing to solve systems of equations containing symbols defined by conditional pattern rewrite rules [7, 8], and CCS to support the integration of various external constraint solvers so as to produce a generic constraint solver.

The architecture of our system is shown in Fig. 1. The system consists of:

1. an *interpreter* which solves goals by a mechanism which combines lazy narrowing steps with steps that collect constraints into a constraint store and dispatch them to the cooperative constraint solving system, and

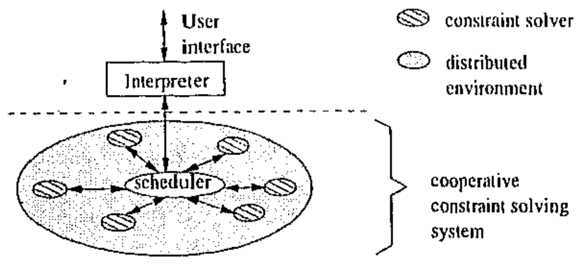


Figure 1: Architecture of CFLP

2. a distributed implementation of a cooperative constraint solver, consisting of
   (a) a *scheduler* which receives the constraints sent by the interpreter and dispatches them to specialized solvers in accordance with a given cooperation strategy, and
   (b) *specialized solvers*, which are resources located in a distributed environment and can be shared by many users.

Our choice for a distributed implementation of the cooperative constraint solver of CFLP was determined by the following observations. First, constraint solving resources are expensive; therefore they are best maintained at a developing server site and shared in a distributed environment. Secondly, our computing environment is becoming more and more net-centric; therefore instead of copying software at user side, we will in the future receive services of constraint solvers (i.e. obtain solutions) provided by constraint solvers via network.

3. MAIN FEATURES

Currently, CFLP has the following features:

- All system components are processes that communicate asynchronously over MathLink connections [10].
- The computation of the CCS is driven by a built-in cooperation strategy that has been proposed for cooperative constraint logic programming [5].
- The constraint solvers integrated in CFLP are built on top of the solving capabilities of Mathematica. We have defined and implemented solvers for
  - systems of linear equations,
  - equations with invertible functions,
  - systems of multivariate polynomial equations, and
  - differential equations.
- To improve the efficiency of the computation of the interpreter, we have integrated in CFLP all the deterministic refinements of lazy narrowing proposed by us so far [7, 8]. The user can choose the underlying calculus of the interpreter.
- The user can adjust the cooperative constraint solver of CFLP by specifying the number and location of the specialized solvers.

4. APPLICATIONS

We describe with two examples the solving capabilities of CFLP.

4.1 Electric Circuit Modeling

The first example shows how electric circuit layouts can be computed with CFLP. This example illustrates the expressive power of the FLP style programming extended with

- higher-order constructs such as λ-abstractions and function variables, and
- constraint solving capabilities for differential equations, linear equations, and systems of polynomial equations.

We first define a function spec which describes the behavior in time t of an electrical component as a function of the current $i[t]$ and voltage $v[t]$ in the circuit. The CFLP rules of spec correspond to a recursive definition, where the base case describes the behavior of elementary circuits such as resistors, capacitors, and inductors, and the inductive case describes the behavior of serial and parallel connections of electrical components.

The CFLP program is given below. We do not explain the underlying electronic laws since it should be easy to read them off from the program.

```
In[1]:= Prog = {
          spec[res[r],λ[{t},v[t]],λ[{t},i[t]]] →
            True ⇐ {λ[{t},v[t]] ≈ λ[{t},r i[t]]},
          spec[ind[l],λ[{t},v[t]],λ[{t},i[t]]] →
            True ⇐ λ[{t},v[t]] ≈ λ[{t},l i'[t]],
          spec[cap[c],λ[{t},v[t]],λ[{t},i[t]]] →
            True ⇐ λ[{t},i[t]] ≈ λ[{t},c v'[t]],
          spec[serial[{}],λ[{t},0],i] → True,
          spec[serial[[comp|T]],v,i] → True ⇐
            {spec[comp,v1,i] ≈ True,
              spec[serial[T],v2,i] ≈ True,
              λ[{t},v[t]] ≈ λ[{t},v1[t] + v2[t]]},
          spec[parallel[{}],v,λ[{t},0]] → True,
          spec[parallel[[comp|T]],v,i] → True ⇐
            {spec[comp,v,i1] ≈ True,
              spec[parallel[T],v,i2] ≈ True,
              λ[{t},i[t]] ≈ λ[{t},i1[t] + i2[t]]}};
```

The universally quantified variables are underlined. Note the usage of the list construct in the recursive specification of serial and parallel connections of electrical components. The CFLP system recognizes the following list specifications:

- $\{t_1, ..., t_n\}$: list consisting of components t_1, ..., t_n,
- $[\, h \mid tl \,]$: CFLP list with head h and tail tl in Prolog-style notation.

Consider the problem of finding the behavior in time of the current in a RLC circuit with R = 2, L = 1 and C = 1/2 (see Fig. 2), under the restrictions that the voltage is 50V and the current was initially set to 0.

In CFLP this problem is reduced to solving the system of equations:

```
In[2]:= Problem =
          {spec[serial[{res[2],ind[1],cap[1/2]}],
          λ[{t},50],i] ≈ True,
            i[0] ≈ 0};
```

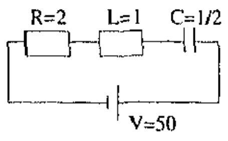


Figure 2: RLC circuit

The variable which we want to compute is i, and it is overlined. Note the usage of the expression λ[{t}, 50] for specifying that the voltage is constant (50 V) in time.

To solve this problem, we call:

```
In[3]:= TSolve[Problem,
          Constructor → {V, res, ind, cap,
                          serial, parallel},
          Rules → Prog]
Add rule :∀{r,i,v} spec[res[r], v, i] → True ⇐
  λ[{t}, v[t]] ≈ λ[{t}, r i[t]]
Add rule :∀{l,i,v} spec[ind[l], v, i] → True ⇐
  λ[{t}, v[t]] ≈ λ[{t}, l i'[t]]
Add rule :∀{c,i,v} spec[cap[c], v, i] → True ⇐
  λ[{t}, i[t]] ≈ λ[{t}, c v'[t]]
Add rule :∀{i} spec[serial[{}], λ[{t}, 0], i] → True
Add rule :∀{comp,T,v1,v2,i,v}
    spec[serial[[comp|T]], v, i] → True ⇐
      spec[comp, v1, i] ≈ True,
      spec[serial[T], v2, i] ≈ True,
      λ[{t}, v[t]] ≈ λ[{t}, v1[t] + v2[t]]
Add rule :∀{v} spec[parallel[{}], v, λ[{t}, 0]] → True
Add rule :∀{comp,i,i1,i2,T,v}
    spec[parallel[[comp|T]], v, i] → True ⇐
      spec[comp, v, i1] ≈ True,
      spec[parallel[T], v, i2] ≈ True,
      λ[{t}, i[t]] ≈ λ[{t}, i1[t] + i2[t]]
{i → λ[{t}, -c1$2 e^-t Sin[t]]}
Out[3]= {{i → λ[{t}, -c1$2 e^-t Sin[t]]}}
```

This call is similar to the `Solve` call of *Mathematica*, but `TSolve` has the specific options

- `Rules`: provides the functional logic program (i.e., conditional rewrite system) used during solving the goal,
- `Constructor`: specifies the data constructors used in the specification of the program and of the goal.

The system computes the parametric solution

$$\{i \to \lambda t.-c1\$2\ e^{-t}\sin(t)\}$$

which is represented in *Mathematica* by

$$\{i \to \lambda[\{t\}, -c1\$2\ e^{-t}\ \sin[t]]\}.$$

4.2 Program Calculation

This example illustrates the capabilities of a computing environment which can perform full higher-order pattern unification and higher-order term rewriting.

Consider the problem of writing a program to check whether a list of numbers is *steep*, i.e, if every element of the list is greater than or equal to the average of the elements that follow it. A CFLP program that does this is:

```
In[4]:= Prog = {
            steep[{}] → True,
            steep[[a|x]] → And[a * len[x] ≥ sum[x],
                                steep[x]],
            sum[{}] → 0, sum[[x|y]] → x + sum[y],
            len[{}] → 0, len[[x|y]] → 1 + len[y]};
```

Prog is modular and easy to understand, but it is inefficient (quadratic complexity). It is desirable to compute the efficient version steepOpt of steep. For doing this, we employ the *fusion calculational rule*:

$$\frac{f[e] = e' \quad f[g[a, ns]] = h[a, f[ns]]}{f[\mathrm{foldr}[g, e, ns]] = \mathrm{foldr}[h, e', ns]}$$

where foldr is defined recursively by

$$\mathrm{foldr}[g, e, \{\}] = e,\ \mathrm{foldr}[g, e, [n|ns]] = g[n, \mathrm{foldr}[g, e, ns]].$$

The inefficient computation is f[foldr[g,e,ns]], and its efficient version is foldr[h,e',ns]. Finding the efficient version amounts to finding h such that f[g[a,ns]]=h[a,f[ns]].

To use the fusion calculational rule, we first formalize the the computation of steep as f[foldr[g,e,ns]]. It is easy to see that

$$\mathrm{steep}[n]=\mathrm{sel}\cdot c3\cdot 1[n]=\ \mathrm{sel}\cdot c3\cdot 1[f[\mathrm{foldr}[g, e, n]]],$$

where f[n]=c3[steep[n],sum[n],len[n]], g=Cons and e={ }. Here, sel·c3·1 denotes the first data selector associated of the data constructor c3. This implies that the following relation holds for any valid arguments x, y, z:

$$\mathrm{sel}\cdot c3\cdot 1[c3[x, y, z]] = x.$$

If we succeed to find h, then

$$\mathrm{steepOpt}[n] = \mathrm{sel}\cdot c3\cdot 1[f[\mathrm{foldr}[h, e', n]]]$$

where e'=f[e]=c3[True,0,0].

To compute h, we solve the equation f[g[a,ns]]=h[a,f[ns]], i.e. the goal

```
In[5]:= Problem = λ[{n, ns}, f[[n|ns]]] ≈
            λ[{n, ns},
             h̄[n, c3[steep[ns], sum[ns], len[ns]]]];

In[6]:= Constructor[Triple = c3[Bool, Float, Float]];

In[7]:= TSolve[Problem,
          Constructor →
           {Plus, Power, Times, GreaterEqual, And},
          Rules → ProgU
           {f[ns] → c3[steep[ns], sum[ns], len[ns]]}]
Add rule :∀{ns} f[ns] → c3[steep[ns], sum[ns], len[ns]]
Add rule :∀{x,y} len[[x|y]] → 1 + len[y]
Add rule :len[{}] → 0
Add rule :∀{a,x} steep[[a|x]] → alen[x] ≥ sum[x]&&steep[x]
Add rule :steep[{}] → True
Add rule :∀{x,y} sum[[x|y]] → x + sum[y]
Add rule :sum[{}] → 0
{h → λ[{x$23, x$24},
    c3[x$23 sel·c3·3[x$24] ≥ sel·c3·2[x$24]&&
        sel·c3·1[x$24], x$23 + sel·c3·2[x$24],
      1 + sel·c3·3[x$24]]]}
Out[7]= {{h → λ[{x$23, x$24}, c3[
             x$23 sel·c3·3[x$24] ≥ sel·c3·2[x$24]&&
              sel·c3·1[x$24],
             x$23 + sel·c3·2[x$24],
             1 + sel·c3·3[x$24]]]}}
```

Note that CFLP handles the operators + (Plus), Power, * (Times), and ≥ (GreaterEqual) as external operators, that is, operators used in constraint specifications. As a consequence, the equations built with these operators are handled by default as constraints and sent to be solved by the constraint solver. In this example we override the default specification of these operators by declaring them as mere constructors in the `Constructor` option of the `TSolve` call. As a consequence, CFLP gives up using constraint solvers for solving equations built with the operators mentioned above, and enables the interpreter to solve them.

4.3 Other Applications

We have investigated several other problems from physics and found that they can be solved with CFLP by reducing them to systems of constraints that can be handled by a cooperation between the specialized solvers of CFLP.

5. CORBA-BASED OPEN ENVIRONMENT

We are currently working on redesigning the scheduler to accept different cooperation strategies provided by the user.

The work includes the replacement of the modules that rely on MathLink by more versatile protocols. We use CORBA [1] in the new implementation because it is the standardized software technology for distributed computing and it fits well with the objective of our open environment for equational solving. CORBA facilitates

(a) to get uniform access to various applications (such as specialized solvers) written in different languages, via object modeling, when they have a common interface specified in OMG IDL,

(b) to locate transparently solving resources in distributed environments, and

(c) to plug them in whenever their capabilities are needed to solve a problem.

As CORBA provides the framework for cooperative constraint solvers and low-level protocols for communication among objects, our task is to define higher-level protocols for communicating among distributed solvers.

5.1 MAXCOR

We have designed MAXCOR (MAth eXchange for CORBA), a framework for exchanging mathematical expressions for CORBA. MAXCOR contain three ingredients:

1. the language for mathematical formulas to be used in conjunction with OMG IDL,

2. the common interface definition in OMG IDL for all objects in MAXCOR,

3. CORBA object wrappers for binding MAXCOR and existing applications.

As for point 1., we adopted MathML [3], the subset of XML. In a simply-minded implementation, MathML documents could be passed as arguments of string type in OMG IDL. However, this would require translation of object-to-string and vice versa on both client and server sides. Instead for efficiency reasons, we employ XML DOM (Document Object Model) and DOM-to-valuetype mapping [2]. XML DOM is standardized by W3C and its implementation is available as an API. Furthermore, DOM-to-valuetype mapping is being specified by OMG and is a part of XML-to-valuetype mapping. DOM-to-valuetype mapping is entirely defined in OMG IDL. Thus our design is platform-independent.

As for point 2., we show below a simplified version of the MAXCOR module declaration. Here the server is a solver and the client is the scheduler that calls the solver.

```
#include"value_dom.idl"

module MAXCOR {
  struct threadID {
    sequence<octet> passwd;
      // password for this thread
    long number;
      // thread number identifying the thread
      // in which the solver runs
  };
  exception mException{
    string reason;
  };

  interface MAXCORObject {
    threadID init(in string parameter)
      raises(mException);
    void deinit(
      in threadID id, in string parameter)
      raises(mException);
    Document execute(
      in threadID id, in string command,
      in Document expr) raises(mException);
    Document serviceSpec(in string command)
      raises(mException);
    string usage(in string parameter);
  };
};
```

In the above program, `value_dom.idl` is the declaration file of XML-to-valuetype mapping and `Document` is the valuetype of DOM object. Type definition of `threadID` is self-explanatory. Operations `init`, `deinit` and `execute` do the real work. Operation `init` creates new threads for running the server, initializes the running environment and establishes the session with its client. Operation `deinit` dismantles the server process when the server completes the service. Operation `execute` invokes the program of the solver. The arguments of `execute` are a string of command name, and an expression which will be evaluated by the server.

Operations `serviceSpec` and `usage` are auxiliary and provide information to the client. Operation `usage` returns the help message related to the service of the solver. Operation `serviceSpec` in our present version returns specification of the argument of the solver.

5.2 Implementation

We have implemented the following APIs: DOM to valuetype mapping in C++ and Java, CORBA object wrappers for Mathematica in C++ and Java and for CPLEX [4] in C++. CPLEX is an application package for integer programming and its wrapped API is used as one of the solvers for CFLP. The solvers that the present version of CFLP are using are all written in Mathematica with the interface complied with MathLink. This interface will be replaced by CORBA object wrappers. We will use MAXCOR for communication between a scheduler and solvers in CFLP. Currently we are working for the full deployment of solvers wrapped as CORBA objects that cooperate with the scheduler of CFLP.

6. CONCLUSION

We introduced an open environment for equational solving. The core of the environment is CFLP system which supports equational reasoning over variety of domains. We showed main features of the system by giving examples that exploits the style of equational reasoning. We further discussed a new implementation of the open environment which is under way based on CORBA technology.

7. ACKNOWLEDGMENTS

This work is supported in part by the Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Scientific Research (B) 12480066, 2000 and Mircea Marin is supported by JSPS postdoc fellowship.

8. REFERENCES

[1] `http://www.corba.org`.

[2] `http://www.omg.org/techprocess/meetings/schedule/XML_Value_RFP.html`.

[3] `http://www.w3.org/Math/`.

[4] Using the CPLEX Callable Library. CPLEX Optimization, Inc., USA, 1995.

[5] H. Hong. CLP(CF): Constraint Logic Programming over Complex Functions. Technical Report 94-09, RISC-Linz, Castle of Hagenberg, Austria, 1992.

[6] M. Marin. *Functional Logic Programming with Distributed Constraint Solving*. PhD thesis, Institute RISC-Linz, Johannes Kepler University, Castle of Hagenberg, Austria, 2000.

[7] M. Marin, T. Ida, and T. Suzuki. On Reducing the Search Space of Higher-Order Lazy Narrowing. In A. Middeldorp and T. Sato, editors, *FLOPS'99*, volume 1722 of *LNCS*, pages 225–240. Springer-Verlag, 1999.

[8] M. Marin, T. Ida, and T. Suzuki. Lazy Narrowing Calculi in Perspective. In *Proceedings of the 9th International Workshop on Functional and Logic Programming*, pages 238–252, Benisassim, Spain, 2000.

[9] T. Nipkow. Functional unification of higher-order patterns. In *Proceedings of 8th IEEE Symposium on Logic in Computer Science*, pages 64–74, 1993.

[10] S. Wolfram. *The Mathematica Book*. Fourth Edition, Wolfram Media Inc. Champaign, Illinios, USA, and Cambridge University Press, 1999.

CFLP: A *Mathematica* Implementation of a Distributed Constraint Solving System

Mircea Marin, Tetsuo Ida

Institute of Information Sciences and Electronics
University of Tsukuba
Tsukuba Ibaraki 305-8573, Japan
mmarin@score.is.tsukuba.ac.jp, ida@score.is.tsukuba.ac.jp

Wolfgang Schreiner

Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University
A-4040 Linz, Austria
Wolfgang.Schreiner@risc.uni-linz.ac.at

The need for combining and making various constraint solvers cooperate is widely recognized. Such an integrated system would allow solving problems that cannot be solved by a single solver.

CFLP (Constraint Functional Logic Programming System) is a distributed software system consisting of a functional logic programming interpreter running on one machine and a number of constraint-solving engines running on other machines. The interpreter is based on a deterministic version of a lazy narrowing calculus which was extended in two main directions: (a) the possibility to specify explicit OR-parallelism, and (b) the possibility to specify constraints over various domains. The OR-parallel features of the interpreter allow the decomposition of the solution space into different subspaces denoted by various sets of constraints; the individ-

Originally presented at the Third International *Mathematica* Symposium (Hagenberg, Austria, August 23–25, 1999).

$$\begin{aligned}&\{a \to \\ &\quad 25m+n+p+q+\frac{1}{3}\left(-18m-3q-\frac{11r}{2}+\frac{11}{2}(-24m+r)\right)+\frac{1}{4}\left(8m-4p-6q-\right.\\ &\qquad \left.7r+7(-24m+r)+2\left(18m+3q+\frac{11r}{2}-\frac{11}{2}(-24m+r)\right)\right),\\ &b \to \frac{1}{4}\left(-8m+4p+6q+7r-7(-24m+r)-\right.\\ &\qquad \left.2\left(18m+3q+\frac{11r}{2}-\frac{11}{2}(-24m+r)\right)\right),\\ &c \to \frac{1}{3}\left(18m+3q+\frac{11r}{2}-\frac{11}{2}(-24m+r)\right),\\ &d \to -24m+r\}\end{aligned}$$

Out[3]=
$$\begin{aligned}&\{\{a \to 25m+n+p+q+\frac{1}{3}\left(-18m-3q-\frac{11r}{2}+\frac{11}{2}(-24m+r)\right)+\frac{1}{4}\\ &\qquad \left(8m-4p-6q-7r+7(-24m+r)+2\left(18m+3q+\frac{11r}{2}-\frac{11}{2}(-24m+r)\right)\right),\\ &\quad b \to \frac{1}{4}\left(-8m+4p+6q+7r-7(-24m+r)-\right.\\ &\qquad \left.2\left(18m+3q+\frac{11r}{2}-\frac{11}{2}(-24m+r)\right)\right),\\ &\quad c \to \frac{1}{3}\left(18m+3q+\frac{11r}{2}-\frac{11}{2}(-24m+r)\right),\ d \to -24m+r\}\}\end{aligned}$$

The equational symbol $\doteq$ denotes *strict* equality, that is, the fact that the *values* of the two sides of the equation coincide. We identify the value of an expression with the result of rewriting it to an expression without defined symbols.

The syntax call of `TSolve` is similar to the syntax call of `Solve`. The first three arguments specify the goal to be solved, the main variables of the goal, and the eliminatable variables of the goal, respectively. In addition, the user of CFLP can control the behavior of the computation through various `TSolve` options. In this example, the following options are used.

`DefinedSymbol`	the list of type-annotated symbols defined by the user
`Rules`	the list of definitions for the user-defined symbols (the functional logic program)
`Constructor`	the list of type-annotated data constructors

The constant coefficients of f are given as data constructors, and the unknown coefficients of g are given as main variables.

A `TSolve` call may yield an infinite number of solutions. In this case it cannot produce an output with the list of all solutions but it prints them out incrementally, as soon as they are computed. `TSolve` calls can be interrupted via a palette which is provided for the user convenience.

In this example, CFLP returns the unique solution

$$\{d \to -24m + r,$$
$$c \to 15m + 7n - \frac{7}{6}(60m + 6n) + 3p - \frac{3}{2}(50m + 12n - 2(60m + 6n) + 2p) + q,$$
$$b \to \frac{1}{2}(50m + 12n - 2(60m + 6n) + 2p),\ a \to \frac{1}{6}(60m + 6n)\}$$

which is printed out as soon as it is computed, and collected in the output of the **TSolve** call.

Note the use of higher-order variables and λ-abstractions in the specification of the goal and functional logic program. CFLP is able to handle equations involving operators defined outside the functional logic program. Furthermore, the computed answer is parametric, since *r* is a variable.

□ Electrical Circuits

This example illustrates the expressive power of the functional logic programming style extended with higher-order constructs (function variables and λ-abstractions) and capabilities to solve differential equations, linear equations, and systems of polynomial equations. We consider the problem of modeling the behavior in time of electrical circuits built from serial and parallel connections of elementary components such as resistors, inductors, and capacitors.

First we introduce a new type constructor **ElComp** with associated data types **res**, **ind**, **cap**, **serial**, and **parallel** to specify the characteristics of the electrical circuits. Thus, a resistor with resistance R = 2 is specified as **res[**2**]**, a serial connection of two resistors R1 and R2 is specified as **serial[{res[**R1**]**, **res[**R2**]}]**, and a parallel connection of a capacitor C and an inductor L is specified as **parallel[{cap[**C**]**, **ind[**L**]}]**. The keyword **TyList** is the CFLP-type constructor for lists.

```
In[4]:= TypeConstructor[ElComp = res[Float] | ind[Float] | cap[Float] |
            serial[TyList[ElComp]] | parallel[TyList[ElComp]]];
```

Next, we define a function **spec** which describes the behavior in time **t** of an electrical component as a function of current **i[t]** and voltage **v[t]** in the circuit. The CFLP definition of **spec** is recursive, where the base case describes the behavior in time of the elementary circuits (resistor, inductor, capacitor), and the inductive case describes the behavior in time of serial and parallel connections of electrical components.

The CFLP program is given below. We do not explain the underlying electronic laws since it should be easy to read them off from the program.

```
In[5]:= Prog =
          {spec[res[r], v : Float → Float, i : Float → Float] →
            True ⇐ {λ[{t}, v[t]] ≈ λ[{t}, r i[t]]},
           spec[ind[l], v : Float → Float, i : Float → Float] →
            True ⇐ λ[{t}, v[t]] ≈ λ[{t}, l i'[t]],
           spec[cap[c], v : Float → Float, i : Float → Float] →
            True ⇐ λ[{t}, i[t]] ≈ λ[{t}, c v'[t]],
           spec[serial[{}], λ[{t}, 0], i : Float → Float] → True,
           spec[serial[[comp | T]],
             v : Float → Float, i : Float → Float] → True ⇐
            {spec[comp, v1, i] ≈ True,
             spec[serial[T], v2, i] ≈ True,
             λ[{t}, v[t]] == λ[{t}, v1[t] + v2[t]]},
           spec[parallel[{}], v, λ[{t}, 0]] → True,
           spec[parallel[[comp | T]], v, i] → True ⇐
            {spec[comp, v, i1] ≈ True,
             spec[parallel[T], v, i2] ≈ True,
             λ[{t}, i[t]] == λ[{t}, i1[t] + i2[t]]}};
```

Consider the problem of finding the behavior in time of the current in an RLC circuit, under the restriction that the voltage is constant in time and the current was initially set to 0.

In this case, the goal which we want to solve (in variable **i**) is

```
In[6]:= G = {i[0] == 0, spec[serial[{res[R0], ind[L0], cap[C0]}],
            λ[{t : Float}, V], λ[{t : Float}, i[t]]] ≐ True};
```

R0, **L0**, **C0**, and **V** are data constructors which denote arbitrary but fixed characteristic values for the resistor, inductor, capacitor, and voltage of the circuit. Now we can ask CFLP to solve the problem.

```
In[7]:= TSolve[G, {i : Float → Float}, {},
          Rules → Prog,
          Constructor → {R0 : Float, L0 : Float, C0 : Float, V : Float},
          DefinedSymbol → {spec : ElComp × (Float → Float) × (Float → Float) → Bool}]
```

```
Type checking program ...
```

```
Type checking goal ...
```

$$\left\{i \to \lambda\left[\{t\cdot 1\},\ C0\left(-\frac{e^{\frac{\left(-\sqrt{C0}\,R0-\sqrt{-4\,L0+C0\,R0^2}\right)t\cdot 1}{2\sqrt{C0}\,L0}}\left(-\sqrt{C0}\,R0+\sqrt{-4\,L0+C0\,R0^2}\right)C1_2}{2\sqrt{C0}\,L0}+\frac{e^{\frac{\left(-\sqrt{C0}\,R0+\sqrt{-4\,L0+C0\,R0^2}\right)t\cdot 1}{2\sqrt{C0}\,L0}}\left(-\sqrt{C0}\,R0+\sqrt{-4\,L0+C0\,R0^2}\right)C1_2}{2\sqrt{C0}\,L0}\right)\right]\right\}$$

Out[7]=
$$\left\{\left\{\mathtt{i}\to\lambda\left[\{\mathtt{t}\cdot 1\},\ \mathrm{CO}\left(-\frac{e^{\frac{\left(-\sqrt{\mathrm{CO}}\ \mathrm{RO}-\sqrt{-4\,\mathrm{LO}+\mathrm{CO}\,\mathrm{RO}^2}\right)\mathtt{t}\cdot 1}{2\sqrt{\mathrm{CO}}\ \mathrm{LO}}}\left(-\sqrt{\mathrm{CO}}\ \mathrm{RO}+\sqrt{-4\,\mathrm{LO}+\mathrm{CO}\,\mathrm{RO}^2}\right)\mathrm{C1}_2}{2\sqrt{\mathrm{CO}}\ \mathrm{LO}}+\frac{e^{\frac{\left(-\sqrt{\mathrm{CO}}\ \mathrm{RO}+\sqrt{-4\,\mathrm{LO}+\mathrm{CO}\,\mathrm{RO}^2}\right)\mathtt{t}\cdot 1}{2\sqrt{\mathrm{CO}}\ \mathrm{LO}}}\left(-\sqrt{\mathrm{CO}}\ \mathrm{RO}+\sqrt{-4\,\mathrm{LO}+\mathrm{CO}\,\mathrm{RO}^2}\right)\mathrm{C1}_2}{2\sqrt{\mathrm{CO}}\ \mathrm{LO}}\right)\right]\right\}\right\}$$

The system computes the parametric solution

$$\left\{i\to\lambda t.e^{\frac{\left(-\sqrt{\mathrm{CO}}\ \mathrm{RO}-\sqrt{-4\ \mathrm{LO}+\mathrm{CO}\ \mathrm{RO}^2}\right)t}{2\sqrt{\mathrm{CO}}\ \mathrm{LO}}}\,\mathrm{C1}_1-e^{\frac{\left(-\sqrt{\mathrm{CO}}\ \mathrm{RO}+\sqrt{-4\ \mathrm{LO}+\mathrm{CO}\ \mathrm{RO}^2}\right)t}{2\sqrt{\mathrm{CO}}\ \mathrm{LO}}}\,\mathrm{C1}_1\right\}$$

which is represented in *Mathematica* by

$$\boxed{\left\{\mathtt{i}\to\lambda\left[\{\mathtt{t}\},\ e^{\frac{\left(-\sqrt{\mathrm{CO}}\ \mathrm{RO}-\sqrt{-4\,\mathrm{LO}+\mathrm{CO}\,\mathrm{RO}^2}\right)\mathtt{t}}{2\sqrt{\mathrm{CO}}\ \mathrm{LO}}}\,\mathrm{C1}_1-e^{\frac{\left(-\sqrt{\mathrm{CO}}\ \mathrm{RO}+\sqrt{-4\,\mathrm{LO}+\mathrm{CO}\,\mathrm{RO}^2}\right)\mathtt{t}}{2\sqrt{\mathrm{CO}}\ \mathrm{LO}}}\,\mathrm{C1}_1\right]\right\}}$$

Note the usage of type annotations in the goal, list of variables, and options of the **TSolve** call. CFLP has integrated a polymorphic type checker to verify the type consistency of the program and goal provided by the user.

☐ A Problem Involving Solver Cooperation

Consider the following program

```
In[8]:=  Needs["TSolve`"]

In[9]:=  Prog = {f[X : Compl] → g[Y : Compl] ⇐ (X + Y == 3 ∨ X² - Y == 9)};
```

in complex variables X, Y, and the goal

```
In[10]:= G = {f[X] ≈ g[Y], g[Y²] ≈ g[Z² - 1],
            λ[{x : Compl}, H'[x]] == λ[{x : Compl}, Z x^Y], H'[1] == 4};
```

with indeterminates X, H, Y, Z. In this example, the operator ∨ denotes logical disjunction, and can be used in goals and conditional parts of rewrite rules to express alternative solutions.

Solving this goal requires a cooperation among solvers for linear, polynomial, and differential equations over the domain of complex numbers. To solve G, we call

```
In[11]:= TSolve[G, {X, H, Y, Z}, {},
          Rules → Prog,
          DefinedSymbol → {f : Compl → Compl},
          Constructor → {g : Compl → Compl}]
```

```
Type checking program ...
Type checking goal ...
```

```
Enabling distributed constraint solving subsystem ...
```

$$\left\{X\to 3+\sqrt{15},\ Y\to -\sqrt{15},\ Z\to 4,\ H\to \lambda\left[\{x\bullet 1\},\ \frac{4\,x\bullet 1^{1-\sqrt{15}}}{1-\sqrt{15}}+C1_2\right]\right\}$$

$$\left\{X\to 3-\sqrt{15},\ Y\to \sqrt{15},\ Z\to 4,\ H\to \lambda\left[\{x\bullet 1\},\ \frac{4\,x\bullet 1^{1+\sqrt{15}}}{1+\sqrt{15}}+C1_4\right]\right\}$$

$$\left\{X\to -\sqrt{9-\sqrt{15}},\ Y\to -\sqrt{15},\ Z\to 4,\ H\to \lambda\left[\{x\bullet 1\},\ \frac{4\,x\bullet 1^{1-\sqrt{15}}}{1-\sqrt{15}}+C1_6\right]\right\}$$

$$\left\{X\to -\sqrt{9+\sqrt{15}},\ Y\to \sqrt{15},\ Z\to 4,\ H\to \lambda\left[\{x\bullet 1\},\ \frac{4\,x\bullet 1^{1+\sqrt{15}}}{1+\sqrt{15}}+C1_8\right]\right\}$$

$$\left\{X\to \sqrt{9-\sqrt{15}},\ Y\to -\sqrt{15},\ Z\to 4,\ H\to \lambda\left[\{x\bullet 1\},\ \frac{4\,x\bullet 1^{1-\sqrt{15}}}{1-\sqrt{15}}+C1_{10}\right]\right\}$$

$$\left\{X\to \sqrt{9+\sqrt{15}},\ Y\to \sqrt{15},\ Z\to 4,\ H\to \lambda\left[\{x\bullet 1\},\ \frac{4\,x\bullet 1^{1+\sqrt{15}}}{1+\sqrt{15}}+C1_{12}\right]\right\}$$

Out[11]=

$$\left\{\left\{X\to 3+\sqrt{15},\ Y\to -\sqrt{15},\ Z\to 4,\ H\to \lambda\left[\{x\bullet 1\},\ \frac{4\,x\bullet 1^{1-\sqrt{15}}}{1-\sqrt{15}}+C1_2\right]\right\},\right.$$

$$\left\{X\to 3-\sqrt{15},\ Y\to \sqrt{15},\ Z\to 4,\ H\to \lambda\left[\{x\bullet 1\},\ \frac{4\,x\bullet 1^{1+\sqrt{15}}}{1+\sqrt{15}}+C1_4\right]\right\},$$

$$\left\{X\to -\sqrt{9-\sqrt{15}},\ Y\to -\sqrt{15},\ Z\to 4,\ H\to \lambda\left[\{x\bullet 1\},\ \frac{4\,x\bullet 1^{1-\sqrt{15}}}{1-\sqrt{15}}+C1_6\right]\right\},$$

$$\left\{X\to -\sqrt{9+\sqrt{15}},\ Y\to \sqrt{15},\ Z\to 4,\ H\to \lambda\left[\{x\bullet 1\},\ \frac{4\,x\bullet 1^{1+\sqrt{15}}}{1+\sqrt{15}}+C1_8\right]\right\},$$

$$\left\{X\to \sqrt{9-\sqrt{15}},\ Y\to -\sqrt{15},\ Z\to 4,\ H\to \lambda\left[\{x\bullet 1\},\ \frac{4\,x\bullet 1^{1-\sqrt{15}}}{1-\sqrt{15}}+C1_{10}\right]\right\},$$

$$\left.\left\{X\to \sqrt{9+\sqrt{15}},\ Y\to \sqrt{15},\ Z\to 4,\ H\to \lambda\left[\{x\bullet 1\},\ \frac{4\,x\bullet 1^{1+\sqrt{15}}}{1+\sqrt{15}}+C1_{12}\right]\right\}\right\}$$

In this case, the solutions computed by CFLP are parametric, and the system has generated the auxiliary variables $C1_7$, $C1_8$, $C1_9$, $C1_{10}$, $C1_{11}$, $C1_{12}$ to express them. It is not hard to see that these are all the solutions of **G**.

The Structure of the System

CFLP is a distributed software system for solving systems of equational goals in theories that can be represented as sets of conditional rewrite rules over a term algebra whose signature is extended with external operators. The external operators are used for expressing constraints over various domains.
The system consists of three components:

- an interpreter based on a higher-order lazy narrowing calculus C
- a cooperative constraint solver consisting of
  - a scheduler which implements a strategy S for solver cooperation
  - various specialized constraint solvers

The general architecture of the system is depicted below.

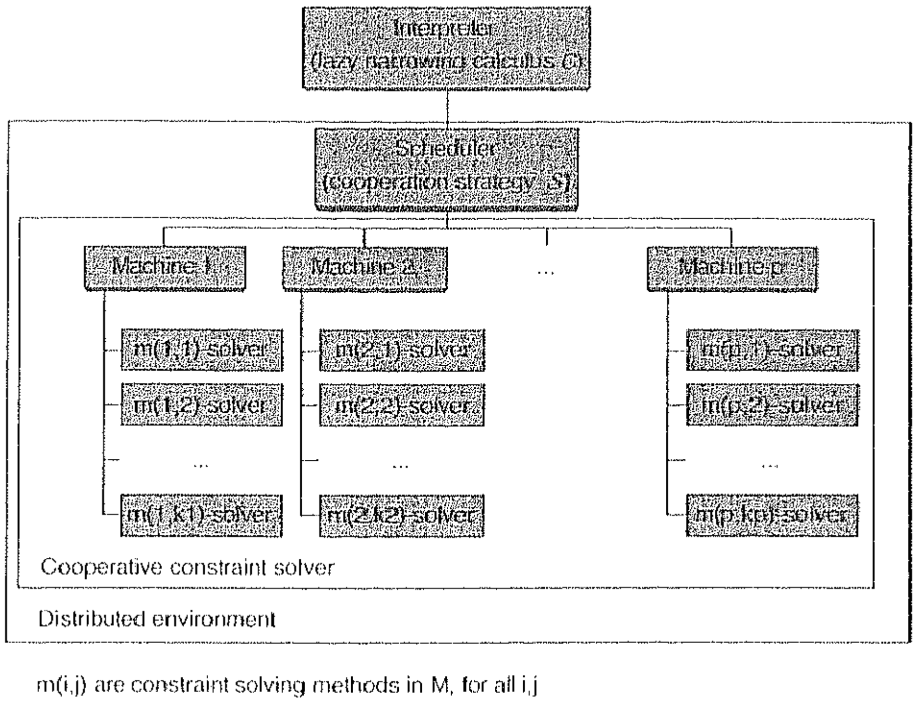


Figure 1. The architecture of CFLP.

The Interpreter

The CFLP interpreter is designed to solve systems of equations between simply-typed λ-terms in theories axiomatized by finite sets of conditional rewrite rules of a certain kind, called *pattern rewrite rules*.

Roughly speaking, simply-typed λ-terms are analogous to the **`Function`** construct of *Mathematica*, modulo the following differences.

1. The keyword `Function` is replaced with the keyword λ.

2. Each such construct (which we will henceforth call a *λ-term*) should be well-typed, that is, a type can be inferred from the types of its base components (variables, constants, and function symbols).

The current version of CFLP recognizes the following base types:

Type	Literals
`Int`	Integer literals of *Mathematica*
`Float`	Real literals of *Mathematica*
`Compl`	Complex literals of *Mathematica*
`String`	String literals of *Mathematica*
`Bool`	True, False

and the following constructed types:

$\tau_1 \times \cdots \times \tau_n \to \tau$	type of functions with n arguments of types $\tau_1,\ldots,\tau_n$, and result of type τ
`TyList[`τ`]`	type of lists with elements of type τ

For lists, we have adopted both the *Mathematica* notation $\{a_1, \ldots, a_n\}$ to represent a list with elements $a_1, \ldots, a_n$, and the Prolog-like notation `[H|T]` to represent a list with head H and tail T.

The CFLP interpreter is based on a higher-order lazy narrowing calculus **C** for pattern rewrite systems [3] extended to handle also conditional pattern rewrite systems [4]. The underlying calculus essentially consists of the unification rules of higher-order patterns [5] and the lazy narrowing rules. Since the search space of such a calculus is extremely large, we have implemented a couple of refinements with smaller search space for solutions, which are sound and complete for classes of functional logic programs and of practical interest.

We have extended these calculi in two main directions:

1. the possibility to specify *constraints*, that is, equations that cannot be solved by narrowing, but for which specialized solvers are available; and

2. the possibility to specify explicit OR-parallelism.

The interpreter successively decomposes the goal toward an answer substitution by applying the inference steps of the underlying lazy narrowing calculus. The only equations that cannot be solved in this way are the constraints. The constraints generated upon derivations are collected and sent to be solved by specialized constraint solvers via the component called the *scheduler* (see Figure 1).

Note that the nondeterministic selection of a rewrite rule for a defined symbol and the explicit OR-formulas cause the initial goal to be reduced to disjoint sets of constraints that can be solved in parallel. Thus, in the last illustrative example, the reduction of the initial goal

$$f[X] \approx g[Y],\ g[Y^2] \approx g[Z^2 - 1],$$
$$\lambda[\{x : \mathrm{Compl}\},\ H'[x]] \approx \lambda[\{x : \mathrm{Compl}\},\ Zx^Y],\ H'[1] == 4$$

involves the decomposition of the equation $f[X] \approx g[Y]$ into simpler equations. The transformation step performed by our lazy narrowing calculus is

$$f[X] \approx g[Y] \Rightarrow X \approx \mathrm{X0},\ g[\mathrm{Y0}] \approx g[Y],\ (\mathrm{X0} + \mathrm{Y0} == 3 \vee \mathrm{X0}^2 - \mathrm{Y0} == 9) \qquad (2)$$

where X0, Y0 are new variables. In this step we have used the fresh variant

$$f[\mathrm{X0}] \to g[\mathrm{Y0}] \Leftarrow (\mathrm{X0} + \mathrm{Y0} == 3 \vee \mathrm{X0}^2 - \mathrm{Y0} == 9)$$

of the rewrite rule which defines *f*.

During step (2), an OR-subgoal is produced and as a result the initial goal is finally decomposed into two disjoint sets of constraints. These sets of constraints are sent to be solved to the scheduler.

□ The Scheduler

The scheduler coordinates the process of solving the systems of constraints received from the interpreter. In order to solve these sets of constraints, the constraint scheduler maintains a dynamic data structure called a *constraint tree*. The nodes of the constraint tree are pairs of the form $\langle\theta, cs\rangle$ where θ is a substitution and cs is a set of constraints.

Whenever a set of constraints cs is received from the interpreter, the scheduler adds a new son $\langle\epsilon, cs\rangle$ to the root of the constraint tree. Here ϵ stands for the empty substitution. The scheduler expands this tree by applying constraint-solving methods in parallel to its leaf nodes.

A leaf node $\langle\theta, cs\rangle$ is expanded with respect to a method m as follows.

1. cs is decomposed into a set cs_1 of constraints to which method m can be applied, and a set cs_2 of other constraints.
2. cs_1 is sent to be solved to a constraint solver which implements method m. We call such a solver an *m-solver*.
3. If the m-solver detects cs_1 inconsistent, then the node $\langle\theta, cs\rangle$ is marked as inconsistent. Otherwise the m-solver returns $\langle\epsilon, cs\rangle$ if it cannot reduce cs_1, or it computes a finite sequence of pairs $\langle\theta_1, cs'_1\rangle, \ldots, \langle\theta_p, cs'_p\rangle$ with the property that θ is a solution of cs if and only if there exists a solution σ_i of some cs'_i $(1 \le i \le p)$, such that $\theta = \theta_i\,\sigma_i$.
4. If the sequence $\langle\theta_1, cs'_1\rangle, \ldots, \langle\theta_p, cs'_p\rangle$ is computed by the m-solver, then the nodes $\langle\theta\theta_i,\ cs_2\,\theta_i \cup cs'_i\rangle$ $(1 \le i \le p)$ are added to the constraint tree as sons of the node $\langle\theta, cs\rangle$.

A node $\langle\theta, cs\rangle$ is *final* if it cannot be reduced by any m-solver which is available.

The implementation of the scheduling algorithm is inspired by the work of Hong [6]. The scheduler can be regarded as a component parameterized with a list $M = \{m_1, \ldots, m_k\}$ of constraint-solving methods. The scheduler implements a cooperation strategy S, which repeatedly applies the sequence $m_1, \ldots, m_k$ of methods to the leaves of the constraint tree until they become either final or inconsistent. As soon as a final node is generated, it is made accessible to the interpreter.

□ The Constraint Solvers

The constraint solvers are implementations of the constraint-solving methods specified to the scheduler through the list M. The current implementation of CFLP provides constraint solvers, which implement the following methods for solving constraints over the domains of real and complex numbers.

`Linear`	a method for solving systems of linear equations and systems of equations with invertible functions
`Polynomial`	a method for solving systems of polynomial equations
`Derivative`	a method for solving derivative equations
`PartialDerivative`	a method for solving partial derivative equations

These methods are tried in the order presented.

All solvers are implemented by separate *Mathematica* processes executing in parallel and communicating with the constraint scheduler via *MathLink* connections. There are two types of constraint solvers in CFLP.

1. Local constraint solvers that run as subsidiary *Mathematica* processes of the CFLP constraint scheduler.
2. Shared constraint solvers are started from outside a CFLP session and can be connected later to more CFLP schedulers, which may run on a different machine.

The user can adjust the distributed constraint-solving component of the system by specifying the number of local constraint solvers that are started at system initialization, and the remote machines on which to look for shared constraint solvers.

The communication mechanism between scheduler and constraint solvers is implemented completely in *MathLink* [7]. Therefore, CFLP is a platform-independent software system and can be used in heterogeneous networks.

■ Conclusions and Future Work

CFLP is a software system consisting of a functional logic interpreter and a distributed constraint-solving system, which provides support for solving systems of equations over various constraint domains. The functional logic component allows the user to define his own abstractions in an easy and comfortable way. In the current implementation we have integrated solvers for linear equations and equations with invertible functions, polynomial equations, differential equations, and partial differential equations. The constraint solvers are all implemented on top of the constraint-solving capabilities of *Mathematica*.

We intend to further develop the system by integrating more constraint-solving capabilities. The constraint solvers may act either on disjoint sets of constraints or on overlapping subsystems. Currently, the constraint solvers are not allowed to act simultaneously on a leaf node of the constraint tree. An optimization would be to act simultaneously with more solvers on the same node in situations when the subsystems of equations are non-overlapping.

Because of the higher-order extensions of the functional logic programming framework (function variables and λ-abstractions), the search space of lazy narrowing is very large. We have identified and implemented various refinements of a lazy narrowing calculus that perform a more deterministic search for solutions for classes of functional logic programs of practical interest. We intend to continue our research and identify better refinements.

The system is intended to be used by researchers in functional and logic programming languages, and by researchers in constraint solving who are willing to make use of its expressive and computational power.

■ References

[1] J. Jaffar and J.-L. Lassez, *Constraint Logic Programming*, Technical Report 86–74, Department of Computer Science, Monash University, Clayton, 1986.

[2] M. Marin, T. Ida, and W. Schreiner, "A Distributed System for Solving Equational Constraints based on Lazy Narrowing Calculi," in *JSSST Workshop on Programming and Programming Languages (PPL '99)*, Atagawa, Japan, March 17–19, 1999 pp. 67–78.

[3] M. Marin, T. Ida, and T. Suzuki, "On Reducing the Search Space of Higher-Order Lazy Narrowing," in *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming, FLOPS '99, LNCS 1722*, (A. Middeldorp, T. Sato, eds.), Berlin and Heidelberg: Springer-Verlag, 1999 pp. 319–334.

[4] C. Prehofer, *Solving Higher-order Equations. From Logic to Programming*, Boston: Birkhäuser, 1998.

[5] T. Nipkow, "Functional Unification of Higher-order Patterns," in *Proceedings of the 8th IEEE Symposium on Logic in Computer Science*, Los Alamito, CA: IEEE Computer Society Press, 1993 pp. 64–74.

[6] H. Hong, "RISC-CLP(CF): Constraint Logic Programming over Complex Functions," in *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning, LPAR '94*, (F. Pfennig ed.), Berlin Heidelberg: Springer-Verlag, 1994 pp. 99–113.

[7] S. Wolfram, *The Mathematica Book*, 3rd ed., Champaign: Wolfram Media and Cambridge: Cambridge University Press, 1996.

■ Additional Material

Available at www.mathematica-journal.com.

cflp.tar.gz

See also www.score.is.tsukuba.ac.jp/software/CFLP

About the Authors

Doctor Mircea Marin is a JSPS postdoctoral fellow at the Institute of Information Sciences and Electronics, University of Tsukuba, Japan. His recent work includes the design and implementation of distributed systems for solving systems of constraints in a higher-order setting by using CORBA middleware, mathematical libraries, and computer algebra systems, such as *Mathematica*. For details on his recent work, see www.score.is.tsukuba.ac.jp/~mmarin.

Doctor Tetsuo Ida is a professor at the Institute of Information Sciences and Electronics, University of Tsukuba, Japan, where he leads a research group on symbolic computation. For many years he has worked on various aspects of symbolic computation, such as rewrite theories, parallel hashing algorithms, and Lisp Machine for symbolic computation. He is an editor of the *Journal of Symbolic Computation*. For details on his recent work, see www.score.is.tsukuba.ac.jp/~ida.

Doctor Wolfgang Schreiner is an assistant professor for the Research Institute for Symbolic Computation (RISC-Linz) at the Johannes Kepler University in Linz, Austria. He has worked for several years on parallel and distributed systems for symbolic and algebraic computation using various declarative languages, mathematical libraries, and computer algebra systems. For details on his recent work, see
www.risc.uni-linz.ac.at/people/schreine.

Runtime Behavior of Conversion Interpretation of Subtyping

Yasuhiko Minamide

Institute of Information Sciences and Electronics
University of Tsukuba
and
PRESTO, JST
minamide@is.tsukuba.ac.jp

Abstract. A programming language with subtyping can be translated into a language without subtyping by inserting conversion functions. Previous studies of this interpretation showed only the extensional correctness of the translation. We study runtime behavior of translated programs and show that this translation preserves execution time and stack space within a factor determined by the types in a program. Both the proofs on execution time and stack space are based on the method of logical relations where relations are extended with the factor of slowdown or increase of stack space.

1 Introduction

A programming language with subtyping can be translated into a language without subtyping by inserting conversion functions. Previous studies of this interpretation showed only the extensional correctness of the translation [3, 13]. In this paper, we study runtime behavior of the conversion interpretation of subtyping in call-by-value evaluation. We show that this translation preserves execution time and stack space within a factor determined by the types in a program, if subtyping relation is a partial order.

The translation of conversion interpretation changes the runtime behavior of programs in several respects. It inserts conversion functions and may increase the total number of closures allocated during execution. It translates tail-calls into non-tail-calls and, therefore, it may increase stack space usage. Although the translation causes these changes of runtime behavior, execution time and stack space usage are preserved asymptotically. This contrasts with type-directed unboxing of Leroy where both time and space complexity are not preserved [11].

Type systems with subtyping can be used to express information obtained by various program analyses such as control flow analyses [7]. One strategy of utilizing types obtained by program analysis is to adopt conversion interpretation of the subtyping. For example, it is possible to choose an optimized representation of values based on types and to insert conversion functions as type-directed unboxing of polymorphic languages [8]. In order to adopt this compilation method we need to show that the conversion interpretation is safe with respect to performance. The results in this paper ensure safety with respect to execution time and stack space.

Both the safety proofs on execution time and stack space are based on the method of logical relations. The method of logical relations has been used for correctness proofs of many type-directed program transformations [8, 14, 12] and was extended to prove time safety of unboxing by Minamide and Garrigue [11]. One motivation of this work is to show that the method of logical relations can be extended to prove safety with respect to stack space. The structure of the proof we obtained for stack space is almost the same as that for the execution time. This is because the operational semantics profiling stack space can be formalized in the same manner as the semantics profiling execution time. We believe this is the first proof concerning stack space based on the method of logical relations.

We believe that the conversion interpretation is also safe with respect to heap space. However, it seems that it is difficult to extend the method of logical relations for heap space. We would like to study safety with respect to heap space in future work.

This paper is organized as follows. We start with a review of conversion interpretation and runtime behavior of translated programs. In Section 3 we define the language we will use in the rest of the paper and formally introduce conversion interpretation. We prove that conversion interpretation preserves stack space and execution time in Section 4 and Section 5. Finally we review related work and presents the conclusions.

2 Review of Conversion Interpretation

We review conversion interpretation of subtyping and intuitively explain that the interpretation preserves stack space and execution time if the subtyping relation is a partial order. Since the subtyping relation is transitive and reflexive, the subtyping relation is a partial order if it contains no equivalent types.

The conversion interpretation is a translation from a language with subtyping into a simply typed language without subtyping. The idea is to insert a conversion function (or coercion) where the subsumption rule is used. If the following subsumption rule is used in the typing derivation,

$$\frac{\Gamma \vdash M : \tau \quad \tau \leq \sigma}{\Gamma \vdash M : \sigma}$$

the conversion function $\mathsf{coerce}_{\tau\leq\sigma}$ of type $\tau \to \sigma$ is inserted and we obtain the following term.

$$\mathsf{coerce}_{\tau\leq\sigma}(M)$$

Coercion $\mathsf{coerce}_{\tau\leq\sigma}$ is inductively defined on structure of types. If there are two base types bigint and int for integers where int is a subtype of bigint, we need to have a coercion primitive int2bigint of type int $\rightarrow$ bigint. A conversion function on function types is constructed as follows.

$$\lambda f.\lambda x.\mathsf{coerce}_{\tau_2 \leq \sigma_2}(f(\mathsf{coerce}_{\sigma_1 \leq \tau_1}(x)))$$

This is a coercion from $\tau_1 \rightarrow \tau_2$ to $\sigma_1 \rightarrow \sigma_2$.

We show that this interpretation of subtyping is safe with respect to execution time and stack space if the subtyping relation is a partial order. Intuitively, this holds because only a finite number of coercions can be applied to any value. If a subtyping relation is not a partial order, i.e., there exist two types τ and σ such that $\tau \leq \sigma$ and $\sigma \leq \tau$, we can easily construct counter examples for both execution time and stack space. A counter example for execution time is the following translation of term M of type τ,

$$\mathsf{coerce}_{\sigma \leq \tau}(\mathsf{coerce}_{\tau \leq \sigma}(\ldots(\mathsf{coerce}_{\sigma \leq \tau}(\mathsf{coerce}_{\tau \leq \sigma}(M)))))$$

where $\tau \leq \sigma$ and $\sigma \leq \tau$. The execution time to evaluate the coercions in the translation depends on the number of the coercions and cannot be bounded by a constant. It may be possible to avoid this silly translation, but it will be difficult to avoid this problem in general if we have equivalent types.

The conversion interpretation translates tail-call applications into non-tail-call applications. Let us consider the following translation of application $x\,y$.

$$\mathsf{coerce}_{\tau \leq \sigma}(x\,y)$$

Even if $x\,y$ is originally at a tail-call position, after translation it is not at a tail-call position. Therefore, it is not straightforward to show the conversion interpretation preserves stack space asymptotically. In fact, if we have equivalent types, we can demonstrate a counter example. Let us consider the following program where types A and B are equivalent.

```
fun f (0, x : A) = x                (* f: int * A -> A *)
  | f (n, x : A) = g (n-1, x)
and g (n, x : A) = f (n, x) : B     (* g: int * A -> B *)
```

We have a type annotation `f (n, x) : B` in the body of g and thus g has type `A -> B`. This program contains only tail-calls, and thus requires only constant stack space. By inserting conversion functions we obtain the following program:

```
fun f (0, x : A) = x
  | f (n, x : A) = B2A (g (n-1, x))
and g (n, x : A) = A2B (f (n, x))
```

where A2B and B2A are coercions between A and B. For this program, evaluation of `f n` requires stack space proportional to n since both the applications of f and g are not tail-calls.

In order to preserve time and stack space complexity, it is essential that the subtyping relation is a partial order. This ensures that there is no infinite subtyping chain of types if we consider only structural subtyping. Thus only a finite number of conversions can be applied to any value if the subtyping relation is a partial order.

3 Language and Conversion Interpretation

In this section we introduce a call-by-value functional language with subtyping and its conversion interpretation. We consider a call-by-value functional language with the following syntax.

$$\begin{array}{l} V ::= x \mid \bar{i} \mid \underline{i} \mid \lambda x.M \mid \mathtt{fix}^n\, x.\lambda y.M \\ M ::= V \mid M\,M \mid \mathtt{let}\ x = M\ \mathtt{in}\ M \end{array}$$

There are two families of integers: $\bar{i}$ and $\underline{i}$ are integer values of types bigint and int respectively. The language includes bounded recursive functions where $\mathtt{fix}^n\, x.\lambda y.M$ is *expanded at most n times* [4]. Any closed program with usual recursive functions can be simulated by bounded recursive functions.

For this language we consider a simple type system extended with subtyping. The types of the language are defined as follows.

$$\tau ::= \mathsf{bigint} \mid \mathsf{int} \mid \tau \to \tau$$

We consider two base types bigint and int where int is a subtype of bigint. A *metavariable σ is also used to denote a type.* The subtyping relation $\tau_1 \leq \tau_2$ is given by the following three rules.

$$\tau \leq \tau \qquad \mathsf{int} \leq \mathsf{bigint} \qquad \frac{\sigma_1 \leq \tau_1 \quad \tau_2 \leq \sigma_2}{\tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2}$$

The rule for transitivity is not included here because it can be derived from the other rules for this subtyping relation. We write $\tau < \sigma$ if $\tau \leq \sigma$ and $\tau \not\equiv \sigma$. It is clear that the subtyping relation is a partial order. The typing judgment has the following form:

$$\Gamma \vdash M{:}\tau$$

where Γ is a type assignment of the form $x_1{:}\tau_1, \ldots, x_n{:}\tau_n$. The rules of the type system are defined as follows.

$$\Gamma \vdash \bar{i}{:}\mathsf{bigint} \qquad\qquad \Gamma \vdash \underline{i}{:}\mathsf{int}$$

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x{:}\tau} \qquad\qquad \frac{\Gamma \vdash M_1{:}\tau_1 \to \tau_2 \quad \Gamma \vdash M_2{:}\tau_1}{\Gamma \vdash M_1 M_2{:}\tau_2}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash M{:}\tau_2}{\Gamma \vdash \lambda x.M{:}\tau_1 \to \tau_2} \qquad\qquad \frac{\Gamma \vdash M{:}\sigma \quad \sigma \leq \tau}{\Gamma \vdash M{:}\tau}$$

$$\frac{\Gamma, y{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash M{:}\tau_2}{\Gamma \vdash \mathtt{fix}^n\, y.\lambda x.M{:}\tau_1 \to \tau_2} \qquad\qquad \frac{\Gamma \vdash M_1{:}\tau_1 \quad \Gamma, x{:}\tau_1 \vdash M_2{:}\tau}{\Gamma \vdash \mathtt{let}\ x = M_1\ \mathtt{in}\ M_2{:}\tau}$$

Note that `let`-expressions do not introduce polymorphic types. They are used to simplify definition of coercions.

We consider a standard natural semantics for this language. A judgment has the following form: $M \Downarrow V$. The rules are given as follows.

$$V \Downarrow V \qquad \frac{M_1 \Downarrow V_1 \quad M_2[V_1/x] \Downarrow V}{\mathtt{let}\ x = M_1\ \mathtt{in}\ M_2 \Downarrow V}$$

$$\frac{M_1 \Downarrow \lambda x.M \quad M_2 \Downarrow V_2 \quad M[V_2/x] \Downarrow V}{M_1 M_2 \Downarrow V}$$

$$\frac{M_1 \Downarrow \mathtt{fix}^{k+1}\, y.\lambda x.M \quad M_2 \Downarrow V_2 \quad M[\mathtt{fix}^k\, y.\lambda x.M/y][V_2/x] \Downarrow V}{M_1 M_2 \Downarrow V}$$

When the recursive function $\mathtt{fix}^{k+1} y.\lambda x.M$ is applied, the bound of the recursive function is decremented.

To formalize the conversion interpretation we need to introduce a target language without subtyping that includes a coercion primitive. We consider the following target language. The only extension is the coercion primitive int2bigint from int into bigint.

$$W ::= x \mid \underline{i} \mid \bar{i} \mid \lambda x.N \mid \mathtt{fix}^n\, x.\lambda y.N$$

$$N ::= W \mid N\ N \mid \mathtt{let}\ x = N\ \mathtt{in}\ N \mid \mathsf{int2bigint}(N)$$

The operational semantics and type system of the language are almost the same as those of the source language. The rule of subsumption is excluded from the type system. The typing rule and evaluation of coercion are defined as follows.

$$\frac{N \Downarrow \underline{i}}{\mathsf{int2bigint}(N) \Downarrow \bar{i}} \qquad \frac{\Gamma \vdash N{:}\mathsf{int}}{\Gamma \vdash \mathsf{int2bigint}(N){:}\mathsf{bigint}}$$

The conversion interpretation is defined inductively on structure of the typing derivation of a program: the translation $\mathcal{C}[\![\Gamma \vdash M{:}\tau]\!]$ below gives a term of the target language.

$$\begin{aligned}
\mathcal{C}[\![\Gamma \vdash x{:}\tau]\!] &= x\\
\mathcal{C}[\![\Gamma \vdash \lambda x.M{:}\tau_1 \to \tau_2]\!] &= \lambda x.\mathcal{C}[\![\Gamma, x{:}\tau_1 \vdash M{:}\tau_2]\!]\\
\mathcal{C}[\![\Gamma \vdash \mathtt{fix}^n\, y.\lambda x.M{:}\tau_1 \to \tau_2]\!] &= \mathtt{fix}^n\, y.\lambda x.\mathcal{C}[\![\Gamma, y{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash M{:}\tau_2]\!]\\
\mathcal{C}[\![\Gamma \vdash M_1 M_2{:}\tau_2]\!] &= \mathcal{C}[\![\Gamma \vdash M_1{:}\tau_1 \to \tau_2]\!]\mathcal{C}[\![\Gamma \vdash M_2{:}\tau_1]\!]\\
\mathcal{C}[\![\Gamma \vdash M{:}\tau]\!] &= \mathsf{coerce}_{\sigma \le \tau}(\mathcal{C}[\![\Gamma \vdash M{:}\sigma]\!])\\
\mathcal{C}[\![\Gamma \vdash \mathtt{let}\ x = M_1\ \mathtt{in}\ M_2{:}\tau_2]\!] &= \mathtt{let}\ x = \mathcal{C}[\![\Gamma \vdash M_1{:}\tau_1]\!]\ \mathtt{in}\ \mathcal{C}[\![\Gamma \vdash M_2{:}\tau_2]\!]
\end{aligned}$$

The coercion used in the translation is defined inductively on structure of derivation of subtyping as follows [1].

$$\begin{aligned}
\mathsf{coerce}_{\tau\leq\tau}(M) &= M \\
\mathsf{coerce}_{\mathsf{int}\leq\mathsf{bigint}}(M) &= \mathsf{int2bigint}(M) \\
\mathsf{coerce}_{\tau_1\to\tau_2\leq\sigma_1\to\sigma_2}(M) &= \mathtt{let}\ x = M\ \mathtt{in}\ \lambda y.\mathsf{coerce}_{\tau_2\leq\sigma_2}(x(\mathsf{coerce}_{\sigma_1\leq\tau_1}(y)))
\end{aligned}$$

Note that $\mathsf{coerce}_{\tau\leq\tau}(M)$ must be not only extensionally equivalent to M, but also intensionally equivalent to M. If we adopt $(\lambda x.x)\,M$ for $\mathsf{coerce}_{\tau\leq\tau}(M)$, we have the same problem when we have equivalent types, and thus execution time and stack space are not preserved.

We define two measures, $\lfloor\tau\rfloor$ and $\lceil\tau\rceil$, of types as follows.

$$\begin{array}{rclcrcl}
\lfloor\mathsf{int}\rfloor &=& 0 & \qquad & \lceil\mathsf{int}\rceil &=& 1 \\
\lfloor\mathsf{bigint}\rfloor &=& 1 & & \lceil\mathsf{bigint}\rceil &=& 0 \\
\lfloor\tau_1\to\tau_2\rfloor &=& \lceil\tau_1\rceil + \lfloor\tau_2\rfloor & & \lceil\tau_1\to\tau_2\rceil &=& \lfloor\tau_1\rfloor + \lceil\tau_2\rceil
\end{array}$$

It is clear that $\sigma < \tau$ implies $\lfloor\sigma\rfloor < \lfloor\tau\rfloor$ and $\lceil\sigma\rceil > \lceil\tau\rceil$. Since $\lfloor\tau\rfloor$ and $\lceil\tau\rceil$ are non-negative integers, we also obtain the following properties.

$$\begin{array}{lclcl}
\tau_n < \ldots < \tau_1 < \tau_0 & \Rightarrow & \lfloor\tau_n\rfloor < \ldots < \lfloor\tau_1\rfloor < \lfloor\tau_0\rfloor & \Rightarrow & n \leq \lfloor\tau_0\rfloor \\
\tau_0 < \tau_1 < \ldots < \tau_n & \Rightarrow & \lceil\tau_0\rceil > \lceil\tau_1\rceil > \ldots > \lceil\tau_n\rceil & \Rightarrow & n \leq \lceil\tau_0\rceil
\end{array}$$

From the property we can estimate the maximum number of conversions applied a value of τ_0. In the following program, we know that $n \leq \lceil\tau_0\rceil$ by the property.

$$\mathsf{coerce}_{\tau_{n-1}\leq\tau_n}(\ldots(\mathsf{coerce}_{\tau_0\leq\tau_1}(V))$$

Intuitively, this is the property that ensures that conversion interpretation preserves execution time and stack within a factor determined by types in a program.

4 Preservation of Stack Space

We show that coercion interpretation of subtyping preserves stack space within a factor determined by types occurring in a program. Strictly speaking, the factor is determined by the types occurring in the typing derivation used in translation of a program. We prove this property by the method of logical relations.

First we extend the operational semantics to profile stack space usage. The extended judgment has the form $M \Downarrow^n V$ where n models the size of stack space required to evaluate M to V. The following are the extended rules.

$$V \Downarrow^1 V \qquad \frac{M_1 \Downarrow^m V_1 \quad M_2[V_1/x] \Downarrow^n V}{\mathtt{let}\ x = M_1\ \mathtt{in}\ M_2 \Downarrow^{\max(m+1,n)} V}$$

[1] We assume that $\tau_1 \to \tau_2 \leq \tau_1 \to \tau_2$ is not derived from $\tau_1 \leq \tau_1$ and $\tau_2 \leq \tau_2$, but from the axiom.

$$\frac{M_1 \Downarrow^l \lambda x.M \quad M_2 \Downarrow^m V_2 \quad M[V_2/x] \Downarrow^n V}{M_1 M_2 \Downarrow^{\mathsf{max}(l+1,m+1,n)} V}$$

$$\frac{M_1 \Downarrow^l \mathtt{fix}^{k+1}\, y.\lambda x.M \quad M_2 \Downarrow^m V_2 \quad M[\mathtt{fix}^k\, y.\lambda x.M/y][V_2/x] \Downarrow^n V}{M_1 M_2 \Downarrow^{\mathsf{max}(l+1,m+1,n)} V}$$

This semantics is considered to model evaluation by an interpreter: $M \Downarrow^n V$ means that a standard interpreter requires n stack frames to evaluate M to V. In the rule of application, evaluation of M_1 and M_2 are considered as non-tail-calls and evaluation of the body of the function is considered as a tail-call. This is the reason that the number of stack frames used to evaluate the application is $\mathsf{max}(l+1, m+1, n)$.

This semantics and the correspondence to a semantics modeling evaluation after compilation is discussed in [10]: the ratio to the stack space used by compiled executable code is bounded by the size of a program.

By the rule for values, a value V is evaluated to itself with 1 stack frame. Instead, you can choose $V \Downarrow^0 V$ as the rule for values. This choice does not matter much because the difference caused by the choice is always only 1 stack frame. We have chosen our rule to simplify our proofs.

We write $e \Downarrow^n$ if $e \Downarrow^n v$ for some v and $e \Downarrow^{\leq n}$ if $e \Downarrow^m$ for some $m \leq n$.

The main result of this section is that the conversion interpretation preserves stack space within a factor determined by the sizes of types appearing in a program.

Theorem 1. *Let $C[\![\emptyset \vdash M{:}\tau]\!] = N$ and let C be an integer such that $C > \lfloor\sigma\rfloor + 3$ for all σ appearing in the derivation of $\emptyset \vdash M{:}\tau$. If $M \Downarrow^n V$ then $N \Downarrow^{\leq Cn} W$ for some W.*

Let us consider the following translation where the type of $\lambda x.\underline{1}$ is obtained by subsumption for $\mathsf{int} \to \mathsf{int} \leq \mathsf{int} \to \mathsf{bigint}$.

$$\mathcal{C}[\![(\lambda x.\underline{1})\underline{2}]\!] = (\mathtt{let}\ \ y = \lambda x.\underline{1}\ \mathtt{in}\ \ \lambda z.\mathsf{int2bigint}(y\, z))\underline{2}$$

The source program is evaluated with 2 stack frames.

$$(\lambda x.\underline{1})\underline{2} \Downarrow^2 \underline{1}$$

On the other hand, the translation is evaluated with 4 stack frames.

$$\frac{\lambda x.\underline{1} \Downarrow^1 \lambda x.\underline{1} \qquad \dfrac{V \Downarrow^1 V \quad \underline{2} \Downarrow^1 \underline{2} \quad \dfrac{(\lambda x.\underline{1})\ \underline{2} \Downarrow^2 \underline{1}}{\mathsf{int2bigint}((\lambda x.\underline{1})\ \underline{2}) \Downarrow^3 \overline{1}}}{(\lambda z.\mathsf{int2bigint}((\lambda x.\underline{1})\ \underline{z}))\ \underline{2} \Downarrow^3 \overline{1}}}{(\mathtt{let}\ \ y = \lambda x.\underline{1}\ \mathtt{in}\ \ \lambda z.\mathsf{int2bigint}(y\, z))\underline{2} \Downarrow^3 \overline{1}}$$

where $V \equiv \lambda z.\mathsf{int2bigint}((\lambda x.\underline{1})\ \underline{z})$. In this case, the factor of increase is 3/2.

We prove the main theorem by the method of logical relations. Before defining the logical relations we define the auxiliary relation $V_1 V_2 \downarrow^n V$ defined as follows.

$$\frac{M[V_2/x] \Downarrow^n V}{(\lambda x.M)V_2 \downarrow^n V} \qquad \frac{M[V_2/x][\mathtt{fix}^k\, y.\lambda x.M/y] \Downarrow^n V}{(\mathtt{fix}^{k+1}\, y.\lambda x.M)V_2 \downarrow^n V}$$

By using this relation we can combine the two rules for the evaluation of the application into the following rule.

$$\frac{M_1 \Downarrow^{l} V_1 \quad M_2 \Downarrow^{m} V_2 \quad V_1 V_2 \downarrow^{n} V}{M_1 M_2 \Downarrow^{\mathsf{max}(l+1, m+1, n)} V}$$

This reformulation simplifies the definition of the logical relations and our proof. We define logical relations $V \approx^{C}_{\tau} W$ indexed by a type τ and a positive integer C as follows.

$$\begin{array}{cl}
\underline{i} \approx^{C}_{\mathsf{int}} \underline{i} & \\
V \approx^{C}_{\mathsf{bigint}} \bar{i} & V = \underline{i} \text{ or } V = \bar{i} \\
V \approx^{C}_{\tau_1 \to \tau_2} W & \begin{cases} \text{for all } V_1 \approx^{C}_{\tau_1} W_1, \text{ if } V V_1 \downarrow^{n+1} V_2 \\ \text{then } W W_1 \downarrow^{\leq Cn + \lfloor \tau_2 \rfloor + 3} W_2 \text{ and } V_2 \approx^{C}_{\tau_2} W_2 \end{cases}
\end{array}$$

We implicitly assume that V and W have type τ for $V \approx^{C}_{\tau} W$. The parameter C corresponds to the factor of increase of stack space usage. Note that the increase of stack space usage depends on only the range type τ_2 of a function type $\tau_1 \to \tau_2$. This is explained by checking the following translation of a function f of type $\tau_1 \to \tau_2$ [2].

$$\mathsf{coerce}_{\tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2}(f) \equiv \lambda y.\mathsf{coerce}_{\tau_2 \leq \sigma_2}(f\ \mathsf{coerce}_{\sigma_1 \leq \tau_1}(y))$$

In this translation, only the coercion $\mathsf{coerce}_{\tau_2 \leq \sigma_2}$ causes increase of stack space usage.

We first show that a conversion from τ to σ behaves well with respect to the logical relations.

Lemma 1. *If $\tau < \sigma$ and $V \approx^{C}_{\tau} W$ then $\mathsf{coerce}_{\tau \leq \sigma}(W) \Downarrow^{2} W'$ and $V \approx^{C}_{\sigma} W'$ for some W'.*

Proof. By induction on derivation of $\tau < \sigma$.

Case: $\mathsf{int} \leq \mathsf{bigint}$. By the definition of $V \approx^{C}_{\mathsf{int}} W$, both V and W must be $\underline{i}$ for some i. Since $\mathsf{coerce}_{\mathsf{int} \leq \mathsf{bigint}}(\underline{i}) = \mathsf{int2bigint}(\underline{i})$, we have $\mathsf{int2bigint}(\underline{i}) \Downarrow^{2} \bar{i}$ and $\underline{i} \approx^{C}_{\mathsf{bigint}} \bar{i}$.

Case: $\tau \equiv \tau_1 \to \tau_2$ and $\sigma \equiv \sigma_1 \to \sigma_2$ where $\sigma_1 \leq \tau_1$ and $\tau_2 \leq \sigma_2$. There are two subcases: $\tau_2 < \sigma_2$ and $\tau_2 \equiv \sigma_2$. We show the former case here. The proof the latter case is similar.

$$\mathsf{coerce}_{\tau \leq \sigma}(W) \Downarrow^{2} \lambda y.\mathsf{coerce}_{\tau_2 \leq \sigma_2}(W(\mathsf{coerce}_{\sigma_1 \leq \tau_1}(y)))$$

Let $V_0 \approx^{C}_{\sigma_1} W_0$ and $V V_0 \downarrow^{m+1} V_2$. By induction hypothesis,

$$\mathsf{coerce}_{\sigma_1 \leq \tau_1}(W_0) \Downarrow^{\leq 2} W_1$$

and $V_0 \approx^{C}_{\tau_1} W_1$ for some W_1. By definition of the logical relations

$$W W_1 \downarrow^{\leq Cm + \lfloor \tau_2 \rfloor + 3} W_3$$

[2] Strictly speaking, it is `let` $x = f$ `in` $\lambda y.\mathsf{coerce}_{\tau_2 \leq \sigma_2}(x\ \mathsf{coerce}_{\sigma_1 \leq \tau_1}(y))$.

and $V_2 \approx^C_{\tau_2} W_3$ for some W_3. Then we obtain the following evaluation.

$$\frac{W \Downarrow^1 W \quad \mathsf{coerce}_{\sigma_1 \leq \tau_1}(W_0) \Downarrow^{\leq 2} W_1 \quad W W_1 \downarrow^{\leq Cm+\lfloor\tau_2\rfloor+3} W_3}{W(\mathsf{coerce}_{\sigma_1 \leq \tau_1}(W_0)) \Downarrow^{\leq \mathsf{max}(2,3,Cm+\lfloor\tau_2\rfloor+3)} W_3}$$

where $\mathsf{max}(2, 3, Cm + \lfloor\tau_2\rfloor + 3) = Cm + \lfloor\tau_2\rfloor + 3$.
By induction hypothesis,

$$\mathsf{coerce}_{\tau_2 \leq \sigma_2}(W_3) \Downarrow^2 W_2$$

and $V_2 \approx^C_{\sigma_2} W_2$ for some W_2. Then

$$\mathsf{coerce}_{\tau_2 \leq \sigma_2}(W(\mathsf{coerce}_{\sigma_1 \leq \tau_1}(W_0))) \Downarrow^{\leq Cm+\lfloor\tau_2\rfloor+4} W_2$$

where $Cm + \lfloor\tau_2\rfloor + 4 \leq Cm + \lfloor\sigma_2\rfloor + 3$

□

The next lemma indicates that we can choose a constant C such that the evaluation of a source program and its translation are related by C. For ρ and ρ' two environments with the same domain, $\rho \approx^C_\Gamma \rho'$ means that they are pointwise related. The main theorem is obtained by restricting this lemma to $\Gamma = \emptyset$ and taking C such that $C > \lfloor\sigma\rfloor + 3$ for all σ appearing in the typing derivation.

Lemma 2. *Let C be an integer such that $C > \lfloor\sigma\rfloor$ for all σ appearing in the derivation of $\Gamma \vdash M{:}\tau$. Let $\mathcal{C}[\![\Gamma \vdash M{:}\tau]\!] = N$ and $\rho \approx^C_\Gamma \rho'$.*
If $\rho(M) \Downarrow^{n+1} V$ then $\rho'(N) \Downarrow^{\leq Cn+\lfloor\tau\rfloor+3} W$ and $V \approx^C_\tau W$ for some W.

Proof. By lexicographic induction on derivation of $\Gamma \vdash M{:}\tau$ and the sum of bounds of recursive functions in M.

Case: $\Gamma \vdash M{:}\sigma$ is derived from $\Gamma \vdash M{:}\tau$ and $\tau \leq \sigma$. We assume $\tau < \sigma$. The case of $\tau \equiv \sigma$ is trivial. By definition, N must be $\mathsf{coerce}_{\tau\leq\sigma}(N_0)$ for some N_0 and $\mathcal{C}[\![\Gamma \vdash M : \tau]\!] = N_0$. By induction hypothesis,

$$\rho'(N_0) \Downarrow^{\leq Cn+\lfloor\tau\rfloor+3} W_0$$

and $V \approx^C_\tau W_0$ for some W_0. By Lemma 1,

$$\rho'(\mathsf{coerce}_{\tau\leq\sigma}(N_0)) \Downarrow^{\leq Cn+\lfloor\tau\rfloor+3+1} W$$

and $V \approx^C_\sigma W$ for some W. The proof of this case completes since $Cn + \lfloor\tau\rfloor + 3 + 1 \leq Cn + \lfloor\sigma\rfloor + 3$.

Case: $\Gamma \vdash M_1M_2{:}\tau_2$ is derived from $\Gamma \vdash M_1{:}\tau_1 \to \tau_2$ and $\Gamma \vdash M_2{:}\tau_1$. By the definition of the translation $N \equiv N_1N_2$ for some N_1 and N_2.
$\rho(M_1M_2) \Downarrow^{k+1} V$ is derived from $\rho(M_1) \Downarrow^l V_1$ and $\rho(M_2) \Downarrow^m V_2$ and $V_1V_2 \downarrow^n V$ where $l \leq k$, $m \leq k$ and $n \leq k+1$. By induction hypothesis for M_1,

$$\rho'(N_1) \Downarrow^{C(l-1)+\lfloor\tau_1\to\tau_2\rfloor+3} W_1$$

and $V_1 \approx^C_{\tau_1 \to \tau_2} W_1$ for some W_1. Then we have $\rho'(N_1) \Downarrow^{\leq Cl+2} W_1$ because $\lfloor \tau_1 \to \tau_2 \rfloor + 1 \leq C$. By induction hypothesis for M_2,

$$\rho'(N_2) \Downarrow^{C(m-1)+\lfloor \tau_1 \rfloor+3} W_2$$

and $V_2 \approx^C_{\tau_1} W_2$ for some W_2. Then we also have $\rho'(N_2) \Downarrow^{\leq Cm+2} W_2$ because $\lfloor \tau_1 \rfloor + 1 \leq C$. By definition of the logical relations

$$W_1 W_2 \downarrow^{\leq C(n-1)+\lfloor \tau_2 \rfloor+3} W$$

and $V \approx^C_{\tau_2} W$ for some W. We have the following inequality.

$$\max(Cl+2+1, Cm+2+1, C(n-1)+\lfloor \tau_2 \rfloor+3) \leq Ck + \lfloor \tau_2 \rfloor + 3$$

Hence,

$$\rho'(N_1 N_2) \Downarrow^{\leq Ck+\lfloor \tau_2 \rfloor+3} W$$

Case: $\Gamma \vdash \mathtt{fix}^{a+1}\, y.\lambda x.M{:}\tau_1 \to \tau_2$ is derived from $\Gamma, y{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash M{:}\tau_2$. By definition, $\mathcal{C}[\![\Gamma, y{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash M{:}\tau_1]\!] = N$ for some N. We have the following evaluation.

$$\rho(\mathtt{fix}^{a+1}\, y.\lambda x.M) \Downarrow^1 \rho(\mathtt{fix}^{a+1}\, y.\lambda x.M)$$

$$\rho'(\mathtt{fix}^{a+1}\, y.\lambda x.N) \Downarrow^1 \rho'(\mathtt{fix}^{a+1}\, y.\lambda x.N)$$

Let $V \approx^C_{\tau_1} W$ and $\rho(M)[V/x][\rho(\mathtt{fix}^a\, y.\lambda x.M)/y] \Downarrow^{n+1} V'$. By induction hypothesis,

$$\rho(\mathtt{fix}^a\, y.\lambda x.M) \approx^C_{\tau_1 \to \tau_2} \rho'(\mathtt{fix}^a\, y.\lambda x.N)$$

Let $\rho_0 = \rho[V/x][\rho(\mathtt{fix}^a\, y.\lambda x.M)/y]$ and $\rho'_0 = \rho'[W/x][\rho'(\mathtt{fix}^a\, y.\lambda x.N)/y]$. We have $\rho_0 \approx^C_{\Gamma, y:\tau_1 \to \tau_2, x:\tau_1} \rho'_0$. By induction hypothesis,

$$\rho'_0(N) \Downarrow^{Cn+\lfloor \tau_2 \rfloor+3} W'$$

and $V' \approx^C_{\tau_2} W'$. Hence, $\rho(\mathtt{fix}^{a+1}\, y.\lambda x.M) \approx^C_{\tau_1 \to \tau_2} \rho'(\mathtt{fix}^{a+1}\, y.\lambda x.N)$.

□

5 Preservation of Execution Time

We introduce the operational semantics profiling execution time and outline the proof that the coercion interpretation of subtyping is also safe with respect to execution time. The operational semantics for execution time is a simple extension of the standard semantics as that for stack space. As the previous section, we first extend judgment of operational semantics to the following form:

$$M \Downarrow^n V$$

where n represents execution time to evaluate M to V. For the rule of application we use an auxiliary relation: $V_1 V_2 \downarrow^n V$ as before. The rules are given as follows.

$$V \Downarrow^1 V \qquad \frac{M_1 \Downarrow^m V_1 \quad M_2[V_1/x] \Downarrow^n V}{\texttt{let } x = M_1 \texttt{ in } M_2 \Downarrow^{m+n+1} V}$$

$$\frac{M_1 \Downarrow^l V_1 \quad M_2 \Downarrow^m V_2 \quad V_1 V_2 \downarrow^n V}{M_1 M_2 \Downarrow^{l+m+n+1} V}$$

$$\frac{M[V_2/x] \Downarrow^n V}{(\lambda x.M)V_2 \downarrow^n V} \qquad \frac{M[V_2/x][\texttt{fix}^k\, y.\lambda x.M/y] \Downarrow^n V}{(\texttt{fix}^{k+1}\, y.\lambda x.M)V_2 \downarrow^n V}$$

All the rules are a straightforward extension of the standard rules.

Then it is shown that the conversion interpretation preserves execution time within a factor determined by the types appearing in a program.

Theorem 2. *Let $\mathcal{C}[\![\emptyset \vdash M : \tau]\!] = N$ and let C be an integer such that $C > 7\lfloor\sigma\rfloor$ for all σ appearing in the derivation of $\emptyset \vdash M : \tau$. If $M \Downarrow^n V$ then $N \Downarrow^{\leq Cn} W$ for some W.*

The factor of slowdown $7\lfloor\sigma\rfloor$ is bigger than the factor of increase of stack space $\lfloor\sigma\rfloor$. To prove this theorem, w use the method of logical relations which are indexed by a slowdown factor as well as a type. The relations $V \approx^C_\tau W$ are defined as follows.

$$\begin{array}{ll} \underline{i} \approx^C_{\mathsf{int}} \underline{i} & \\ V \approx^C_{\mathsf{bigint}} \bar{i} & V = \underline{i} \text{ or } V = \bar{i} \\ V \approx^C_{\tau_1 \to \tau_2} W & \left\{ \begin{array}{l} \text{for all } V_1 \approx^C_{\tau_1} W_1, \text{ if } VV_1 \downarrow^{n+1} V_2 \\ \text{then } WW_1 \downarrow^{Cn+7\lfloor\tau_1\to\tau_2\rfloor+1} W_2 \text{ and } V_2 \approx^C_{\tau_2} W_2 \end{array} \right. \end{array}$$

The important difference from the relations for stack space is that slowdown of the applications depends on the domain type τ_1 as well as the range type τ_2 of a function type $\tau_1 \to \tau_2$.

With this definition, the main theorem is proved in the same manner as the proof for stack space. It is shown that a conversion function behaves well with respect to the logical relations as before. Then the generalization of the main theorem is proved by induction on the derivation of the conversion interpretation of a program.

6 Conclusions and Related Work

We have shown that conversion interpretation of subtyping preserves execution time and stack space within a factor determined by the types in a program if the subtyping relation is a partial order. Type-directed unboxing of Leroy is a translation similar to conversion interpretation of subtyping, but it does not preserve execution time and stack space. This is because conversions of equivalent types appear in type-directed unboxing.

We have considered only a very simple type system which does not include product types and recursive types. We believe the results in this paper can be easily extended for product types. However, our results cannot be extended for recursive types. If we consider recursive types, cost of one application of coercion cannot be bounded by a constant as Leroy discussed in his work on type-directed unboxing for polymorphic languages [8]. Thus the conversion interpretation does not preserve execution time nor stack space usage in the presence of subtyping on recursive types.

We have shown that the conversion interpretation is safe with respect to time and stack space by the method of logical relations. We believe the conversion interpretation is also safe with respect to heap space, but it will be difficult to adopt the same method for heap space. We have no idea how to formalize logical relations for heap space because the semantics profiling heap space is much more complicated than those for time and stack space.

In the rest of this section I review other proof methods to show safety of program transformations with respect to performance.

David Sands studied time safety of transformations for call-by-name languages [16, 15]. In his study he extended applicative bisimulation and its context lemma to account execution time. Applicative bisimulation with the context lemma greatly simplifies safety proofs of many program transformations. As with the method of logical relations, it will be difficult to extend this method if we consider heap space or various extensions of languages.

Another approach is to analyze states of evaluation more directly, where proofs are often based on induction on length of evaluation. Blelloch and Greiner showed that an implementation of NESL based on an abstract machine preserves execution time and space within a constant factor based on this approach [2]. Minamide showed that the CPS transformation preserves space within a constant factor [9]. Gustavsson and Sands developed a theory of a space improvement relation for a call-by-need programming language [5, 6]. They clarified their proofs by considering evaluation of programs with holes based on a context calculus [17]. Bakewell and Runciman proposed an operational model for lazy functional programming languages based on graph rewriting [1]. As a proof method for the model they considered an extension of bisimulation.

Acknowledgments

This work is partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Encouragement of Young Scientists of Japan, No. 13780193, 2001.

References

[1] A. Bakewell and C. Runciman. A model for comparing the space usage of lazy evaluators. In *Proceedings of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 151–162, 2000.

[2] G. E. Blelloch and J. Greiner. A provably time and space efficient implementation of NESL. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, 1996.

[3] V. Breazu-Tannen, C. A. Gunter, and A. Scedrov. Computing with coercions. In *Proceedings of the 1990 ACM Conference on LISP and Functional programming*, pages 44–60, 1990.

[4] C. A. Gunter. *Semantics of Programming Languages*, chapter 4. The MIT Press, 1992.

[5] J. Gustavsson and D. Sands. A foundation for space-safe transformations of call-by-need programs. In *Proceedings of the Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS99)*, volume 26 of *ENTCS*, 1999.

[6] J. Gustavsson and D. Sands. Possibilities and limitations of call-by-need space improvement. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 265–276, 2001.

[7] N. Heintze. Control-flow analysis and type systems. In *Proceedings of the 1995 International Static Analysis Symposium*, volume 983 of *LNCS*, pages 189–206, 1995.

[8] X. Leroy. Unboxed objects and polymorphic typing. In *the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188, 1992.

[9] Y. Minamide. A space-profiling semantics of call-by-value lambda calculus and the CPS transformation. In *Proceedings of the Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS99)*, volume 26 of *ENTCS*, 1999.

[10] Y. Minamide. A new criterion for safe program transformations. In *Proceedings of the Forth International Workshop on Higher Order Operational Techniques in Semantics (HOOTS)*, volume 41(3) of *ENTCS*, Montreal, 2000.

[11] Y. Minamide and J. Garrigue. On the runtime complexity of type-directed unboxing. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional programming*, pages 1–12, 1998.

[12] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Proceeding of the ACM Symposium on Principles of Programming Languages*, pages 271–283, 1996.

[13] J. C. Mitchell. *Foundations for Programming Languages*, chapter 10. The MIT Press, 1996.

[14] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transaction on Programming Languages and Systems*, 17(6):844–895, 1995.

[15] D. Sands. A naive time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4):495–541, 1995.

[16] D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, 167(1&2):193–233, 1996.

[17] D. Sands. Computing with contexts: A simple approach. In *Proceedings of the Second Workshop on Higher-Order Operational Techniques in Semantics (HOOTS II)*, volume 10 of *ENTCS*, 1998.

Complete Selection Functions for a Lazy Conditional Narrowing Calculus

Aart Middeldorp

Institute of Information Sciences and Electronics
University of Tsukuba, Tsukuba 305-8573, Japan
ami@is.tsukuba.ac.jp

Taro Suzuki

Department of Computer Software
University of Aizu, Aizu-Wakamatsu 965-8580, Japan
taro@u-aizu.ac.jp

Mohamed Hamada

Department of Computer Software
University of Aizu, Aizu-Wakamatsu 965-8580, Japan
hamada@u-aizu.ac.jp

Abstract

In this paper we extend the lazy narrowing calculus LNC of Middeldorp, Okui, and Ida [26] to conditional rewrite systems. The resulting lazy conditional narrowing calculus LCNC is highly non-deterministic. We investigate for which classes of conditional rewrite systems the completeness of LCNC is ensured. In order to improve the efficiency of the calculus, we pay special attention to the removal of non-determinism due to the selection of equations in goals by fixing a selection strategy.

1 Introduction

Narrowing (Fay [7], Hullot [19]) was originally invented as a general method for solving unification problems in equational theories that are presented by confluent term rewriting systems (TRSs for short). More recently, narrowing was proposed as the computational mechanism of several functional-logic programming languages (Hanus [16]) and several new completeness results concerning the completeness of various narrowing strategies and calculi have been obtained in the past few years. Here completeness means that for every solution to a given goal a solution that is at least as general is computed by the narrowing strategy. Since narrowing is a complicated operation, numerous calculi consisting of a small number of more elementary inference rules that simulate narrowing have been proposed (e.g. [8, 18, 23, 30, 17, 26, 10, 29, 22, 11]).

Completeness issues for the lazy narrowing calculus LNC—which is based on the calculus TRANS of Hölldobler [18]—have been extensively studied in [26] and [25]. In [26] Middeldorp *et al.* prove that LNC is *strongly* complete whenever *basic* narrowing (Hullot [19]) is complete. Strong completeness means that the choice of the equation in goals can be made don't care non-deterministic, resulting in a huge reduction of the search space as well as easing implementations. For the completeness of basic narrowing several sufficient conditions are known, including termination. It is also shown in [26] that LNC is complete for arbitrary confluent TRSs and normalized solutions with respect to the selection function $\mathcal{S}_{\text{left}}$ that selects the leftmost equation in every goal. (For this general class of TRSs, LNC is not strongly complete [26, Counterexample 10].) Based on the latter result Middeldorp and Okui [25] present restrictions on the participating TRSs and solutions which guarantee that all non-determinism due to the choice of inference rules of LNC is removed. The resulting deterministic calculus LNC_d satisfies the optimality property that different derivations compute incomparable solutions for a class of TRSs that properly includes the class of TRSs for which a similar result was obtained by Antoy *et al.* in the setting of needed narrowing [1].

In this paper we extend LNC to deal with conditional TRSs (CTRSs for short). We present three main completeness results:

- LCNC with $\mathcal{S}_{\text{left}}$ is complete with respect to normalizable solutions for the class of confluent but not necessarily terminating conditional rewrite systems without so-called extra variables in the conditional parts of the rewrite rules.

- LCNC is strongly complete whenever basic conditional narrowing is complete. The latter is known for decreasing and confluent CTRSs without extra variables in the rewrite rules (Middeldorp and Hamoen [24]), for level-complete CTRSs with extra variables in the conditions only (Giovannetti and Moiso [9], Middeldorp and Hamoen [24]), and for terminating and shallow-confluent normal CTRSs with extra variables (Werner [33]).

- LCNC is complete for the class of terminating and level-confluent conditional rewrite systems without any restrictions on the distribution of variables in the rewrite rules. Unlike the previous two results, the proof of this last result does not provide any complete selection strategy. As a matter of fact, the selection strategy used in the proof is not effective in that it refers to the rewrite sequence that shows that the solution that we want to approximate with LCNC is actually a solution. It is an open question whether this result can be strengthened to completeness with respect to a fixed selection function or even to strong completeness.

The first two results generalize two of the three main results of [26] to the conditional case. The third result has no counterpart in the unconditional case. We stress that without a complete selection function, in implementations we need to backtrack over the choice of equations in goals in order to guarantee that all solutions are enumerated. This complicates implementations and, worse, leads to a dramatic increase in the search space, even more so since in conditional narrowing (whether presented as a single inference rule or in the form of a calculus like LCNC) the conditions of the applied rewrite rule are added to the current goal after every narrowing step.

The remainder of the paper is organized as follows. In the next section we recall some definitions pertaining to conditional rewriting and we present the calculus LCNC. Sections 3, 4, and 5 are devoted to the proofs of our three completeness results. We make some concluding remarks and list several open problems in Section 6. The Appendix contains the proofs of two technical lemmata in Section 4.

The results presented in this paper previously appeared in [14, 15, 12, 31].

2 Preliminaries

We assume familiarity with the basics of (conditional) term rewriting and narrowing. Surveys can be found in [2, 4, 21, 24]. We just recall some basic definitions in order to fix our notation and terminology.

A conditional term rewriting system (CTRS) over a signature $\mathcal{F}$ is a set $\mathcal{R}$ of (conditional) rewrite rules of the form $l \to r \Leftarrow c$ where the conditional part c is a (possibly empty) sequence $s_1 \approx t_1, \ldots, s_n \approx t_n$ of equations. All terms $l, r, s_1, \ldots, s_n, t_1, \ldots, t_n$ must belong to $\mathcal{T}(\mathcal{F}, \mathcal{V})$ and we require that l is not a variable. Here $\mathcal{V}$ denotes a countably infinite set of variables. Following [24], CTRSs are classified according to the distribution of variables in rewrite rules. A 1-CTRS contains no extra variables (i.e., $\mathcal{V}\mathrm{ar}(r, c) \subseteq \mathcal{V}\mathrm{ar}(l)$ for all rewrite rules $l \to r \Leftarrow c$), a 2-CTRS may contain extra variables in the conditions only ($\mathcal{V}\mathrm{ar}(r) \subseteq \mathcal{V}\mathrm{ar}(l)$ for all rewrite rules $l \to r \Leftarrow c$), and a 3-CTRS may also have extra variables in the right-hand sides provided these occur in the corresponding conditions ($\mathcal{V}\mathrm{ar}(r) \subseteq \mathcal{V}\mathrm{ar}(l, c)$ for all rewrite rules $l \to r \Leftarrow c$). Extra variables enable a more natural style of writing specifications of programs. For instance, using extra variables we can easily write the following specification of the efficient computation of Fibonacci numbers:

$$\begin{array}{rcll}
0 + y & \to & y & \\
\mathsf{s}(x) + y & \to & \mathsf{s}(x + y) & \\
\mathsf{fib}(0) & \to & \langle 0, \mathsf{s}(0) \rangle & \\
\mathsf{fib}(\mathsf{s}(x)) & \to & \langle z, y + z \rangle & \Leftarrow \;\; \mathsf{fib}(x) \approx \langle y, z \rangle
\end{array}$$

However, the presence of extra variables comes with a price. For instance, completeness results for narrowing that hold for arbitrary confluent TRSs and 1-CTRSs typically do not carry over to 2-CTRSs and 3-CTRSs without requiring some kind of termination assumption.

We assume that every CTRS contains the rewrite rule $x \approx x \to \mathtt{true}$. Here $\approx$ and $\mathtt{true}$ are function symbols that do not occur in the other rewrite rules. These symbols may only occur at the root position of terms. Let $\mathcal{R}$ be a CTRS. We inductively define unconditional TRSs $\mathcal{R}_n$ for $n \geqslant 0$ as follows:

$$\begin{array}{lcl}
\mathcal{R}_0 & = & \{\, x \approx x \to \mathtt{true} \,\}, \\
\mathcal{R}_{n+1} & = & \{\, l\theta \to r\theta \mid l \to r \Leftarrow c \in \mathcal{R} \text{ and } c\theta \to^*_{\mathcal{R}_n} \top \,\}.
\end{array}$$

Here $\top$ stands for any sequence of $\mathtt{true}$s. We define $s \to_{\mathcal{R}} t$ if and only if there exists an $n \geqslant 0$ such that $s \to_{\mathcal{R}_n} t$. We abbreviate $\to_{\mathcal{R}_n}$ to $\to_n$ and

$\to_{\mathcal{R}}$ to $\to$ if the CTRS $\mathcal{R}$ can be inferred from the context. Our CTRS are known as *join* CTRSs in the term rewriting literature.

A CTRS $\mathcal{R}$ is level-confluent (Giovannetti and Moiso [9]) if every $\mathcal{R}_n$ is confluent, i.e., ${}_n^*\!\leftarrow \cdot \to_n^* \subseteq \to_n^* \cdot {}_n^*\!\leftarrow$ for all $n \geqslant 0$, and shallow-confluent if ${}_m^*\!\leftarrow \cdot \to_n^* \subseteq \to_n^* \cdot {}_m^*\!\leftarrow$ for all $m, n \geqslant 0$. Shallow-confluent CTRSs are level-confluent but the reverse is not true. A CTRS $\mathcal{R}$ is level-terminating if every $\mathcal{R}_n$ is terminating. Level-termination is a weaker ([24]) property than termination. A level-complete CTRS $\mathcal{R}$ is both level-confluent and level-terminating. A CTRS $\mathcal{R}$ over a signature $\mathcal{F}$ is decreasing (Dershowitz *et al.* [6]) if there exists a well-founded order $\succ$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$ with the following three properties: $\succ$ contains $\to_{\mathcal{R}}$, $\succ$ has the subterm property (i.e., $\rhd \subseteq \succ$ where $s \rhd t$ if and only if t is proper subterm of s), and $l\theta \succ s\theta, t\theta$ for every rewrite rule $l \to r \Leftarrow c$ of $\mathcal{R}$, every equation $s \approx t$ in c, and every substitution θ. Note that according to this definition 2-CTRSs and 3-CTRSs are never decreasing. Decreasing CTRSs are terminating and, when there are finitely many rewrite rules, have a decidable rewrite relation. Sufficient syntactic conditions for level-confluence of 3-CTRSs are presented in Suzuki *et al.* [32].

An equation is a term of the form $s \approx t$. The constant $\mathtt{true}$ is also viewed as an equation. A goal is a sequence of equations. A substitution θ is a ($\mathcal{R}$-)solution of a goal G if $s\theta \leftrightarrow_{\mathcal{R}}^* t\theta$ for every equation $s \approx t$ in G. This is equivalent to validity of the equations in $G\theta$ in all models of the underlying conditional equational system of $\mathcal{R}$ (Kaplan [20]) and for confluent $\mathcal{R}$ to $G\theta \to_{\mathcal{R}}^* \top$. We abbreviate the latter to $\mathcal{R} \vdash G\theta$. A *normalized* solution satisfies the additional property that variables are mapped to normal forms with respect to $\mathcal{R}$.

For a substitution θ and a set of variables W, we denote $(W \setminus \mathcal{D}(\theta)) \cup \mathcal{I}(\theta\restriction_W)$ by $\mathcal{V}\mathrm{ar}_W(\theta)$. Here $\mathcal{D}(\theta) = \{x \in \mathcal{V} \mid \theta(x) \neq x\}$ denotes the domain of θ, which is always assumed to be finite, and $\mathcal{I}(\theta\restriction_W) = \bigcup_{x \in \mathcal{D}(\theta) \cap W} \mathcal{V}\mathrm{ar}(x\theta)$ the set of variables introduced by the restriction of θ to W.

The *lazy conditional narrowing calculus* LCNC consists of the following five inference rules:

[o] *outermost narrowing*

$$\frac{G', f(s_1, \ldots, s_n) \simeq t, G''}{G', s_1 \approx l_1, \ldots, s_n \approx l_n, r \approx t, c, G''}$$

if there exists a fresh variant $f(l_1, \ldots, l_n) \to r \Leftarrow c$ of a rewrite rule in

$\mathcal{R}$,

[i] *imitation*

$$\frac{G', f(s_1, \dots, s_n) \simeq x, G''}{(G', s_1 \approx x_1, \dots, s_n \approx x_n, G'')\theta}$$

if $\theta = \{x \mapsto f(x_1, \dots, x_n)\}$ with $x_1, \dots, x_n$ fresh variables,

[d] *decomposition*

$$\frac{G', f(s_1, \dots, s_n) \approx f(t_1, \dots, t_n), G''}{G', s_1 \approx t_1, \dots, s_n \approx t_n, G''},$$

[v] *variable elimination*

$$\frac{G', s \simeq x, G''}{(G', G'')\theta}$$

if $x \notin \mathcal{V}\mathrm{ar}(s)$ and $\theta = \{x \mapsto s\}$,

[t] *removal of trivial equations*

$$\frac{G', x \approx x, G''}{G', G''}.$$

In the rules [o], [i], and [v], $s \simeq t$ stands for $s \approx t$ or $t \approx s$. (Since the inference rules never produce the equation `true`, we assume that LCNC deals only with goals that do not contain `true`.) Note that the outermost narrowing rule is applicable as soon as the root symbol of one side s of an equation equals the root symbol of the left-hand side l of a rewrite rule. The *parameter-passing* equations $s_1 \approx l_1, \dots, s_n \approx l_n$ code the problem of unifying s and l. Further note that unlike higher-order narrowing calculi (e.g. [29, 22]) we do not permit outermost narrowing at variable positions. This makes the task of proving completeness (much) more challenging but results in a much smaller search space.

The only difference between LCNC and the calculus LNC of [26] is in the outermost narrowing rule: In LCNC we add the conditional part of the applied rewrite rule to the new goal.

If G and G' are the upper and lower goal in the inference rule $[\alpha]$ ($\alpha \in \{\mathrm{o}, \mathrm{i}, \mathrm{d}, \mathrm{v}, \mathrm{t}\}$), we write $G \Rightarrow_{[\alpha]} G'$. This is called an LCNC-step. The applied rewrite rule or substitution may be supplied as subscript, that is, we write things like $G \Rightarrow_{[\mathrm{o}], l \to r \Leftarrow c} G'$ and $G \Rightarrow_{[\mathrm{i}], \theta} G'$. A finite LCNC-derivation $G_1 \Rightarrow_{\theta_1} \cdots \Rightarrow_{\theta_{n-1}} G_n$ may be abbreviated to $G_1 \Rightarrow_{\theta}^{*} G_n$ where θ is the composition $\theta_1 \cdots \theta_{n-1}$ of the substitutions $\theta_1, \ldots, \theta_{n-1}$ computed along its steps. An LCNC-refutation is an LCNC-derivation ending in the empty goal $\Box$.

Example 2.1 *Consider the CTRS $\mathcal{R}$ for computing Fibonacci numbers from the introduction. The goal* $\mathsf{fib}(x) \approx \langle x, x\rangle$ *admits the solution* $\{x \mapsto \mathsf{s}(0)\}$ *because of the following rewrite sequence:*

$$\mathsf{fib}(\mathsf{s}(0)) \;\to\; \langle \mathsf{s}(0), 0 + \mathsf{s}(0)\rangle \;\to\; \langle \mathsf{s}(0), \mathsf{s}(0)\rangle$$

In the first step the rewrite rule $\mathsf{fib}(\mathsf{s}(x)) \to \langle z, y + z\rangle \Leftarrow \mathsf{fib}(x) \approx \langle y, z\rangle$ *is applied with substitution* $\{x \mapsto 0, y \mapsto 0, z \mapsto \mathsf{s}(0)\}$*; the instantiated condition is satisfied because of the rewrite rule* $\mathsf{fib}(0) \to \langle 0, \mathsf{s}(0)\rangle$. *The following* LCNC-*derivation ends in the unsolvable goal* $\mathsf{s}(0) \approx 0$*:*

$$\begin{array}{l}
\underline{\mathsf{fib}(x) \approx \langle x, x\rangle} \\
\quad \Downarrow_{[\mathrm{o}],\ \mathsf{fib}(0) \to \langle 0, \mathsf{s}(0)\rangle} \\
\underline{x \approx 0},\ \langle 0, \mathsf{s}(0)\rangle \approx \langle x, x\rangle \\
\quad \Downarrow_{[\mathrm{v}],\ \{x \mapsto 0\}} \\
\underline{\langle 0, \mathsf{s}(0)\rangle \approx \langle 0, 0\rangle} \\
\quad \Downarrow_{[\mathrm{d}]} \\
\underline{0 \approx 0},\ \mathsf{s}(0) \approx 0 \\
\quad \Downarrow_{[\mathrm{d}]} \\
\mathsf{s}(0) \approx 0
\end{array}$$

The underlined equations are selected in each step. Note that none of the inference rules of LCNC *are applicable to* $\mathsf{s}(0) \approx 0$. *In the first step of the above derivation the rewrite rule* $\mathsf{fib}(0) \to \langle 0, \mathsf{s}(0)\rangle$ *is chosen. If we choose the rule* $\mathsf{fib}(\mathsf{s}(x)) \to \langle z, y + z\rangle \Leftarrow \mathsf{fib}(x) \approx \langle y, z\rangle$ *instead,* LCNC *is able to solve the goal* $\mathsf{fib}(x) \approx \langle x, x\rangle$*:*

$$\begin{array}{l}
\underline{\mathsf{fib}(x) \approx \langle x, x\rangle} \\
\quad \Downarrow_{[\mathrm{o}],\ \mathsf{fib}(\mathsf{s}(x_1)) \to \langle z_1, y_1 + z_1\rangle \Leftarrow \mathsf{fib}(x_1) \approx \langle y_1, z_1\rangle}
\end{array}$$

$$x \approx \mathsf{s}(x_1),\ \underline{\langle z_1, y_1 + z_1\rangle \approx \langle x, x\rangle},\ \mathsf{fib}(x_1) \approx \langle y_1, z_1\rangle$$

$$\Downarrow_{[\mathsf{d}]}$$

$$x \approx \mathsf{s}(x_1),\ \underline{z_1 \approx x},\ y_1 + z_1 \approx x,\ \mathsf{fib}(x_1) \approx \langle y_1, z_1\rangle$$

$$\Downarrow_{[\mathsf{v}]},\ \{z_1 \mapsto x\}$$

$$x \approx \mathsf{s}(x_1),\ y_1 + x \approx x,\ \underline{\mathsf{fib}(x_1) \approx \langle y_1, x\rangle}$$

$$\Downarrow_{[\mathsf{o}]},\ \mathsf{fib}(0) \to \langle 0, \mathsf{s}(0)\rangle$$

$$x \approx \mathsf{s}(x_1),\ y_1 + x \approx x,\ \underline{x_1 \approx 0},\ \langle 0, \mathsf{s}(0)\rangle \approx \langle y_1, x\rangle$$

$$\Downarrow_{[\mathsf{v}]},\ \{x_1 \mapsto 0\}$$

$$x \approx \mathsf{s}(0),\ y_1 + x \approx x,\ \underline{\langle 0, \mathsf{s}(0)\rangle \approx \langle y_1, x\rangle}$$

$$\Downarrow_{[\mathsf{d}]}$$

$$x \approx \mathsf{s}(0),\ y_1 + x \approx x,\ \underline{0 \approx y_1},\ \mathsf{s}(0) \approx x$$

$$\Downarrow_{[\mathsf{v}]},\ \{y_1 \mapsto 0\}$$

$$\underline{x \approx \mathsf{s}(0)},\ 0 + x \approx x,\ \mathsf{s}(0) \approx x$$

$$\Downarrow_{[\mathsf{v}]},\ \{x \mapsto \mathsf{s}(0)\}$$

$$0 + \mathsf{s}(0) \approx \mathsf{s}(0),\ \underline{\mathsf{s}(0) \approx \mathsf{s}(0)}$$

$$\Downarrow_{[\mathsf{d}]}$$

$$0 + \mathsf{s}(0) \approx \mathsf{s}(0),\ \underline{0 \approx 0}$$

$$\Downarrow_{[\mathsf{d}]}$$

$$\underline{0 + \mathsf{s}(0) \approx \mathsf{s}(0)}$$

$$\Downarrow_{[\mathsf{o}]},\ 0 + y_2 \to y_2$$

$$0 \approx 0,\ \mathsf{s}(0) \approx y_2,\ \underline{y_2 \approx \mathsf{s}(0)}$$

$$\Downarrow_{[\mathsf{v}]},\ \{y_2 \mapsto \mathsf{s}(0)\}$$

$$\underline{0 \approx 0},\ \mathsf{s}(0) \approx \mathsf{s}(0)$$

$$\Downarrow_{[\mathsf{d}]}$$

$$\underline{\mathsf{s}(0) \approx \mathsf{s}(0)}$$

$$\Downarrow_{[\mathsf{d}]}$$

$$\underline{0 \approx 0}$$

$$\Downarrow_{[\mathsf{d}]}$$

$$\square$$

The solution computed by this refutation is obtained by composing the substitutions $\{z_1 \mapsto x\}$, $\{x_1 \mapsto 0\}$, $\{y_1 \mapsto 0\}$, $\{x \mapsto \mathsf{s}(0)\}$, $\{y_2 \mapsto \mathsf{s}(0)\}$ *employed in the* $\Rightarrow_{[\mathsf{v}]}$*-steps, and restricting the resulting substitution to the variable* x *in the initial goal, which yields* $\{x \mapsto \mathsf{s}(0)\}$.

The following lemma states the soundness of LCNC. The routine induction proof is omitted.

Lemma 2.2 *Let* $\mathcal{R}$ *be a CTRS and* G *a goal. If* $G \Rightarrow_\theta^* \square$ *then* $\theta\restriction_{\mathcal{V}\mathrm{ar}(G)}$ *is an* $\mathcal{R}$*-solution of* G. $\square$

3 Leftmost Selection

This section contains our first main result, the completeness of LCNC for arbitrary confluent 1-CTRSs with respect to normalized solutions and the leftmost selection function $\mathcal{S}_{\mathsf{left}}$. So we assume throughout this section that the sequence G' of equations to the left of the selected equation in the inference rules of LCNC is empty.

In Middeldorp *et al.* [26] the same result is proved for *unconditional* TRSs by means of a complicated inductive transformation process that operates on narrowing sequences. In the proof presented in this section we use conditional rewrite sequences instead. The advantage of rewriting is that rewrite steps applied to different parts of a goal or equation can be swapped at will, which greatly facilitates a proof of completeness with respect to a particular selection strategy. In the proof below we use the variant of conditional rewriting in which the list of instantiated conditions of the applied rewrite rule is explicitly added to the goal after every rewrite step. Formally, we use the relation $\rightarrowtail$ defined as follows: $G \rightarrowtail G'$ if $G = G_1, e, G_2$, $e \rightarrow e'$ by applying the conditional rewrite rule $l \rightarrow r \Leftarrow c$ with substitution θ (so $e' = e[r\sigma]_p$ for some position p in e and $\mathcal{R} \vdash c\sigma$), and $G' = G_1, e', c\sigma, G_2$. It is well-known ([3, 24]) that $\mathcal{R} \vdash G$ if and only if $G \rightarrowtail^* \top$. We assume without loss of generality that in a rewrite proof $G \rightarrowtail^* \top$ always the leftmost equation different from `true` is selected.

Below we define a couple of basic transformations on rewrite proofs $\Pi\colon G\theta \rightarrowtail^* \top$. In order to make the completeness proof work, we need to keep track of a number of variables along the transformation process. Since these variables cannot be inferred from the current rewrite proof Π, together with G and θ, we need to enrich rewrite proofs. This is the reason why we consider quadruples, called *states*, of the form $\langle G, \theta, \Pi, X\rangle$ where G is a goal, θ a solution of G, Π: $G\theta \rightarrowtail^* \top$ a rewrite proof of $G\theta$, and X a finite set of variables associated to Π. Variables in X are said to be *of interest* and variables in G but not in X are called *intermediate*. In order to avoid confusion, we occasionally write X-intermediate or even $\underline{\Pi}$-intermediate (when comparing different states with the same X).

We require that the properties defined below are satisfied.

Definition 3.1 *A state* $\underline{\Pi} = \langle G, \theta, \Pi, X\rangle$ *is called* normal *if* $\theta\lceil_X$ *is normalized. We say that* $\underline{\Pi}$ *satisfies the* variable condition *if for every equation* $s \approx t$ *in* $G = G_1, s \approx t, G_2$ *the following three conditions hold:*

VC1 *all intermediate variables in s occur in G_1 or all intermediate variables in t occur in G_1,*

VC2 *if $s\theta$ is rewritten in Π then all intermediate variables in s occur in G_1,*

VC3 *if $t\theta$ is rewritten in Π then all intermediate variables in t occur in G_1.*

A normal state with the variable condition is called admissible.

Example 3.2 *Consider the confluent 1-CTRS consisting of the rewrite rules*

$$\begin{aligned} \mathsf{f}(x,y) &\rightarrow \mathsf{g}(y) \Leftarrow x \approx \mathsf{a} \\ \mathsf{a} &\rightarrow \mathsf{b} \\ \mathsf{g}(\mathsf{b}) &\rightarrow \mathsf{b} \end{aligned}$$

and the goal $G = \mathsf{f}(x,y) \approx \mathsf{g}(y), \mathsf{g}(\mathsf{g}(y)) \approx \mathsf{a}$. *We determine θ, Π, and X such that $\underline{\Pi} = \langle G, \theta, \Pi, X\rangle$ is admissible. First of all, since the variable y occurs in both sides of the leftmost equation $\mathsf{f}(x,y) \approx \mathsf{g}(y)$, it must be a variable of interest for otherwise* VC1 *is violated. So $y \in X$ and hence, to satisfy normality, $\theta(y)$ should be a normal form. The only normal form that satisfies the second equation of G is* b, *with associated rewrite proof*

$$\mathsf{g}(\mathsf{g}(\mathsf{b})) \approx \mathsf{a} \rightarrowtail \mathsf{g}(\mathsf{b}) \approx \mathsf{a} \rightarrowtail \mathsf{b} \approx \mathsf{a} \rightarrowtail \mathsf{b} \approx \mathsf{b} \rightarrowtail \mathtt{true}.$$

Since $\mathsf{g}(\mathsf{g}(y)) \approx \mathsf{a}$ *does not contain intermediate variables, it satisfies* VC1, VC2, *and* VC3. *Likewise, since* $\mathsf{g}(y)$ *lacks intermediate variables,* $\mathsf{f}(x,y) \approx \mathsf{g}(y)$ *satisfies* VC3. *The only way to solve this equation is by applying the first rewrite rule to its (instantiated) left-hand side. Hence, to satisfy* VC2,

x cannot be an intermediate variable. Consequently, $x \in X$ and thus to conclude that $\underline{\Pi}$ is admissible it only remains to show that we can substitute a normal form for x such that the equation $\mathsf{f}(x, \mathsf{b}) \approx \mathsf{g}(\mathsf{b})$ is solvable. It is easy to see that we should again take the normal form b, with associated rewrite proof

$$\begin{aligned} \mathsf{f}(\mathsf{b},\mathsf{b}) \approx \mathsf{g}(\mathsf{b}) &\rightarrowtail \mathsf{g}(\mathsf{b}) \approx \mathsf{g}(\mathsf{b}), \mathsf{b} \approx \mathsf{a} \rightarrowtail \mathtt{true}, \mathsf{b} \approx \mathsf{a} \\ &\rightarrowtail \mathtt{true}, \mathsf{b} \approx \mathsf{b} \rightarrowtail \top. \end{aligned}$$

An example of a goal without admissible states is $\mathsf{f}(x) \approx x$ with respect to the TRS consisting of the rule $\mathsf{a} \rightarrow \mathsf{f}(\mathsf{a})$. Note that $\mathsf{f}(x) \approx x$ has no normalized solutions.

Lemma 3.3 *Let $\underline{\Pi} = \langle G, \theta, \Pi, X\rangle$ be an admissible state with $G = s \approx t, H$.*

1. *If $s\theta$ is rewritten in Π then s is not a variable and does not include intermediate variables.*
2. *If $t\theta$ is rewritten in Π then t is not a variable and does not include intermediate variables.*

Proof We prove the first statement. Suppose the left-hand side of $s\theta \approx t\theta$ is rewritten in Π. By **VC2** all intermediate variables in s should occur in the equations to the left of $s \approx t$ in G. Since $s \approx t$ is the leftmost equation, there cannot be intermediate variables in s. Hence, if s is a variable then it must be a variable of interest and thus $s\theta$ is a normal form because $\underline{\Pi}$ is normal. This however contradicts the assumption that $s\theta$ is rewritten in Π. The second statement is proved in exactly the same way (with **VC3** instead of **VC2**). □

In the following transformation lemmata $\underline{\Pi}$ denotes an admissible state $\langle G, \theta, \Pi, X\rangle$ such that $G = s \approx t, H$ and, in Lemmata 3.4, 3.5, and 3.7, W denotes a finite set of variables such that $\mathcal{V}\mathrm{ar}(G) \subseteq W$. Recall our earlier assumption that Π respects $\mathcal{S}_{\mathsf{left}}$. In particular, in the first step of Π the equation $s \approx t$ is selected.

Lemma 3.4 *Let $s = f(s_1, \ldots, s_n)$ and suppose that a reduct of $s\theta \approx t\theta$ in Π is rewritten at position 1. If $l \rightarrow r \Leftarrow c$ is the employed rewrite rule in the first such step then there exists an admissible state $\phi_{[\mathsf{o}]}(\underline{\Pi}) = \langle G', \theta', \Pi', X\rangle$ with $G' = s \approx l, r \approx t, c, H$ such that $\theta' = \theta\ [W]$.*

Proof The given rewrite proof Π is of the form

$$G\theta \rightarrowtail^* l\tau \approx t', C, H\theta \rightarrowtail r\tau \approx t', c\tau, C, H\theta \rightarrowtail^* \top.$$

Here C are the instantiated conditions of the rewrite rules applied in the rewrite sequence from $s\theta \approx t\theta$ to $l\tau \approx t'$. Without loss of generality we assume that $\mathcal{V}\mathrm{ar}(l \to r \Leftarrow c) \cap (X \cup W) = \varnothing$ and $\mathcal{D}(\tau) = \mathcal{V}\mathrm{ar}(l \to r \Leftarrow c)$. Hence the substitution $\theta' = \theta \cup \tau$ is well-defined. Since $\mathcal{D}(\tau) \cap W = \varnothing$, $\theta' = \theta\ [W]$. We have $G'\theta' = s\theta \approx l\tau, r\tau \approx t\theta, c\tau, H\theta$. The first part of Π can be transformed into

$$\begin{aligned} G'\theta' &\rightarrowtail^* l\tau \approx l\tau, C_1, r\tau \approx t\theta, c\tau, H\theta \quad \rightarrowtail^* \ l\tau \approx l\tau, C_1, r\tau \approx t', C_2, c\tau, H\theta \\ &\rightarrowtail \mathtt{true}, C_1, r\tau \approx t', C_2, c\tau, H\theta. \end{aligned}$$

Here C_1, C_2 and C consist of the same equations in possibly different order. Hence by rearranging the steps in the remaining part of Π we obtain

$$\mathtt{true}, C_1, r\tau \approx t', C_2, c\tau, H\theta \rightarrowtail^* \top.$$

Concatenating these two derivations yields the rewrite proof Π'. It remains to show that the state $\phi_{[o]}(\underline{\Pi}) = \langle G', \theta', \Pi', X\rangle$ is admissible. Since $\theta'\restriction_X = \theta\restriction_X \cup \tau\restriction_X = \theta\restriction_X$, $\phi_{[o]}(\underline{\Pi})$ inherits normality from $\underline{\Pi}$. For the variable condition we need some more effort. Below we make tacit use of the observation that a variable $x \in \mathcal{V}\mathrm{ar}(s \approx t, H)$ is $\underline{\Pi}$-intermediate in G if and only if x is $\phi_{[o]}(\underline{\Pi})$-intermediate in G'. First consider the equation $s \approx l$. Since $s\theta'$ is rewritten in Π', we obtain from Lemma 3.3(1) that s does not contain intermediate variables. Hence **VC1** and **VC2** are trivially satisfied. By construction, the right-hand side of $s\theta' \approx l\theta'$ is never rewritten in Π'. Hence **VC3** holds vacuously. Next consider the equations in $r \approx t, c$. Because we deal with CTRSs without extra variables, all variables in r and c occur in l and hence the three variable conditions are true for the equations in c and $r \approx t$ satisfies **VC1** and **VC2**. Suppose the right-hand side of $r\theta' \approx t\theta'$ is rewritten in Π'. By construction of Π', this is only possible if the right-hand side of $s\theta \approx t\theta$ is rewritten in Π. According to Lemma 3.3(2) t does not contain intermediate variables and thus the equation $r \approx t$ in G' satisfies **VC3**. Finally, consider an equation $s' \approx t'$ in $H = H_1, s' \approx t', H_2$. Let V_1 be the set of intermediate variables in s' and V_2 the set of intermediate variables in t'. Since $\underline{\Pi}$ is admissible, $V_1 \subseteq \mathcal{V}\mathrm{ar}(s \approx t, H_1)$ or $V_2 \subseteq \mathcal{V}\mathrm{ar}(s \approx t, H_1)$. Hence also $V_1 \subseteq \mathcal{V}\mathrm{ar}(s \approx l, r \approx t, H_1)$ or $V_2 \subseteq \mathcal{V}\mathrm{ar}(s \approx l, r \approx t, H_1)$. This proves **VC1**. The proof of **VC2** is just as easy: If $s'\theta'$ is rewritten in Π' then $s'\theta$

is rewritten in Π and thus all intermediate variables in s' occur in $s \approx t, H_1$ and therefore also in $s \approx l, r \approx t, H_1$. Finally, **VC3** is proved in exactly the same way. □

Note that the above proof breaks down if we admit extra variables in the conditional rewrite rules. Further note that the proof remains valid if we would put the equation $r \approx t$ after the conditions c in G''.

Lemma 3.5 *Let $s = f(s_1, \ldots, s_n)$ and $t \in \mathcal{V}$. If* $\mathrm{root}(t\theta) = f$ *then there exists an admissible state $\phi_{[i]}(\underline{\Pi}) = \langle G', \theta', \Pi, X' \rangle$ with $G' = G\sigma_1$ such that $\sigma_1\theta' = \theta\ [W]$. Here $\sigma_1 = \{t \mapsto f(x_1, \ldots, x_n)\}$ with $x_1, \ldots, x_n \notin W$.*

Proof Write $t\theta = f(t_1, \ldots, t_n)$ and define $\theta' = \theta \cup \{x_i \mapsto t_i \mid 1 \leqslant i \leqslant n\}$. One easily verifies that $\sigma_1\theta' = \theta\ [W]$. We have $G'\theta' = G\sigma_1\theta' = G\theta$ and thus Π is a rewrite proof of $G'\theta'$. We define $X' = \bigcup_{x \in X} \mathcal{V}\mathrm{ar}(x\sigma_1)$. Equivalently,

$$X' = \begin{cases} (X \setminus \{t\}) \cup \{x_1, \ldots, x_n\} & \text{if } t \in X, \\ X & \text{otherwise.} \end{cases}$$

It remains to show that $\phi_{[i]}(\underline{\Pi})$ is admissible. First we show that $\theta' \restriction_{X'}$ is normalized. We consider two cases. If $t \in X$ then $\theta' \restriction_{X'} = \theta \restriction_{X \setminus \{t\}} \cup \{x_i \mapsto t_i \mid 1 \leqslant i \leqslant n\}$. The substitution $\theta \restriction_{X \setminus \{t\}}$ is normalized because $\theta \restriction_X$ is normalized. Furthermore, since every t_i is a subterm of $t\theta$ and $t\theta$ is a normal form because $\underline{\Pi}$ is normal and $t \in X$, $\{x_i \mapsto t_i \mid 1 \leqslant i \leqslant n\}$ is normalized as well. If $t \notin X$ then $\theta' \restriction_{X'} = \theta \restriction_X$ is normalized by assumption. Next we show that every equation in G' satisfies the variable condition. Again we consider two cases.

1. Suppose that $t \in X$. Then $x_1, \ldots, x_n$ belong to X' and thus the right-hand side $t\sigma_1 = f(x_1, \ldots, x_n)$ of the leftmost equation $s\sigma_1 \approx t\sigma_1$ in G' does not contain X'-intermediate variables. Hence $s\sigma_1 \approx t\sigma_1$ satisfies **VC1** and **VC3**. Suppose $s\sigma_1\theta'$ is rewritten in Π. Then, as $\underline{\Pi}$ is admissible, s does not contain X-intermediate variables. We have to show that $s\sigma_1$ does not contain X'-intermediate variables. Since the variables $x_1, \ldots, x_n$ are of interest, it follows that every X'-intermediate variable in $s\sigma_1$ is X-intermediate in s. Therefore $s\sigma_1 \approx t\sigma_1$ satisfies **VC3**.

   Next consider an equation $s'\sigma_1 \approx t'\sigma_1$ in $H\sigma_1 = (H_1, s' \approx t', H_2)\sigma_1$. Let V_1 be the set of X'-intermediate variables in $s'\sigma_1$ and V_2 the set of

X'-intermediate variables in $t'\sigma_1$. Since the variables $x_1, \ldots, x_n$ are not X'-intermediate and t is not X-intermediate, V_1 (V_2) coincides with the set of X-intermediate variables in s' (t'). Since $\underline{\Pi}$ is admissible, $V_1 \subseteq \mathcal{V}\mathrm{ar}(s \approx t, H_1)$ or $V_2 \subseteq \mathcal{V}\mathrm{ar}(s \approx t, H_1)$. Because $t \notin V_1 \cup V_2$, $V_1 \subseteq \mathcal{V}\mathrm{ar}((s \approx t, H_1)\sigma_1)$ or $V_2 \subseteq \mathcal{V}\mathrm{ar}((s \approx t, H_1)\sigma_1)$. This concludes the proof of **VC1**. Next we prove **VC2**. If $s'\sigma_1\theta' = s'\theta$ is rewritten in Π then $V_1 \subseteq \mathcal{V}\mathrm{ar}(s \approx t, H_1)$ and as $t \notin V_1$ also $V_1 \subseteq \mathcal{V}\mathrm{ar}((s \approx t, H_1)\sigma_1)$. The proof of **VC3** is just as easy.

2. Suppose that $t \notin X$. Then t is $\underline{\Pi}$-intermediate and $x_1, \ldots, x_n$ are $\phi_{[i]}(\underline{\Pi})$-intermediate. First consider the equation $s\sigma_1 \approx t\sigma_1$. Since t is intermediate, s cannot contain intermediate variables. In particular, t does not occur in s and therefore $s\sigma_1 = s$. So $s\sigma_1 \approx t\sigma_1$ satisfies **VC1** and **VC2**. Since t is a variable, $t\sigma_1\theta' = t\theta$ cannot be rewritten in Π as a consequence of Lemma 3.3(2) and hence **VC3** is satisfied too.

   Next consider an equation $s'\sigma_1 \approx t'\sigma_1$ in $H\sigma_1 = (H_1, s' \approx t', H_2)\sigma_1$. Let V_1' (V_2') be the set of $\phi_{[i]}(\underline{\Pi})$-intermediate variables in $s'\sigma_1$ ($t'\sigma_1$) and let V_1 (V_2) be the set of $\underline{\Pi}$-intermediate variables in s' (t'). We have

$$V_1' = \begin{cases} (V_1 \setminus \{t\}) \cup \{x_1, \ldots, x_n\} & \text{if } t \in \mathcal{V}\mathrm{ar}(s'), \\ V_1 & \text{otherwise,} \end{cases}$$

   and

$$V_2' = \begin{cases} (V_2 \setminus \{t\}) \cup \{x_1, \ldots, x_n\} & \text{if } t \in \mathcal{V}\mathrm{ar}(t'), \\ V_2 & \text{otherwise.} \end{cases}$$

   Because $\underline{\Pi}$ is admissible, $V_1 \subseteq \mathcal{V}\mathrm{ar}(s \approx t, H_1)$ or $V_2 \subseteq \mathcal{V}\mathrm{ar}(s \approx t, H_1)$. We consider the former. To conclude **VC1** it is sufficient to show that $V_1' \subseteq \mathcal{V}\mathrm{ar}((s \approx t, H_1)\sigma_1)$. We distinguish two cases. If $t \in \mathcal{V}\mathrm{ar}(s')$ then $t \in V_1$ and thus $x_1, \ldots, x_n \in \mathcal{V}\mathrm{ar}((s \approx t, H_1)\sigma_1)$. Since we also have the inclusion $V_1 \setminus \{t\} \subseteq \mathcal{V}\mathrm{ar}((s \approx t, H_1)\sigma_1)$, $V_1' \subseteq \mathcal{V}\mathrm{ar}((s \approx t, H_1)\sigma_1)$ holds. If $t \notin \mathcal{V}\mathrm{ar}(s')$ then $t \notin V_1$ and thus $V_1' = V_1 \subseteq \mathcal{V}\mathrm{ar}((s \approx t, H_1)\sigma_1)$. This proves **VC1**. For **VC2** we reason as follows. Suppose $s'\sigma_1\theta' = s'\theta$ is rewritten in Π. This implies that $V_1 \subseteq \mathcal{V}\mathrm{ar}(s \approx t, H_1)$ and hence $V_1' \subseteq \mathcal{V}\mathrm{ar}((s \approx t, H_1)\sigma_1)$ by using similar arguments as before. The proof of **VC3** is again very similar.

□

Lemma 3.6 *Let $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$, and suppose that no reduct of $s\theta \approx t\theta$ in Π is rewritten at position 1 or 2. There exists an admissible state $\phi_{[\mathrm{d}]}(\underline{\Pi}) = \langle G', \theta, \Pi', X \rangle$ with $G' = s_1 \approx t_1, \ldots, s_n \approx t_n, H$.*

Proof The given rewrite proof Π is of the form

$$G\theta \ \rightarrowtail^* \ f(u_1, \ldots, u_n) \approx f(u_1, \ldots, u_n), C, H\theta \ \rightarrowtail \ \mathtt{true}, C, H\theta \ \rightarrowtail^* \ \top.$$

Here C are the instantiated conditions of the rewrite rules applied in the rewrite sequence from $s\theta \approx t\theta$ to $f(u_1, \ldots, u_n) \approx f(u_1, \ldots, u_n)$. The first part of Π can be transformed into

$$\begin{aligned} G'\theta' \ &\rightarrowtail^* \ u_1 \approx u_1, C_1, \ldots, u_n \approx u_n, C_n, H\theta \\ &\rightarrowtail^* \ \mathtt{true}, C_1, \ldots, \mathtt{true}, C_n, H\theta. \end{aligned}$$

Here $C_1, \ldots, C_n$ and C consist of the same equations in possibly different order. Hence by rearranging the steps in the latter part of Π we obtain

$$\mathtt{true}, C_1, \ldots, \mathtt{true}, C_n, H\theta \ \rightarrowtail^* \ \top.$$

Concatenating these two derivations yields the rewrite proof Π' of $G'\theta$. It remains to show that the state $\phi_{[\mathrm{d}]}(\underline{\Pi}) = \langle G', \theta, \Pi', X \rangle$ is admissible. Since $\underline{\Pi}$ has the same θ and X, normality is obvious. Because $\underline{\Pi}$ satisfies condition VC1, s or t does not contain intermediate variables. Hence there are no intermediate variables in $s_1, \ldots, s_n$ or in $t_1, \ldots, t_n$. Consequently, the equations $s_1 \approx t_1, \ldots, s_n \approx t_n$ in G' satisfy VC1. The conditions VC2 and VC3 are also easily verified. For instance, suppose that $t_i\theta$ is rewritten in Π'. Then, by construction of Π', $t\theta$ is rewritten in Π. According to Lemma 3.3, t does not contain intermediate variables and since t_i is a subterm of t, t_i also lacks intermediate variables. By using similar arguments one easily verifies that the equations in H satisfy the three conditions. □

Lemma 3.7 *Let $t \in \mathcal{V}$, $s \neq t$, and suppose that in the first step of Π $s\theta \approx t\theta$ is rewritten at the root position. There exists an admissible state $\phi_{[\mathrm{v}]}(\underline{\Pi}) = \langle G', \theta, \Pi', X' \rangle$ with $G' = H\sigma_1$ such that $\sigma_1\theta = \theta$ $[W]$. Here $\sigma_1 = \{t \mapsto s\}$.*

Proof Since $s\theta \approx t\theta$ is rewritten to $\mathtt{true}$ by the rule $x \approx x \to \mathtt{true}$, we must have $s\theta = t\theta$. Hence $t\sigma_1\theta = s\theta = t\theta$. For variables y different from t we have $y\sigma_1\theta = y\theta$. Hence $\sigma_1\theta = \theta$ $[W]$. Since $\mathcal{V}\mathrm{ar}(H) \subseteq W$, $G'\theta = H\sigma_1\theta = H\theta$

and thus from the tail of the rewrite proof Π: $G\theta \rightarrowtail \mathtt{true}$, $H\theta \rightarrowtail^* \top$ we can extract a rewrite proof Π' of $G'\theta$. We define $X' = \cup_{x \in X} \mathcal{V}\mathrm{ar}(x\sigma_1)$. Clearly

$$X' = \begin{cases} (X \setminus \{t\}) \cup \mathcal{V}\mathrm{ar}(s) & \text{if } t \in X, \\ X & \text{otherwise.} \end{cases}$$

It remains to show that $\phi_{[v]}(\underline{\Pi})$ is admissible. First we show that $\theta\lceil_{X'}$ is normalized. We consider two cases. If $t \in X$ then $\theta\lceil_{X'} = \theta\lceil_{X\setminus\{t\}\cup\mathcal{V}\mathrm{ar}(s)}$. The substitution $\theta\lceil_{X\setminus\{t\}}$ is normalized because $\theta\lceil_X$ is normalized. If $x \in \mathcal{V}\mathrm{ar}(s)$ then $x\theta$ is a subterm of $s\theta = t\theta$. Since $t\theta$ is a normal form by the normality of $\underline{\Pi}$, so is $x\theta$. Hence $\theta\lceil_{\mathcal{V}\mathrm{ar}(s)}$ is normalized as well. If $t \notin X$ then $\theta\lceil_{X'} = \theta\lceil_X$ is normalized by assumption. Next we show that every equation in G' satisfies the variable condition. Let $s'\sigma_1 \approx t'\sigma_1$ be an equation in $H\sigma_1 = (H_1, s' \approx t', H_2)\sigma_1$. Let V_1' (V_2') be the set of X'-intermediate variables in $s'\sigma_1$ ($t'\sigma_1$) and let V_1 (V_2) be the set of intermediate variables in s' (t'). We consider two cases.

1. Suppose that $t \in X$. Then $\mathcal{V}\mathrm{ar}(s) \subseteq X'$. So the variables in $\mathcal{V}\mathrm{ar}(s)$ are not X'-intermediate and t is not X-intermediate. It follows that $V_1' = V_1$ and $V_2' = V_2$. Since $\underline{\Pi}$ is admissible, $V_1 \subseteq \mathcal{V}\mathrm{ar}(H_1)$ or $V_2 \subseteq \mathcal{V}\mathrm{ar}(H_1)$. Because $t \notin V_1 \cup V_2$, $V_1' \subseteq \mathcal{V}\mathrm{ar}(H_1\sigma_1)$ or $V_2' \subseteq \mathcal{V}\mathrm{ar}(H_1\sigma_1)$. This concludes the proof of **VC1**. The proofs of **VC2** and **VC3** also easily follow from the identities $V_1' = V_1$ and $V_2' = V_2$ and the fact that $t \notin V_1 \cup V_2$.

2. Suppose that $t \notin X$. Then t is $\underline{\Pi}$-intermediate and all variables in $\mathcal{V}\mathrm{ar}(s)$ are $\phi_{[v]}(\underline{\Pi})$-intermediate. We have

$$V_1' = \begin{cases} (V_1 \setminus \{t\}) \cup \mathcal{V}\mathrm{ar}(s) & \text{if } t \in \mathcal{V}\mathrm{ar}(s'), \\ V_1 & \text{otherwise,} \end{cases}$$

and

$$V_2' = \begin{cases} (V_2 \setminus \{t\}) \cup \mathcal{V}\mathrm{ar}(s) & \text{if } t \in \mathcal{V}\mathrm{ar}(t'), \\ V_2 & \text{otherwise.} \end{cases}$$

Because $\underline{\Pi}$ is admissible, $V_1 \subseteq \mathcal{V}\mathrm{ar}(H_1)$ or $V_2 \subseteq \mathcal{V}\mathrm{ar}(H_1)$. We consider the latter. To conclude **VC1** it is therefore sufficient to show that $V_2' \subseteq \mathcal{V}\mathrm{ar}(H_1\sigma_1)$. We distinguish two cases. If $t \in \mathcal{V}\mathrm{ar}(t')$ then $t \in V_2$ and thus $\mathcal{V}\mathrm{ar}(s) \subseteq \mathcal{V}\mathrm{ar}(H_1\sigma_1)$. Since the inclusion $V_2 \setminus \{t\} \subseteq \mathcal{V}\mathrm{ar}(H_1\sigma_1)$

also holds, $V_2' \subseteq \mathcal{V}\mathrm{ar}(H_1\sigma_1)$ as desired. If $t \notin \mathcal{V}\mathrm{ar}(t')$ then $t \notin V_2$ and thus $V_2' = V_2 \subseteq \mathcal{V}\mathrm{ar}(H_1\sigma_1)$. This completes the proof of **VC1**. The proofs of **VC2** and **VC3** are based on similar arguments and omitted.

□

Lemma 3.8 *Let $t \in \mathcal{V}$, $s = t$, and suppose that in the first step of Π $s\theta \approx t\theta$ is rewritten at the root position. There exists an admissible state $\phi_{[t]}(\underline{\Pi}) = \langle G', \theta, \Pi', X\rangle$ with $G' = H$.*

Proof The given rewrite proof Π has the form $G\theta \rightarrowtail \texttt{true}, H\theta \rightarrowtail^* \top$. From the tail of Π we extract a rewrite proof Π' of $G'\theta = H\theta$. It is easy to show that $\phi_{[t]}(\underline{\Pi})$ is admissible. □

Lemma 3.9 *There exists an admissible state $\phi_{\mathrm{swap}}(\underline{\Pi}) = \langle G', \theta, \Pi', X\rangle$ with $G' = t \approx s, H$.*

Proof The given rewrite proof Π: $(s \approx t, H)\theta \rightarrowtail^* \top$ is transformed into a rewrite proof Π' of $(t \approx s, H)\theta$ by simply swapping the two sides of every reduct of $s\theta \approx t\theta$. This clearly does not affect normality and since the variable condition is symmetric with respect the two sides of an equation it follows that $\phi_{\mathrm{swap}}(\underline{\Pi})$ is admissible. □

We want to stress that swapping different equations (as opposed to the two sides of a single equation as in the preceding lemma) does *not* preserve the variable condition. This makes a lot of sense, since if it would preserve the variable condition then we could prove strong completeness of LCNC but from [26] we already know that the LNC is not strongly complete (for the class of confluent TRSs with respect to normalized solutions).

In the proof of the main theorem below, we use induction on admissible states with respect to the well-founded order defined below. This order is essentially the same as the one used in the completeness proofs of [26].

Definition 3.10 *The complexity $|\underline{\Pi}|$ of a state $\underline{\Pi} = \langle G, \theta, \Pi, X\rangle$ is defined as the triple consisting of (1) the number of rewrite steps in Π at non-root positions, (2) the multiset $|\mathcal{M}\mathcal{V}\mathrm{ar}(G)\theta|$, and (3) the number of occurrences of symbols different from $\approx$ and* `true` *in G. Here $\mathcal{M}\mathcal{V}\mathrm{ar}(G)$ denotes the* *multiset of variable occurrences in G, and for any multiset $M = \{t_1, \ldots, t_n\}$ of terms, $M\theta$ and $|M|$ denote the multisets $\{t_1\theta, \ldots, t_n\theta\}$ and $\{|t_1|, \ldots, |t_n|\}$, respectively. The well-founded order $\gg$ on states is defined as follows: $\underline{\Pi}_1 \gg \underline{\Pi}_2$ if $|\underline{\Pi}_1|$ $\mathsf{lex}(>, >_{\mathsf{mul}}, >)$ $|\underline{\Pi}_2|$. Here $>$ denotes the standard order on natural numbers and $>_{\mathsf{mul}}$ denotes the multiset extension of $>$, i.e., $M >_{\mathsf{mul}} N$ for finite multisets M, N if and only if there exist multisets X and Y such that $\varnothing \neq X \subseteq M$, $N = (M - X) \uplus Y$, and for every $y \in Y$ there exists an $x \in X$ with $x \succ y$; with $-$ and $\uplus$ denoting multiset difference and sum. Furthermore, $\mathsf{lex}(>, >_{\mathsf{mul}}, >)$ denotes the lexicographic product of $>$, $>_{\mathsf{mul}}$, and $>$.*

From [5] we know that $>_{\mathsf{mul}}$ inherits well-foundedness from $>$. Consequently, the lexicographic product of $>$, $>_{\mathsf{mul}}$, and $>$ is a well-founded order and hence $\gg$ is a well-founded order on states.

Lemma 3.11 *Let $\underline{\Pi}$ be a state and $\alpha \in \{\mathrm{o}, \mathrm{i}, \mathrm{d}, \mathrm{v}, \mathrm{t}\}$. We have $\underline{\Pi} \gg \phi_{[\alpha]}(\underline{\Pi})$ whenever the latter is defined. Moreover, $|\underline{\Pi}| = |\phi_{\mathsf{swap}}(\underline{\Pi})|$.*

Proof Basically the same as the proof of Lemma 20 in [26]. For $\alpha = \mathrm{o}$ we observe a decrease in the first component of $|\underline{\Pi}|$. Here it is essential that we work with $\rightarrowtail$ instead of the ordinary rewrite relation $\rightarrow$; in this way steps that take place in the conditional part of the applied rewrite rule are already accounted for in $|\underline{\Pi}|$. For $\alpha \in \{\mathrm{i}, \mathrm{d}, \mathrm{v}, \mathrm{t}\}$ the number of rewrite steps at non-root positions remains the same. For $\alpha \in \{\mathrm{i}, \mathrm{v}, \mathrm{t}\}$ the second component of $|\underline{\Pi}|$ decreases. For $\alpha = \mathrm{d}$ the second component remains the same while the third component of $|\underline{\Pi}|$ decreases. □

Theorem 3.12 *Let $\mathcal{R}$ be a confluent CTRS without extra variables and G a goal. For every normalized solution θ of G there exists an* LCNC*-refutation $G \Rightarrow^*_\sigma \square$ respecting $\mathcal{S}_{\mathsf{left}}$ such that $\sigma \leqslant \theta$ $[\mathcal{V}\mathrm{ar}(G)]$.*

Proof Because $\mathcal{R}$ is confluent, $G\theta$ admits a rewrite proof Π. Consider the state $\underline{\Pi} = \langle G, \theta, \Pi, X\rangle$ with $X = \mathcal{V}\mathrm{ar}(G)$. By assumption $\theta\restriction_X$ is normalized. Since all variables of G are of interest, G does not contain intermediate variables and hence the variable condition is trivially satisfied. Therefore $\underline{\Pi}$ is admissible. We use induction on the complexity of $\underline{\Pi}$. In order to make the induction work we prove $\sigma \leqslant \theta$ $[W]$ for a finite set of variables W that includes $\mathcal{V}\mathrm{ar}(G)$. The base case is trivial since G must be the empty goal (and thus we can take $\sigma = \varepsilon$, the empty substitution). For the induction step we proceed as follows. We prove the existence of an LCNC-step $\Psi_1 \colon G \Rightarrow_{\sigma_1} G'$ that respects $\mathcal{S}_{\mathsf{left}}$ and an admissible state $\underline{\Pi'} = \langle G', \theta', \Pi', X' \rangle$ such that $\sigma_1\theta' = \theta\ [W]$. Let $G = s \approx t, H$. We distinguish the following cases, depending on what happens to $s\theta \approx t\theta$ in Π.

1. Suppose no reduct of $s\theta \approx t\theta$ is rewritten at position 1 or 2. We distinguish five further cases.

   (a) Suppose $s, t \notin \mathcal{V}$. We may write $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$. From Lemma 3.6 we obtain an admissible state $\phi_{[\mathsf{d}]}(\underline{\Pi}) = \langle G', \theta', \Pi', X' \rangle$ with $G' = s_1 \approx t_1, \ldots, s_n \approx t_n, H$, $\theta' = \theta$, and $X' = X$. We have $\Psi_1 \colon G \Rightarrow_{[\mathsf{d}]} G'$. Take $\sigma_1 = \varepsilon$ and $\underline{\Pi'} = \phi_{[\mathsf{d}]}(\underline{\Pi})$.

   (b) Suppose $t \in \mathcal{V}$ and $s = t$. According to Lemma 3.3 no $s\theta$ and $t\theta$ are not rewritten and hence in the first step of Π $s\theta \approx t\theta$ is rewritten at the root position. Hence Lemma 3.8 is applicable, yielding an admissible state $\phi_{[\mathsf{t}]}(\underline{\Pi}) = \langle G', \theta', \Pi', X' \rangle$ with $G' = H$, $\theta' = \theta$, and $X' = X$. We have $\Psi_1 \colon G \Rightarrow_{[\mathsf{t}]} G'$. Take $\sigma_1 = \varepsilon$ and $\underline{\Pi'} = \phi_{[\mathsf{t}]}(\underline{\Pi})$.

   (c) Suppose $t \in \mathcal{V}$, $s \neq t$, and a reduct of $s \approx t$ is rewritten at a non-root position. From Lemma 3.3 we infer that s is not a variable and moreover that $t\theta$ is not rewritten in Π. Hence we may write $s = f(s_1, \ldots, s_n)$ and we have $\mathrm{root}(t\theta) = f$. Consequently, Lemma 3.5 is applicable, yielding an admissible state $\phi_{[\mathsf{i}]}(\underline{\Pi}) = \langle G'', \theta'', \Pi'', X'' \rangle$ with $G'' = G\sigma_1$, $\Pi'' = \Pi$, and $\sigma_1\theta'' = \theta\ [W]$ for the substitution $\sigma = \{t \mapsto f(x_1, \ldots, x_n)\}$. We have $G'' = (f(s_1, \ldots, s_n) \approx f(x_1, \ldots, x_n), H)\sigma_1$. By assumption no reduct of $s\sigma\theta'' \approx t\sigma\theta''$ is rewritten at position 1 or 2. Hence we can apply Lemma 3.6. This yields an admissible state $\phi_{[\mathsf{d}]}(\phi_{[\mathsf{i}]}(\underline{\Pi})) = \langle G', \theta', \Pi', X' \rangle$ with $G' = (s_1 \approx x_1, \ldots, s_n \approx x_n, H)\sigma_1$, $\theta' = \theta''$, and $X' = X''$. We have $\Psi_1 \colon G \Rightarrow_{[\mathsf{i}], \sigma_1} G'$ and $\sigma_1\theta' = \sigma_1\theta'' = \theta\ [W]$. Take $\underline{\Pi'} = \phi_{[\mathsf{d}]}(\phi_{[\mathsf{i}]}(\underline{\Pi}))$.

   (d) Suppose $t \in \mathcal{V}$, $s \neq t$, and the first rewrite step takes place at the root position of $s \approx t$. Lemma 3.7 yields an admissible state $\phi_{[\mathsf{v}]}(\underline{\Pi}) = \langle G', \theta', \Pi', X' \rangle$ with $G' = G\sigma$, $\Pi' = \Pi$, and $\sigma_1\theta' = \theta\ [W]$ for the substitution $\sigma_1 = \{t \mapsto s\}$. We have $\Psi_1 \colon G \Rightarrow_{[\mathsf{v}], \sigma_1} G'$. Take $\underline{\Pi'} = \phi_{[\mathsf{v}]}(\underline{\Pi})$.

(e) In the remaining case we have $t \notin \mathcal{V}$ and $s \in \mathcal{V}$. This case reduces to case 1(c) or 1(d) by an appeal to Lemma 3.9.

2. Suppose a reduct of $s\theta \approx t\theta$ is rewritten at position 1. Let $l = f(l_1, \ldots, l_n) \to r \Leftarrow c$ be the employed rewrite rule the first time this happens. From Lemma 3.4 we obtain an admissible state $\phi_{[\mathsf{o}]}(\underline{\Pi}) = \langle G'', \theta'', \Pi'', X'' \rangle$ with $G'' = s \approx l, r \approx t, c, H$, $X'' = X$, and $\theta'' = \theta$ $[W]$. According to Lemma 3.3, s cannot be a variable. Hence we may write $s = f(s_1, \ldots, s_n)$. Let $G' = s_1 \approx l_1, \ldots, s_n \approx l_n, r \approx t, c, H$. We have $\Psi_1 \colon G \Rightarrow_{[\mathsf{o}]} G''$. Note that Lemma 3.6 is applicable to $\phi_{[\mathsf{o}]}(\underline{\Pi})$ since by construction no reduct of $s\theta'' \approx l\theta''$ is rewritten at position 1 and 2. This results in an admissible state $\phi_{[\mathsf{d}]}(\phi_{[\mathsf{o}]}(\underline{\Pi})) = \langle G', \theta', \Pi', X' \rangle$ with $\theta' = \theta''$ and $X' = X$. Clearly $\theta' = \theta$ $[W]$. Take $\sigma_1 = \varepsilon$ and $\underline{\Pi}' = \phi_{[\mathsf{d}]}(\phi_{[\mathsf{o}]}(\underline{\Pi}))$.

3. Suppose a reduct of $s\theta \approx t\theta$ is rewritten at position 2. This case reduces to the previous one by an appeal to Lemma 3.9.

In all cases we obtain $\underline{\Pi}'$ from $\underline{\Pi}$ by applying one or two transformation steps $\phi_{[\mathsf{o}]}$, $\phi_{[\mathsf{i}]}$, $\phi_{[\mathsf{d}]}$, $\phi_{[\mathsf{v}]}$, $\phi_{[\mathsf{t}]}$ together with an additional application of ϕ_{swap} in case 1(e) and 3. According to Lemma 3.11 $\underline{\Pi}'$ has smaller complexity than $\underline{\Pi}$. Let $W' = \mathcal{V}\mathrm{ar}_W(\sigma_1) \cup \mathcal{V}\mathrm{ar}(G')$. We have $\mathcal{V}\mathrm{ar}(G') \subseteq W'$ and thus we can apply the induction hypothesis to $\underline{\Pi}'$. This yields an LCNC-refutation $\Psi' \colon G' \Rightarrow^*_{\sigma'} \Box$ respecting $\mathcal{S}_{\mathsf{left}}$ such that $\sigma' \leqslant \theta'$ $[W']$. Define $\sigma = \sigma_1\sigma'$. From $\sigma_1\theta' = \theta$ $[W]$, $\sigma' \leqslant \theta'$ $[W']$, and $\mathcal{V}\mathrm{ar}_W(\sigma_1) \subseteq W'$ we infer that $\sigma \leqslant \theta$ $[W]$. Concatenating the LCNC-step Ψ_1 and the LCNC-refutation Ψ' yields the desired LCNC-refutation Ψ. $\Box$

An immediate corollary of Theorem 3.12 is the completeness of LNC for confluent TRSs and normalized solutions with respect to leftmost selection. We remark that the proof in [26] of this result is considerably more complicated than the proof given above.

4 Strong Completeness

In this section we prove that LCNC is strongly complete whenever basic conditional narrowing is complete. We obtain this result by an inductive transformation process that operates on basic conditional narrowing sequences. To this end we extend the proof of Middeldorp, Okui, and Ida [26] who obtained this result for unconditional TRSs. The structure of the proof is exactly the same as the one in [26, Section 4]. Below we state the relevant lemmata and we present complete proof details of some of the lemmata. The proofs of the other lemmata are straightforward modifications of the corresponding ones in [26]. But first we recall a number of definitions pertaining to (basic) conditional narrowing.

The conditional narrowing calculus CNC consists of the following inference rule:

$$\frac{G', e, G''}{(G', e[r]_p, c, G'')\theta}$$

if there exist a fresh variant $l \to r \Leftarrow c$ of a rewrite rule in $\mathcal{R}$, a non-variable position p in e, and a most general unifier θ of $e_{|p}$ and l.

In the above situation we write $G', e, G'' \rightsquigarrow_\theta (G', e[r]_p, c, G'')\theta$. For a CNC-derivation $\Pi\colon G \rightsquigarrow^*_\theta G'$, $\Pi\theta$ denotes the corresponding rewrite sequence $G\theta \to^* G'$. Conditional narrowing is sound: If $G \rightsquigarrow^*_\theta \top$ then $\theta\restriction_{\mathcal{V}\mathrm{ar}(G)}$ is a solution of G.

A position constraint for a goal G is a mapping that assigns to every equation $e \in G$ a subset of $\mathcal{P}\mathrm{os}_\mathcal{F}(e)$. The position constraint that assigns to every $e \in G$ the set $\mathcal{P}\mathrm{os}_\mathcal{F}(e)$ is denoted by $\bar{G}$. A CNC-derivation Π:

$$G_1 \rightsquigarrow_{\theta_1, e_1, p_1, l_1 \to r_1 \Leftarrow c_1} \cdots \rightsquigarrow_{\theta_{n-1}, e_{n-1}, p_{n-1}, l_{n-1} \to r_{n-1} \Leftarrow c_{n-1}} G_n$$

is based on a position constraint B_1 for G_1 if $p_i \in B_i(e_i)$ for $1 \leqslant i \leqslant n-1$. Here the position constraints $B_2, \dots, B_{n-1}$ for the goals $G_2, \dots, G_{n-1}$ are inductively defined by

$$B_{i+1}(e) = \begin{cases} B_i(e') & \text{if } e' \in G_i \setminus \{e_i\} \\ \mathcal{B}(B_i(e_i), p_i, r_i) & \text{if } e' = e_i[r_i]_{p_i} \\ \mathcal{P}\mathrm{os}_\mathcal{F}(e') & \text{if } e' \in c_i \end{cases}$$

for all $1 \leqslant i < n-1$ and $e = e'\theta_i \in G_{i+1}$, with $\mathcal{B}(B_i(e_i), p_i, r_i)$ abbreviating the set of positions

$$(B_i(e_i) \setminus \{q \in B_i(e_i) \mid q \geqslant p_i\}) \cup \{p_i q \in \mathcal{P}\mathrm{os}_\mathcal{F}(e) \mid q \in \mathcal{P}\mathrm{os}_\mathcal{F}(r_i)\}.$$

We say that Π is basic if it is based on $\bar{G}$. So in a basic CNC-derivation no steps take place at subterms introduced by previous narrowing substitutions. Basic CNC has a much smaller search space than CNC. We refer to Middeldorp and Hamoen [24] for a survey of completeness results for CNC and basic CNC.

Definition 4.1 *Let $G \rightsquigarrow_{\theta,p,l\to r\Leftarrow c} G_1$ be a* CNC*-step and e an equation in G. If e is the selected equation in this step, then e is narrowed into the equation $e[r]_p\theta$ in G_1. In this case we say that $e[r]_p\theta$ is the descendant of e in G_1. Otherwise, e is simply instantiated to the equation $e\theta$ in G_1 and we call $e\theta$ the descendant of e. The notion of descendant extends to* CNC*-derivations in the obvious way.*

Observe that in a CNC-refutation $G \rightsquigarrow^* \top$ every equation $e \in G$ has exactly one descendant true in $\top$, but, unlike the unconditional case, not every true in $\top$ descends from an equation in G.

Lemma 4.2 (12) *Let δ be a variable renaming. For every* CNC*-refutation $\Pi\colon G \rightsquigarrow^+_\theta \top$ there exists a* CNC*-refutation $\varphi_\delta(\Pi)\colon G\delta \rightsquigarrow^+_{\delta^{-1}\theta} \top$.* □

The number between parentheses refers to the corresponding statement in [26].

In the following five lemmata Π denotes a CNC-refutation $G \rightsquigarrow^+_\theta \top$ with $G = G', s \approx t, G''$ such that $s \approx t$ is selected in the first step of Π and W denotes a finite set of variables that includes all variables in the initial goal G of Π. The first transformation lemma depends on the applied rewrite rule. So in the proof we have to take its conditional part into account.

Lemma 4.3 (13) *Suppose narrowing is applied to a descendant of $s \approx t$ in Π at position 1. If $l \to r \Leftarrow c$ is the applied rewrite rule in the first such step then there exists a* CNC*-refutation $\varphi_{[o]}(\Pi)\colon G', s \approx l, r \approx t, c, G'' \rightsquigarrow^*_{\theta_1} \top$ such that $\theta_1 = \theta\ [W]$.*

Proof Write $l = f(l_1,\ldots,l_n)$. The given refutation Π is of the form

$$\begin{aligned} G \ &\rightsquigarrow^*_{\tau_1}\ G'_1, f(u_1,\ldots,u_n) \approx t', G''_1 \ \rightsquigarrow_{\tau_2,1,l\to r\Leftarrow c}\ (G'_1, r \approx t', c, G''_1)\tau_2 \\ &\rightsquigarrow^*_{\tau_3}\ \top \end{aligned}$$

with $\tau_1\tau_2\tau_3 = \theta$. Write $G''_1 = C, G''_2$ such that $s \approx t \rightsquigarrow^* f(u_1,\ldots,u_n) \approx t', C$ and $G'' \rightsquigarrow^* G''_2$. So C consists of all descendants of the (instantiated) equations that appear in the conditional parts of the applied rewrite rules in the derivation from $s \approx t$ to $f(u_1,\ldots,u_n) \approx t'$. The first part of Π can be transformed into

$$G', s \approx l, r \approx t, c, G'' \ \rightsquigarrow^*_{\tau_1}\ G'_1, f(u_1,\ldots,u_n) \approx l, C_1, r \approx t', C_2, c, G''_2$$

where $C_1, C_2 =_\cdot C$. The last part of Π can be rearranged into

$$(G'_1, C_1, r \approx t', C_2, c, G''_2)\tau_2 \ \rightsquigarrow^*_{\tau_3} \ \top.$$

Let x be a fresh variable (so $x \notin W$) and define the substitution υ_2 as the (disjoint) union of τ_2 and $\{x \mapsto l\tau_2\}$. Because υ_2 is a most general unifier of $f(u_1, \ldots, u_n) \approx l$ and $x \approx x$, the CNC-derivation Π can be transformed into the refutation $\varphi_{[o]}(\Pi)$:

$$\begin{array}{rcl} G', s \approx l, r \approx t, c, G'' & \rightsquigarrow^*_{\tau_1} & G'_1, f(u_1, \ldots, u_n) \approx l, C_1, r \approx t', C_2, c, G''_2 \\ & \rightsquigarrow_{\upsilon_2} & (G'_1, \mathtt{true}, C_1, r \approx t', C_2, c, G''_2)\upsilon_2 \\ & = & (G'_1, \mathtt{true}, C_1, r \approx t', C_2, c, G''_2)\tau_2 \\ & \rightsquigarrow^*_{\tau_3} & \top. \end{array}$$

Let $\theta_1 = \tau_1\upsilon_2\tau_3$. We have $\theta_1 = \theta \cup \{x \mapsto l\tau_2\tau_3\}$ and because $x \notin W$ we obtain $\theta_1 = \theta\ [W]$. □

For the tedious proof of the next lemma we refer to the appendix.

Lemma 4.4 (14) *Let $s = f(s_1, \ldots, s_n)$ and $t \in \mathcal{V}$. If $\mathrm{root}(t\theta) = f$ then there exists a* CNC*-refutation $\varphi_{[i]}(\Pi)\colon G\sigma_1 \rightsquigarrow^*_{\theta_1} \top$ such that Π subsumes $\varphi_{[i]}(\Pi)$, $\Pi\theta = \varphi_{[i]}(\Pi)\theta_1$, and $\sigma_1\theta_1 = \theta\ [W]$. Here $\sigma_1 = \{t \mapsto f(x_1, \ldots, x_n)\}$ with $x_1, \ldots, x_n \notin W$.* □

Lemma 4.5 (15) *Let $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$, and suppose that narrowing is never applied to a descendant of $s \approx t$ in Π at position 1 or 2. There exists a* CNC*-refutation $\varphi_{[d]}(\Pi)\colon G', s_1 \approx t_1, \ldots, s_n \approx t_n, G'' \rightsquigarrow^*_{\theta_1} \top$ such that $\theta_1 \leqslant \theta\ [W]$.*

Proof The given refutation Π must be of the form

$$G \ \rightsquigarrow^*_{\tau_1} \ G'_1, s' \approx t', G''_1 \ \rightsquigarrow_{\tau_2, \epsilon} \ (G'_1, \mathtt{true}, G''_1)\tau_2 \ \rightsquigarrow^*_{\tau_3} \ \top$$

with $s' = f(s'_1, \ldots, s'_n)$, $t' = f(t'_1, \ldots, t'_n)$, and $\tau_1\tau_2\tau_3 = \theta$. Write $G''_1 = C, G''_2$ such that $s \approx t \rightsquigarrow^* \mathtt{true}, C$ (and thus $G'' \rightsquigarrow^* G''_2$). The first part of Π can be transformed into Π_1:

$$G', s_1 \approx t_1, \ldots, s_n \approx t_n, G'' \ \rightsquigarrow^*_{\tau_1} \ G'_1, s'_1 \approx t'_1, C_1, \ldots, s'_n \approx t'_n, C_n, G''_2.$$

Here $C_1, \ldots, C_n$ and C consists of the same equations in possibly different order. Consider the step from $G'_1, s' \approx t', G''_1$ to $(G'_1, \mathtt{true}, G''_1)\tau_2$. Let $x \approx$

$x \to \mathtt{true}$ be the employed rewrite rule, so τ_2 is a most general unifier of $x \approx x$ and $s' \approx t'$. There clearly exists a rewrite sequence

$$\begin{aligned}&(G_1', s_1' \approx t_1', C_1, \dots, s_n' \approx t_n', C_n, G_2'')\tau_2\\ &\quad \to_\epsilon^* \ (G_1', \mathtt{true}, C_1, \dots, \mathtt{true}, C_n, G_2'')\tau_2.\end{aligned}$$

Since all steps in this rewrite sequence take place at root positions using the unconditional rewrite rule $x \approx x \to \mathtt{true}$, lifting can be applied, resulting in a CNC-derivation Π_2:

$$\begin{aligned}&G_1', s_1' \approx t_1', C_1, \dots, s_n' \approx t_n', C_n, G_2''\\ &\quad \rightsquigarrow_{\upsilon_2,\epsilon}^* \ (G_1', \mathtt{true}, C_1, \dots, \mathtt{true}, C_n, G_2'')\upsilon_2\end{aligned}$$

such that $\upsilon_2 \leqslant \tau_2 \ [W \cup \mathcal{I}(\tau_1)]$. We distinguish two cases.

1. Suppose $G_1', G_1'' = \square$. In this case $\tau_3 = \varepsilon$. We simply define $\varphi_{[d]}(\Pi) = \Pi_1; \Pi_2$. Note that $(G_1', \mathtt{true}, C_1, \dots, \mathtt{true}, C_n, G_2'')\upsilon_2 = \top$. Let $\theta_1 = \tau_1\upsilon_2$. From $\upsilon_2 \leqslant \tau_2 \ [W \cup \mathcal{I}(\tau_1)]$ we infer that $\theta_1 \leqslant \tau_1\tau_2 = \theta \ [W]$.

2. The case $G_1', G_1'' \neq \square$ is more involved. First observe that υ_2 is a unifier of s' and t'. Using the fact that τ_2 is a most general unifier of $s' \approx t'$ and $x \approx x$, it is not difficult to show that $\tau_2 \leqslant \upsilon_2 \ [\mathcal{V} \setminus \{x\}]$. Since $x \notin W \cup \mathcal{I}(\tau_1)$ we have in particular $\tau_2 \leqslant \upsilon_2 \ [W \cup \mathcal{I}(\tau_1)]$. It follows that there exists a variable renaming δ such that $\upsilon_2 = \tau_2\delta \ [W \cup \mathcal{I}(\tau_1)]$. Clearly $\mathcal{V}\mathrm{ar}(G_1', G_1'') \subseteq W \cup \mathcal{I}(\tau_1)$. The last part of Π can be transformed (by changing the number of occurrences of $\mathtt{true}$ as well as the order of the equations in each goal) into

$$\Pi_3\colon (G_1', \mathtt{true}, C_1, \dots, \mathtt{true}, C_n, G_2'')\tau_2 \ \rightsquigarrow_{\tau_3}^+ \ \top.$$

An application of Lemma 4.2 results in the CNC-refutation

$$\varphi_\delta(\Pi_3)\colon (G_1', \mathtt{true}, C_1, \dots, \mathtt{true}, C_n, G_2'')\upsilon_2 \ \rightsquigarrow_{\delta^{-1}\tau_3}^+ \ \top.$$

Define $\varphi_{[d]}(\Pi) = \Pi_1; \Pi_2; \varphi_\delta(\Pi_3)$. Let $\theta_1 = \tau_1\upsilon_2\delta^{-1}\tau_3$. We have $\theta_1 = \tau_1\tau_2\tau_3 = \theta \ [W]$.

$\square$

The proofs of the two remaining transformation lemmata are exactly the same as the unconditional ones.

Lemma 4.6 (16) *Let $t \in \mathcal{V}$, $t \notin \mathcal{V}\mathrm{ar}(s)$, and suppose that the first step of* Π *takes place at the root position. There exists a* CNC*-refutation* $\varphi_{[\mathrm{v}]}(\Pi)$: $(G', G'')\sigma_1 \leadsto^*_{\theta_1} \top$ *with* $\sigma_1 = \{t \mapsto s\}$ *such that* $\sigma_1\theta_1 \leqslant \theta\ [W]$. □

Lemma 4.7 (17) *Let $t \in \mathcal{V}$, $s = t$, and suppose that the first step of* Π *takes place at the root position. There exists a* CNC*-refutation* $\varphi_{[\mathrm{t}]}(\Pi)$: $G', G'' \leadsto^*_{\theta_1} \top$ *such that* $\theta_1 \leqslant \theta\ [W]$. □

Definition 4.8 *The complexity* $|\Pi|$ *of a* CNC*-refutation* $\Pi\colon G \leadsto^*_\theta \top$ *and the well-founded order* $\gg$ *on* CNC*-refutations are defined as in Definition 3.10. The only difference is that in part (1) of the complexity measure we count the number of* narrowing *steps at non-root positions in* Π.

Lemma 4.9 (20) *Let* Π *be a* CNC*-refutation and* $\alpha \in \{\mathrm{o}, \mathrm{i}, \mathrm{d}, \mathrm{v}, \mathrm{t}\}$. *We have* $\Pi \gg \varphi_{[\alpha]}(\Pi)$ *whenever* $\varphi_{[\alpha]}(\Pi)$ *is defined.* □

Lemma 4.10 (21) *For every* CNC*-refutation* $\Pi\colon G', s \approx t, G'' \leadsto^*_\theta \top$ *there exists a* CNC*-refutation* $\varphi_{\mathrm{swap}}(\Pi)\colon G', t \approx s, G'' \leadsto^*_\theta \top$ *with the same complexity.* □

Lemma 4.11 (24) *Let* Π *be a basic* CNC*-refutation and* $\alpha \in \{\mathrm{o}, \mathrm{i}, \mathrm{d}, \mathrm{v}, \mathrm{t}\}$. *The* CNC*-refutation* $\varphi_{[\alpha]}(\Pi)$ *is basic whenever it is defined.* □

Lemma 4.12 is the key lemma. It can be diagrammatically depicted as follows:

$$\begin{array}{llllll} \forall & \text{basic} & \Pi & : & G & \leadsto^+_\theta \ \top \\ \exists & & \Psi_1 & : & \Downarrow_{\sigma_1} & \\ \exists & \text{basic} & \Pi_1 & : & G_1 & \leadsto^*_{\theta_1} \ \top \end{array} \quad \text{such that} \quad \left\{ \begin{array}{l} \sigma_1\theta_1 \leqslant \theta\ [W] \\ \Pi \gg \Pi_1 \end{array} \right.$$

Although the proof is very similar to that of Lemma 25 in [26], we present the complete proof in order to show how the previous lemmata are used.

Lemma 4.12 (25) *For every basic* CNC*-refutation* $\Pi\colon G \leadsto^+_\theta \top$ *there exist an* LCNC*-step* $\Psi_1\colon G \Rightarrow_{\sigma_1} G_1$ *and a basic* CNC*-refutation* $\Pi_1\colon G_1 \leadsto^*_{\theta_1} \top$ *such that* $\sigma_1\theta_1 \leqslant \theta\ [W]$, $\Pi \gg \Pi_1$, *and the equation selected in the first step of* Π *is selected in* Ψ_1.

Proof We distinguish the following cases, depending on what happens to the selected equation $e = s \approx t$ in the first step of Π. Let $G = G', e, G''$.

1. Suppose narrowing is never applied to a descendant of $s \approx t$ at position 1 or 2. We distinguish four further cases.

   (a) Suppose $s, t \notin \mathcal{V}$. We may write $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$. Let $G_1 = G', s_1 \approx t_1, \ldots, s_n \approx t_n, G''$. We have $\Psi_1 \colon G \Rightarrow_{[\mathsf{d}]} G_1$. Lemma 4.5 yields a CNC-refutation $\varphi_{[\mathsf{d}]}(\Pi) \colon G_1 \rightsquigarrow^*_{\theta_1} \top$ such that $\theta_1 \leqslant \theta\ [W]$. Take $\sigma_1 = \varepsilon$.

   (b) Suppose $t \in \mathcal{V}$ and $s = t$. In this case the first step of Π_1 must take place at the root of e. Let $G_1 = G', G''$. We have $\Psi_1 \colon G \Rightarrow_{[\mathsf{t}]} G_1$. Lemma 4.7 yields a CNC-refutation $\varphi_{[\mathsf{t}]}(\Pi) \colon G_1 \rightsquigarrow^*_{\theta_1} \top$ such that $\theta_1 \leqslant \theta\ [W]$. Take $\sigma_1 = \varepsilon$.

   (c) Suppose $t \in \mathcal{V}$ and $s \neq t$. We distinguish two further cases, depending on what happens to e in the first step of Π.

      i. Suppose narrowing is applied to e at the root position. Let $\sigma_1 = \{t \mapsto s\}$ and $G_1 = (G', G'')\sigma_1$. We have $\Psi_1 \colon G \Rightarrow_{[\mathsf{v}], \sigma_1} G_1$. Lemma 4.6 yields a CNC-refutation $\varphi_{[\mathsf{v}]}(\Pi) \colon G_1 \rightsquigarrow^*_{\theta_1} \top$ such that $\sigma_1\theta_1 \leqslant \theta\ [W]$.

      ii. Suppose narrowing is not applied to e at the root position. This implies that $s \notin \mathcal{V}$. Hence we may write $s = f(s_1, \ldots, s_n)$. Let $\sigma_1 = \{t \mapsto f(x_1, \ldots, x_n)\}$, $G_1 = (G', s_1 \approx x_1, \ldots, s_n \approx x_n, G'')\sigma_1$, and $G_2 = G\sigma_1$. Here $x_1, \ldots, x_n$ are fresh variables. We have $\Psi_1 \colon G \Rightarrow_{[\mathsf{i}], \sigma_1} G_1$. From Lemma 4.4 we obtain a CNC-refutation $\Pi_2 = \varphi_{[\mathsf{i}]}(\Pi) \colon G_2 \rightsquigarrow^*_{\theta_2} \top$ such that $\sigma_1\theta_2 = \theta\ [W]$. Let $W' = W \cup \{x_1, \ldots, x_n\}$. Clearly $\mathcal{V}\mathrm{ar}(G_2) \subseteq W'$. An application of Lemma 4.5 to Π_2 results in a CNC-refutation $\Pi_1 = \varphi_{[\mathsf{d}]}(\Pi_2) \colon G_1 \rightsquigarrow^*_{\theta_1} \top$ such that $\theta_1 \leqslant \theta_2\ [W']$. Using the inclusion $\mathcal{V}\mathrm{ar}_W(\sigma_1) \subseteq W'$ we obtain $\sigma_1\theta_1 \leqslant \sigma_1\theta_2 = \theta\ [W]$.

   (d) In the remaining case we have $t \notin \mathcal{V}$ and $s \in \mathcal{V}$. This case reduces to case 1(c) by an appeal to Lemma 4.10.

2. Suppose narrowing is applied to a descendant of e at position 1. Let $l = f(l_1, \ldots, l_n) \to r \Leftarrow c$ be the used rewrite rule the first time this happens. Because Π is basic, s cannot be a variable, for otherwise narrowing would be applied to a subterm introduced by previous narrowing substitutions. Hence we may write $s = f(s_1, \ldots, s_n)$. Let $G_1 = G', s_1 \approx l_1, \ldots, s_n \approx l_n, r \approx t, c, G''$ and $G_2 = G', s \approx l, r \approx t, c, G''$. We have $\Psi_1 \colon G \Rightarrow_{[\mathsf{o}]} G_1$. From Lemma 4.3 we obtain a

CNC-refutation $\Pi_2 = \varphi_{[o]}(\Pi)\colon G_2 \rightsquigarrow^*_{\theta_2} \top$ such that $\theta_2 = \theta\ [W]$. Let $W' = W \cup \mathcal{V}\mathrm{ar}(l \to r \Leftarrow c)$. Clearly $\mathcal{V}\mathrm{ar}(G_2) \subseteq W'$. An application of Lemma 4.5 to Π_2 results in a CNC-refutation $\Pi_1 = \varphi_{[d]}(\Pi_2)\colon G_1 \rightsquigarrow^*_{\theta_1} \top$ such that $\theta_1 \leqslant \theta_2\ [W']$. Using $W \subseteq W'$ we obtain $\theta_1 \leqslant \theta\ [W]$. Take $\sigma_1 = \varepsilon$.

3. Suppose narrowing is applied to a descendant of e at position 2. This case reduces to the previous one by an appeal to Lemma 4.10.

In all cases we obtain Π_1 from Π by applying one or two transformation steps $\varphi_{[o]}$, $\varphi_{[i]}$, $\varphi_{[d]}$, $\varphi_{[v]}$, $\varphi_{[t]}$ together with an additional application of φ_{swap} in case 1(d) and (3). According to Lemma 4.11 Π_1 is basic. According to Lemmata 4.9 and 4.10 Π_1 has smaller complexity than Π. ☐

The proof of the following switching lemma can be found in the appendix.

Lemma 4.13 (26) *Let G_1 be a goal containing distinct equations e_1 and e_2. For every* CNC*-derivation $G_1 \rightsquigarrow_{\tau_1, e_1, p_1, l_1 \to r_1 \Leftarrow c_1} G_2 \rightsquigarrow_{\tau_2, e_2\tau_1, p_2, l_2 \to r_2 \Leftarrow c_2} G_3$ with $p_2 \in \mathcal{P}\mathrm{os}_{\mathcal{F}}(e_2)$ there exists a* CNC*-derivation $G_1 \rightsquigarrow_{\upsilon_2, e_2, p_2, l_2 \to r_2 \Leftarrow c_2} H_2 \rightsquigarrow_{\upsilon_1, e_1\upsilon_2, p_1, l_1 \to r_1 \Leftarrow c_1} H_3$ such that $G_3 = H_3$ and $\tau_1\tau_2 = \upsilon_2\upsilon_1$.* ☐

Lemma 4.14 (27) *Let S be an arbitrary selection function. For every basic* CNC*-refutation $\Pi\colon G \rightsquigarrow^*_\theta \top$ there exists a basic* CNC*-refutation $\varphi_S(\Pi)\colon G \rightsquigarrow^*_\theta \top$ respecting S with the same complexity.* ☐

Theorem 4.15 (28) *Let $\mathcal{R}$ be an arbitrary CTRS and $\Pi\colon G \rightsquigarrow^*_\theta \top$ a basic* CNC*-refutation. For every selection function S there exists an* LCNC*-refutation $\Psi\colon G \Rightarrow^*_\sigma \square$ respecting S such that $\sigma \leqslant \theta\ [\mathcal{V}\mathrm{ar}(G)]$.*

Proof We use well-founded induction on the complexity of the given basic CNC-refutation Π. In order to make the induction work we prove $\sigma \leqslant \theta\ [W]$ for a finite set of variables W that includes $\mathcal{V}\mathrm{ar}(G)$ instead of $\sigma \leqslant \theta\ [\mathcal{V}\mathrm{ar}(G)]$. The base case is trivial: G must be the empty goal. For the induction step we proceed as follows. First we use Lemma 4.14 to transform Π into a basic CNC-refutation $\varphi_S(\Pi)\colon G \rightsquigarrow^+_\theta \top$ respecting S with equal complexity. According to Lemma 4.12 there exist an LCNC-step $\Psi_1\colon G \Rightarrow_{\sigma_1} G_1$ respecting S and a basic CNC-refutation $\Pi_1\colon G_1 \rightsquigarrow^*_{\theta_1} \top$ such that $\sigma_1\theta_1 \leqslant \theta\ [W]$ and $\varphi_S(\Pi) \gg \Pi_1$. Let $W' = \mathcal{V}\mathrm{ar}_W(\sigma_1) \cup \mathcal{V}\mathrm{ar}(G_1)$. Clearly W' is a finite set of variables that includes $\mathcal{V}\mathrm{ar}(G_1)$. The induction hypothesis yields an LCNC-refutation $\Psi'\colon G_1 \Rightarrow^*_{\sigma'} \square$ respecting S such that $\sigma' \leqslant \theta_1\ [W']$. Now define

$\sigma = \sigma_1\sigma'$. From $\sigma_1\theta_1 \leqslant \theta\ [W]$, $\sigma' \leqslant \theta_1\ [W']$, and $\mathcal{V}\mathrm{ar}_W(\sigma_1) \subseteq W'$, we infer that $\sigma \leqslant \theta\ [W]$ and thus also $\sigma \leqslant \theta\ [\mathcal{V}\mathrm{ar}(G)]$. Concatenating the LCNC-step Ψ_1 and the LCNC-refutation Ψ' yields the desired LCNC-refutation Ψ. $\square$

Since basic conditional narrowing is known to be complete for decreasing and confluent CTRSs ([24]), level-complete 2-CTRSs ([9, 24]), and terminating and shallow-confluent normal 3-CTRSs ([33, Satz 9.7]), we obtain as corollary the strong completeness of LCNC for these three classes of CTRSs.

Let us illustrate the above proof on a small example.

Example 4.16 *Consider the CTRS $\mathcal{R}$ consisting of the following rewrite rules:*

$$\begin{array}{rcl} \mathsf{f}(\mathsf{g}(x)) & \to & x \ \Leftarrow\ x \approx \mathsf{b} \\ \mathsf{a} & \to & \mathsf{b} \end{array}$$

This CTRS is easily seen to be decreasing (take e.g. the recursive path order with precedence $\mathsf{f} > \mathsf{b}$ *and* $\mathsf{a} > \mathsf{b}$*) and confluent. Consider the goal* $G = \mathsf{g}(\mathsf{f}(x)) \approx x$*. The substitution* $\theta = \{x \mapsto \mathsf{g}(\mathsf{a})\}$ *is a solution of* G *because we have the rewrite sequence*

$$G\theta = \mathsf{g}(\mathsf{f}(\mathsf{g}(\mathsf{a}))) \approx \mathsf{g}(\mathsf{a}) \ \to\ \mathsf{g}(\mathsf{a}) \approx \mathsf{g}(\mathsf{a}) \ \to\ \mathtt{true}.$$

Basic conditional narrowing computes the solution $\{x \mapsto \mathsf{g}(\mathsf{b})\}$ *which is different from* θ *but equal to* θ *modulo (the equational theory induced by)* $\mathcal{R}$*:*

$$\begin{array}{l} \mathsf{g}(\underline{\mathsf{f}(x)}) \approx x \ \rightsquigarrow_{\{x \mapsto \mathsf{g}(x_1)\}}\ \underline{\mathsf{g}(x_1) \approx \mathsf{g}(x_1)},\ x_1 \approx \mathsf{b} \ \rightsquigarrow\ \mathtt{true},\ \underline{x_1 \approx \mathsf{b}} \\ \qquad\qquad \rightsquigarrow_{\{x_1 \mapsto \mathsf{b}\}}\ \top \end{array}$$

Starting from this basic narrowing refutation, the above proof yields the following LCNC*-refutation; the numbers on the right refer to the various cases* *in the proof:*

$$\begin{array}{lr}
\mathsf{g}(\mathsf{f}(x)) \approx x & \\
\quad \Downarrow_{[\mathrm{i}]},\ \{x \mapsto \mathsf{g}(x_1)\} & \mathit{1(c)ii} \\
\mathsf{f}(\mathsf{g}(x_1)) \approx x_1 & \\
\quad \Downarrow_{[\mathrm{o}]},\ \mathsf{f}(\mathsf{g}(x_2)) \to x_2 \Leftarrow x_2 \approx \mathsf{b} & \mathit{2} \\
\mathsf{g}(x_1) \approx \mathsf{g}(x_2),\ x_2 \approx x_1,\ x_2 \approx \mathsf{b} & \\
\quad \Downarrow_{[\mathrm{d}]} & \mathit{1(a)} \\
x_1 \approx x_2,\ x_2 \approx x_1,\ x_2 \approx \mathsf{b} & \\
\quad \Downarrow_{[\mathrm{v}]},\ \{x_1 \mapsto x_2\} & \mathit{1(c)i} \\
x_2 \approx x_2,\ x_2 \approx \mathsf{b} & \\
\quad \Downarrow_{[\mathrm{t}]} & \mathit{1(b)} \\
x_2 \approx \mathsf{b} & \\
\quad \Downarrow_{[\mathrm{v}]},\ \{x_2 \mapsto \mathsf{b}\} & \mathit{1(c)i} \\
\quad \square &
\end{array}$$

In every step of this refutation the leftmost equation is selected.

The following variant of a well-known example due to Giovannetti and Moiso [9] shows that LCNC is not (strongly) complete for terminating and confluent 2-CTRSs, even under the additional normality restriction.

Example 4.17 *Consider the 2-CTRS consisting of the following rewrite rules:*

$$\begin{array}{rcll}
\mathsf{a} & \to & \mathsf{b} & \\
\mathsf{a} & \to & \mathsf{c} & \\
\mathsf{b} & \to & \mathsf{c} & \Leftarrow\ \mathsf{f}(x,\mathsf{b}) \approx \mathsf{d},\ \mathsf{f}(x,\mathsf{c}) \approx \mathsf{d} \\
\mathsf{f}(\mathsf{b},\mathsf{b}) & \to & \mathsf{d} & \\
\mathsf{f}(\mathsf{c},\mathsf{c}) & \to & \mathsf{d} &
\end{array}$$

Confluence and termination of $\mathcal{R}$ are easily shown. The goal $\mathsf{b} \approx \mathsf{c}$ *admits the empty substitution as solution but one easily shows that* LCNC *cannot solve the goal* $\mathsf{b} \approx \mathsf{c}$.

It is interesting to note that the previous first-order example refutes the completeness of Prehofer's conditional lazy narrowing calculus CLN for weakly normalizing and confluent higher-order normal CTRSs [27, Theorem 3.2].[1]

5 Level-Complete CTRSs

In this section we present our third and final completeness result, for the class of level-complete conditional (3-)CTRSs. Note that this class is not covered by the results of the preceding section because of the incompleteness of basic conditional narrowing (see Example 5.4 below).

Definition 5.1 *Let $\mathcal{R}$ be a CTRS and G a goal with solution θ. The level* $\mathsf{level}(e)$ *of an equation $e \in G\theta$ is the smallest n such that $e \to_n^* \mathtt{true}$. We say that the rewrite sequence $G\theta \to_{\mathcal{R}}^* \top$ is* level-minimal *if the depths of its rewrite steps do not exceed the level of the originating equations. Clearly, every rewrite sequence $G\theta \to_{\mathcal{R}}^* \top$ can be transformed into a level-minimal one (which may be longer than the given sequence).*

We use induction with respect to the well-founded order on rewrite sequences defined below.

Definition 5.2 *Let $\mathcal{R}$ be a level-terminating CTRS. For every $n \geqslant 0$, let $\mathcal{G}_n = \{G \mid \mathcal{R}_n \vdash G\}$ be the set of all goals G whose level does not exceed n. Let G be a goal and θ a substitution such that $G\theta \in \mathcal{G}_n$. With every equation $e = s \approx t \in G\theta$ we associate the pair $|e| = (\mathsf{level}(e), \{s,t\})$ whose second component is the multiset which contains both sides of e. We equip the set of these pairs with the order $\sqsupset^n = \mathsf{lex}(>, \succ^n_{\mathsf{mul}})$ where $\succ^n = (\to_n \cup \rhd)^+$. The multiset consisting of all $|e\theta|$ for $e \in G$ is denoted by $(G;\theta)$. Let $\Pi\colon G \to^* H$ and $\Pi'\colon G' \to^* H'$ be rewrite sequences such that $G \in \mathcal{G}_n$. We write $\Pi \gg \Pi'$ if $(G;\varepsilon) \sqsupset^n_{\mathsf{mul}} (G';\varepsilon)$. Note that this implies that $G' \in \mathcal{G}_n$.*

Since $\to_n$ is closed under contexts, the relation $\succ^n$ is a well-founded order for every level-terminating CTRS $\mathcal{R}$ and every $n \geqslant 0$. As in the previous section, it follows that $\gg$ is a well-founded order on rewrite sequences. Note that this order depends only on the initial goals of rewrite sequences. In the proof below we make use of the well-known equivalence of $M \succ_{\mathsf{mul}} N$ and $M - N \succ_{\mathsf{mul}} N - M$.

[1]Cf. also [28, Theorem 6.8.9].

Theorem 5.3 *Let $\mathcal{R}$ be a terminating and level-confluent CTRS. For every solution θ of a goal G there exists an* LCNC*-refutation $G \Rightarrow^*_\sigma \square$ such that $\sigma \leqslant_{\mathcal{R}} \theta$ $[\mathcal{V}\mathrm{ar}(G)]$.*

Proof We use well-founded induction with respect the well-founded order $\gg$ on rewrite sequences. Let Π: $G\theta \rightarrow^* \top$ be any rewrite sequence from $G\theta$ to $\top$ and let κ be the level of Π. In order to make the induction work we prove $\sigma \leqslant_{\mathcal{R}} \theta$ $[W]$ for a finite set of variables W that includes $\mathcal{V}\mathrm{ar}(G)$. The base case is trivial since G must be the empty goal (and thus we can take $\sigma = \varepsilon$). For the induction step we proceed as follows. First we transform Π into a level-minimal rewrite sequence. (This does not affect the complexity $(G;\theta)$.) By rearranging the order of the equations in Π, we can assume that the level of the leftmost equation in $G\theta$ is minimal. Write $G = e, H$ and $e = s \approx t$. Since it does not effect level-minimality, we may swap rewrite steps that take place in different sides of $e\theta$ at will. This will simplify the notation in some of the cases below. For the same reason we may assume that in Π always the leftmost equation different from `true` is rewritten. Let $\ell = \mathsf{level}(e\theta)$. So $\kappa \geqslant \mathsf{level}(e'\theta) \geqslant \ell$ for all $e' \in H$. We will show the existence of an LCNC-step Ψ_1: $G \Rightarrow_{\sigma_1} G_1$ and a rewrite sequence Π_1: $G_1\theta_1 \rightarrow^* \top$ such that $\sigma_1\theta_1 \leqslant_{\mathcal{R}} \theta$ $[W]$ and $\Pi \gg \Pi_1$. We distinguish the following cases.

1. Suppose $s, t \notin \mathcal{V}$. We distinguish two further cases, depending on what happens to $e\theta$ in Π.

   (a) Suppose no reduct of $e\theta$ is rewritten at position 1 or 2. We may write $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$. For $1 \leqslant i \leqslant n$ let e_i be the equation $s_i \approx t_i$. Let $G_1 = e_1, \ldots, e_n, H$. We have Ψ_1: $G \Rightarrow_{[\mathsf{d}]} G_1$. Let $\sigma_1 = \varepsilon$ and $\theta_1 = \theta$. The rewrite sequence Π, which must be of the form

   $$G\theta \rightarrow^*_\ell f(u_1, \ldots, u_n) \approx f(u_1, \ldots, u_n), H\theta \rightarrow \mathtt{true}, H\theta \rightarrow^* \top,$$

   can be transformed into Π_1:

   $$G_1\theta_1 \rightarrow^* u_1 \approx u_1, \ldots, u_n \approx u_n, H\theta \rightarrow^* \top, H\theta \rightarrow^* \top.$$

   Clearly $\sigma_1\theta_1 = \theta$. It is also clear that $G_1\theta_1 \in \mathcal{G}_\kappa$. It remains to show that $\Pi \gg \Pi_1$. We have $(G;\theta) - (G_1;\theta_1) = \{|e\theta|\}$ and $(G_1;\theta_1) - (G;\theta) = \{|e_1\theta|, \ldots, |e_n\theta|\}$. For all $1 \leqslant i \leqslant n$ we have $s\theta \rhd s_i\theta$ and $t\theta \rhd t_i\theta$ and thus also $s\theta \succ^\kappa s_i\theta$ and $t\theta \succ^\kappa t_i\theta$.

Hence $\{s\theta, t\theta\} \succ^{\kappa}_{\mathsf{mul}} \{s_i\theta, t_i\theta\}$ and therefore it suffices to show that $\mathsf{level}(e\theta) \geqslant \mathsf{level}(e_i\theta)$. We have $s_i\theta \rightarrow^*_\ell u_i$ and $t_i\theta \rightarrow^*_\ell u_i$. Therefore $\mathcal{R}_\ell \vdash e_i\theta$ and hence the level of $e_i\theta$ does not exceed the level of $e\theta$. We conclude that $\Pi \gg \Pi_1$.

(b) Suppose a reduct of $e\theta$ is rewritten at position 1 or 2. Without loss of generality we assume that the first such reduct is rewritten at position 1, using the fresh variant $l \rightarrow r \Leftarrow c$ of a rewrite rule in $\mathcal{R}$ with $l = f(l_1, \ldots, l_n)$. We have $s = f(s_1, \ldots, s_n)$. The rewrite sequence Π is of the form

$$G\theta \rightarrow^*_\ell f(s'_1, \ldots, s'_n) \approx t\theta, H\theta \rightarrow_\ell r\tau \approx t\theta, H\theta \rightarrow^* \top$$

with $f(s'_1, \ldots, s'_n) = l\tau$ and $c\tau \rightarrow^*_{\ell-1} \top$. For $1 \leqslant i \leqslant n$ let $e_i = s_i \approx l_i$ and define $G_1 = e_1, \ldots, e_n, r \approx t, c, H$. We have $\Psi_1 \colon G \Rightarrow_{[\mathsf{o}]} G_1$. Let $\sigma_1 = \varepsilon$ and define θ_1 as the disjoint union of θ and τ. Because

$$G_1\theta_1 = s_1\theta \approx l_1\tau, \ldots, s_n\theta \approx l_n\tau, r\tau \approx t\theta, c\tau, H\theta$$

we obtain the rewrite sequence Π_1:

$$\begin{aligned} G_1\theta_1 &\rightarrow^*_\ell s'_1 \approx l_1\tau, \ldots, s'_n \approx l_n\tau, r\tau \approx t\theta, c\tau, H\theta \\ &\rightarrow^* \top, r\tau \approx t\theta, c\tau, H \rightarrow^* \top. \end{aligned}$$

Since $W \cap \mathcal{D}(\tau) = \varnothing$ we have $\sigma_1\theta_1 = \theta\ [W]$. In order to conclude that $\Pi \gg \Pi_1$ we have to show that $(G;\theta) - (G_1;\theta_1) = \{|e\theta|\} \sqsupset^{\kappa}_{\mathsf{mul}} \{|e_1\theta_1|, \ldots, |e_n\theta_1|, |r\tau \approx t\theta|\} \uplus (c;\tau) = (G_1;\theta_1) - (G;\theta)$. For all $1 \leqslant i \leqslant n$ we have $s\theta \rhd s_i\theta \rightarrow^*_\ell s'_i = l_i\tau$ and consequently $\{s\theta, t\theta\} \succ^{\kappa}_{\mathsf{mul}} \{s_i\theta, l_i\tau\}$. Furthermore, the level of $e_i\theta_1$ does not exceed ℓ. Hence $|e\theta| \sqsupset^{\kappa} |e_1\theta_1|, \ldots, |e_n\theta_1|$. Next consider the equation $r\tau \approx t\theta$. Since $r\tau \approx t\theta \rightarrow^*_\ell \mathtt{true}$, the level of $r\tau \approx t\theta$ is at most ℓ. Also, $s\theta \rightarrow^*_\ell l\tau \rightarrow_\ell r\tau$ and thus $\{s\theta, t\theta\} \succ^{\kappa}_{\mathsf{mul}} \{r\tau, t\theta\}$. Hence $|e\theta| \sqsupset^{\kappa} |r\tau \approx t\theta|$. Finally, from $c\tau \rightarrow_{\ell-1} \top$ we infer that the level of every equation in $c\tau$ is less than ℓ. Therefore also $\{|e\theta|\} \sqsupset^{\kappa}_{\mathsf{mul}} (c;\tau)$.

2. Suppose $s \notin \mathcal{V}$ and $t \in \mathcal{V}$. We distinguish two further cases.

(a) Suppose no reduct of $e\theta$ is rewritten at position 1. Write $s = f(s_1, \ldots, s_n)$. Let $\sigma_1 = \{t \mapsto f(x_1, \ldots, x_n)\}$ and $e_i = x_i \approx s_i$

for $1 \leqslant i \leqslant n$. Here $x_1, \ldots, x_n$ are fresh variables. Define $G_1 = (e_1, \ldots, e_n, H)\sigma_1$. We have $\Psi_1 \colon G \Rightarrow_{[i],\sigma_1} G_1$. The rewrite sequence Π is of the form

$$G\theta \to_\ell^* f(s_1', \ldots, s_n') \approx f(s_1', \ldots, s_n'), H\theta \to \mathtt{true}, H\theta \to^* \top.$$

Define θ_1 as the disjoint union of θ and $\{x_1 \mapsto s_1', \ldots, x_n \mapsto s_n'\}$. We have

$$G_1\theta_1 = s_1' \approx s_1\sigma_1\theta_1, \ldots, s_n' \approx s_n\sigma_1\theta_1, H\sigma_1\theta_1.$$

Because $t\theta \to_\ell^* f(s_1', \ldots, s_n') = f(x_1, \ldots, x_n)\theta_1 = t\sigma_1\theta_1$ and $x\theta = x\sigma_1\theta_1$ for variables x different from $x_1, \ldots, x_n, t$, we have $x\theta \to_\ell^* x\sigma_1\theta_1$ for all variables $x \in W$. Hence $\sigma_1\theta_1 =_\mathcal{R} \theta\ [W]$, $H\theta \to_\ell^* H\sigma_1\theta_1$, and $s_i\sigma_1\theta_1 \,{}_\ell^*\!\leftarrow s_i\theta \to_\ell^* s_i'$ for all $1 \leqslant i \leqslant n$. Since the level of every equation in $H\theta$ is at least ℓ, level-confluence yields $H\sigma_1\theta_1 \to^* \top$ such that for every equation $e' \in H$ the level of $e'\sigma_1\theta_1$ is less than or equal to the level of $e'\theta$. In addition, we obtain terms $u_1, \ldots, u_n$ such that $s_i\sigma_1\theta_1 \to_\ell^* u_i \,{}_\ell^*\!\leftarrow s_i'$ for all $1 \leqslant i \leqslant n$. Hence we obtain the rewrite sequence Π_1:

$$G_1\theta_1 \to_\ell^* u_1 \approx u_1, \ldots, u_n \approx u_n, H\sigma_1\theta_1 \to^* \top.$$

We show that $\Pi \gg \Pi_1$. We have $(G; \theta) = \{|e\theta|\} \uplus (H; \theta)$ and $(G_1; \theta_1) = \{|e_1\theta_1|, \ldots, |e_n\theta_1|\} \uplus (H\sigma_1; \theta_1)$. First we show that $|e\theta| \sqsupset^\kappa |e_1\theta_1|, \ldots, |e_n\theta_1|$. For all $1 \leqslant i \leqslant n$ we have $s\theta \rhd s_i\theta \to_\ell^* s_i'$ and $s\theta \rhd s_i\theta \to_\ell^* s_i\sigma_1\theta_1$ and thus also $s\theta \succ^\ell s_i'$ and $s\theta \succ^\ell s_i\sigma_1\theta_1$. Since $\succ^\ell \subseteq \succ^\kappa$ we obtain $\{s\theta, t\theta\} \succ_{\mathsf{mul}}^\kappa \{s_i', s_i\sigma_1\theta_1\}$. From $s_i\sigma_1\theta_1 \to_\ell^* u_i \,{}_\ell^*\!\leftarrow s_i'$ we infer that the level of $e_i\theta_1$ does not exceed ℓ and therefore $|e\theta| \sqsupset^\kappa |e_1\theta_1|, \ldots, |e_n\theta_1|$. To conclude that $\Pi \gg \Pi_1$ it now suffices to show that $(H; \theta) \sqsupseteq_{\mathsf{mul}}^\kappa (H\sigma_1; \theta_1)$. Here $\sqsupseteq_{\mathsf{mul}}^\kappa$ denotes the reflexive closure of $\sqsupset_{\mathsf{mul}}^\kappa$. Let $e' = u \approx v \in H$. We already observed that $\mathsf{level}(e'\theta) \geqslant \mathsf{level}(e'\sigma_1\theta_1)$. Furthermore, $e'\theta \to_\ell^* e'\sigma_1\theta_1$ and thus $\{u\theta, v\theta\} \succeq_{\mathsf{mul}}^\kappa \{u\sigma_1\theta_1, v\sigma_1\theta_1\}$. Hence $|e'\theta| \sqsupseteq^\kappa |e'\sigma_1\theta_1|$, implying the desired $(H; \theta) \sqsupseteq_{\mathsf{mul}}^\kappa (H\sigma_1; \theta_1)$.

(b) Suppose a reduct of $e\theta$ is rewritten at position 1. In this case we proceed as in case 1(b).

3. Suppose $s \in \mathcal{V}$ and $t \notin \mathcal{V}$. This case is similar to case 2.

4. Suppose $s, t \in \mathcal{V}$. We distinguish two further cases.

   (a) Suppose $s = t$. Let $G_1 = H$. We have $\Psi_1 \colon G \Rightarrow_{[\mathsf{t}]} G_1$. Let $\sigma_1 = \varepsilon$ and $\theta_1 = \theta$. From Π we extract the rewrite sequence $\Pi_1 \colon G_1\theta_1 \to^* \top$. We clearly have $\sigma_1\theta_1 = \theta$ and $\Pi \gg \Pi_1$ as $(G_1;\theta_1) \subset (G;\theta)$.

   (b) Suppose $s \neq t$. Let $\sigma_1 = \{s \mapsto t\}$ and $G_1 = H\sigma_1$. We have $\Psi_1 \colon G \Rightarrow_{[\mathsf{v}],\sigma_1} G_1$. Let θ_1 be the $\mathcal{R}_\ell$-normal form of θ, i.e., $\theta_1(x) = x\theta{\downarrow_\ell}$ for all variables $x \in \mathcal{V}$. Here $x\theta{\downarrow_\ell}$ denotes the unique normal form of $x\theta$ with respect to the confluent and terminating TRS $\mathcal{R}_\ell$. Since $e\theta \to_\ell^* \mathtt{true}$, we have $s\theta{\downarrow_\ell} = t\theta{\downarrow_\ell}$. Hence $s\theta \to_\ell^* t\theta{\downarrow_\ell} = s\sigma_1\theta_1$. Because $x\theta \to_\ell^* x\theta{\downarrow_\ell} = x\sigma_1\theta_1$ for variables x different from s, we obtain $x\theta \to_\ell^* x\sigma_1\theta_1$ for all variables $x \in W$. Therefore $\sigma_1\theta_1 =_{\mathcal{R}} \theta\ [W]$. From Π we extract the rewrite sequence $H\theta \to^* \top$. Since $H\theta \to_\ell^* H\sigma_1\theta_1 = G_1\theta_1$, level-confluence yields a rewrite sequence $\Pi_1 \colon G_1\theta_1 \to^* \top$ such that for every equation $e' \in H$ the level of $e'\sigma_1\theta_1$ is less than or equal to the level of $e'\theta$. Hence we obtain $(H;\theta) \sqsupseteq^{\kappa}_{\mathsf{mul}} (G_1;\theta_1)$ as in case 2(a) and thus $(G;\theta) = \{|e\theta|\} \uplus (H;\theta) \sqsupset^{\kappa}_{\mathsf{mul}} (G_1;\theta_1)$. We conclude that $\Pi \gg \Pi_1$.

Let $W' = \mathcal{V}\mathrm{ar}_W(\sigma_1) \cup \mathcal{V}\mathrm{ar}(G_1)$. Clearly $\mathcal{V}\mathrm{ar}(G_1) \subseteq W'$. Hence we can apply the induction hypothesis to $\Pi_1 \colon G_1\theta_1 \to^* \top$. This yields an LCNC-refutation $\Psi' \colon G_1 \Rightarrow^*_{\sigma'} \Box$ such that $\sigma' \leqslant_{\mathcal{R}} \theta_1\ [W']$. Define $\sigma = \sigma_1\sigma'$. From $\sigma_1\theta_1 \leqslant_{\mathcal{R}} \theta\ [W]$, $\sigma' \leqslant_{\mathcal{R}} \theta_1\ [W']$, and $\mathcal{V}\mathrm{ar}_W(\sigma_1) \subseteq W'$ we infer that $\sigma \leqslant_{\mathcal{R}} \theta\ [W]$. Concatenating the LCNC-step Ψ_1 and the LCNC-refutation Ψ' yields the desired LCNC-refutation Ψ. □

Let us illustrate the above proof on a small example.

Example 5.4 *Consider the CTRS $\mathcal{R}$ consisting of the rewrite rules*

$$\begin{array}{rcll} \mathsf{a} & \to & \mathsf{b} & \Leftarrow \mathsf{f}(x) \approx \mathsf{g}(\mathsf{b}) \\ \mathsf{f}(\mathsf{b}) & \to & \mathsf{g}(x) & \Leftarrow \mathsf{f}(x) \approx \mathsf{g}(\mathsf{b}) \\ \mathsf{f}(\mathsf{a}) & \to & \mathsf{g}(\mathsf{b}) & \end{array}$$

of Werner [33, Beispiel 9.1]. This 3-CTRS is level-confluent and terminating. Consider the goal $G = \mathsf{f}(\mathsf{b}) \approx \mathsf{g}(\mathsf{b})$. The empty substitution ε is a solution of G because we have the rewrite sequence

$$\Pi \colon G\varepsilon = \mathsf{f}(\mathsf{b}) \approx \mathsf{g}(\mathsf{b}) \to_2 \mathsf{g}(\mathsf{a}) \approx \mathsf{g}(\mathsf{b}) \to_2 \mathsf{g}(\mathsf{b}) \approx \mathsf{g}(\mathsf{b}) \to_0 \mathtt{true}.$$

In both the first and the second rewrite step the instantiated condition of the applied rewrite rule is $\mathsf{f(a)} \approx \mathsf{g(b)}$ *which rewrites to* `true` *by applying the third rewrite rule. Below we show a possible* LCNC*-refutation* Ψ *constructed in the above proof. (Note that in general there are several possibilities for* Ψ *as the equation in* $G\theta$ *of minimal level is not uniquely determined.) The selected equations are underlined and the numbers on the right refer to the various cases in the proof:*

$$
\begin{array}{cl}
\underline{\mathsf{f(b)} \approx \mathsf{g(b)}} & \\
\Downarrow_{[\mathrm{o}]},\ \mathsf{f(b)} \to \mathsf{g}(x_1) \Leftarrow \mathsf{f}(x_1) \approx \mathsf{g(b)} & \textit{1(b)} \\
\underline{\mathsf{b} \approx \mathsf{b}},\ \mathsf{g}(x_1) \approx \mathsf{g(b)},\ \mathsf{f}(x_1) \approx \mathsf{g(b)} & \\
\Downarrow_{[\mathrm{d}]} & \textit{1(a)} \\
\mathsf{g}(x_1) \approx \mathsf{g(b)},\ \underline{\mathsf{f}(x_1) \approx \mathsf{g(b)}} & \\
\Downarrow_{[\mathrm{o}]},\ \mathsf{f(a)} \to \mathsf{g(b)} & \textit{1(b)} \\
\mathsf{g}(x_1) \approx \mathsf{g(b)},\ \underline{x_1 \approx \mathsf{a}},\ \mathsf{g(b)} \approx \mathsf{g(b)} & \\
\Downarrow_{[\mathrm{i}]},\ \{x_1 \mapsto \mathsf{a}\} & \textit{3(a)} \\
\mathsf{g(a)} \approx \mathsf{g(b)},\ \underline{\mathsf{g(b)} \approx \mathsf{g(b)}} & \\
\Downarrow_{[\mathrm{d}]} & \textit{1(a)} \\
\mathsf{g(a)} \approx \mathsf{g(b)},\ \underline{\mathsf{b} \approx \mathsf{b}} & \\
\Downarrow_{[\mathrm{d}]} & \textit{1(a)} \\
\underline{\mathsf{g(a)} \approx \mathsf{g(b)}} & \\
\Downarrow_{[\mathrm{d}]} & \textit{1(a)} \\
\underline{\mathsf{a} \approx \mathsf{b}} & \\
\Downarrow_{[\mathrm{o}]},\ \mathsf{a} \to \mathsf{b} \Leftarrow \mathsf{f}(x_2) \approx \mathsf{g(b)} & \textit{1(b)} \\
\underline{\mathsf{b} \approx \mathsf{b}},\ \mathsf{f}(x_2) \approx \mathsf{g(b)} & \\
\Downarrow_{[\mathrm{d}]} & \textit{1(a)} \\
\underline{\mathsf{f}(x_2) \approx \mathsf{g(b)}} & \\
\Downarrow_{[\mathrm{o}]},\ \mathsf{f(a)} \to \mathsf{g(b)} & \textit{1(b)} \\
\underline{x_2 \approx \mathsf{a}},\ \mathsf{g(b)} \approx \mathsf{g(b)} & \\
\Downarrow_{[\mathrm{i}]},\ \{x_2 \mapsto \mathsf{a}\} & \textit{3(a)}
\end{array}
$$

$$\begin{array}{cc}
\underline{\mathsf{g(b)} \approx \mathsf{g(b)}} & \\
\Downarrow_{[\mathsf{d}]} & \textit{1(a)} \\
\underline{\mathsf{b} \approx \mathsf{b}} & \\
\Downarrow_{[\mathsf{d}]} & \textit{1(a)} \\
\square &
\end{array}$$

It is interesting to note that basic conditional narrowing fails to solve the goal $\mathsf{f(b)} \approx \mathsf{g(b)}$ (Werner [33]). Hence the completeness of LCNC for the above CTRS does not follow from the strong completeness result of the previous section. An interesting question for future research is to establish or refute the strong completeness of LCNC for the class of terminating and level-confluent CTRSs and, if strong completeness fails to hold, to identify complete selection functions.

6 Conclusion

In this paper we presented a number of completeness results for the lazy conditional narrowing calculus LCNC. The only result that does not rely on some kind of termination condition is the one of Section 3, for confluent 1-CTRSs with respect to normalized solutions and leftmost selection. However, unlike the results of Sections 4 and 5, this result does not permit extra variables in the conditions and right-hand sides of the rewrite rules. An important question is to find a class of non-terminating CTRSs with extra variables for which (strong) completeness of LCNC (as well as ordinary conditional narrowing) can be established. We believe that the class proposed by Suzuki *et al.* [32] is a promising candidate.

Even when completeness with respect to a specific selection function like $\mathcal{S}_{\mathsf{left}}$ is known, the search space of LCNC is still very large, owing mainly to the delayed matching of patterns in the application of the outermost narrowing rule as well as the non-determinism due to the choice of the inference rule. In future work we would like to investigate under what conditions we can eliminate this non-determinism. In the unconditional case this question has been fully answered ([25]), but it seems doubtful whether the same conditions work for arbitrary confluent (1-)CTRSs. In [13] Ida and Hamada present an implementation of LCNC_{d} in the symbolic computation language Mathematica. LCNC_{d} is the conditional counterpart of the deterministic calculus LNC_{d}

([25]) and incorporates leftmost selection. It is unknown for which classes of CTRSs and solutions this calculus is complete. (No completeness results are reported in [13].)

Acknowledgements

Taro Suzuki is partially supported by the Grant-in-Aid for Encouragement of Young Scientist 11780204. Aart Middeldorp is partially supported by the Grant-in-Aid for Scientific Research (B) 12480066 of the Japanese Ministry of Education, Culture, Sports, Science and Technology.

References

[1] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.

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

[3] A. Bockmayr. *Beiträge zur Theorie des Logisch-Funktionalen Programmierens.* PhD thesis, Universität Karlsruhe, 1990. In German.

[4] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. North-Holland, 1990.

[5] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22:465–476, 1979.

[6] N. Dershowitz, M. Okada, and G. Sivakumar. Canonical conditional rewrite systems. In *Proceedings of the 9th International Conference on Automated Deduction*, volume 310 of *LNCS*, pages 538–549, 1988.

[7] M. Fay. First-order unification in equational theories. In *Proceedings of the 4th Conference on Automated Deduction*, pages 161–167, 1979.

[8] J. Gallier and W. Snyder. Complete sets of transformations for general E-unification. *Theoretical Computer Science*, 67:203–260, 1989.

[9] E. Giovannetti and C. Moiso. A completeness result for E-unification algorithms based on conditional narrowing. In *Proceedings of the Workshop on Foundations of Logic and Functional Programming*, volume 306 of *LNCS*, pages 157–167, 1986.

[10] J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. A higer-order rewriting logic for functional logic programming. In *Proceedings of the International Conference on Logic Programming*, pages 153–167. The MIT Press, 1997.

[11] J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. Polymorphic types in functional logic programming. *Journal of Functional and Logic Programming*, 2001(1), 2001.

[12] M. Hamada. Strong completeness of a narrowing calculus for conditional rewrite systems with extra variables. In *Proceedings of the 6th Australasian Theory Symposium, Electronic Notes in Theoretical Computer Science*, volume 31. Elsevier Science Publishers, 2000.

[13] M. Hamada and T. Ida. Deterministic and non-deterministic lazy conditional narrowing and their implementations. *Transactions of Information Processing Society of Japan*, 39(3):656–663, 1998.

[14] M. Hamada and A. Middeldorp. Strong completeness of a lazy conditional narrowing calculus. In *Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming*, pages 14–32, Shonan Village, 1997. World Scientific.

[15] M. Hamada, A. Middeldorp, and T. Suzuki. Completeness results for a lazy conditional narrowing calculus. In *Combinatorics, Computation and Logic: Proceedings of 2nd Discrete Mathematics and Theoretical Computer Science Conference and the 5th Australasian Theory Symposium*, pages 217–231, Auckland, 1999. Springer-Verlag Singapore.

[16] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19 & 20:583–628, 1994.

[17] M. Hanus. Lazy unification with simplification. In *Proceedings of the 5th European Symposium on Programming*, volume 788 of *LNCS*, pages 272–286, 1994.

[18] S. Hölldobler. *Foundations of Equational Logic Programming*, volume 353 of *LNAI*. 1989.

[19] J.-M. Hullot. Canonical forms and unification. In *Proceedings of the 5th Conference on Automated Deduction*, volume 87 of *LNCS*, pages 318–334, 1980.

[20] S. Kaplan. Conditional rewrite rules. *Theoretical Computer Science*, 33:175–193, 1984.

[21] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.

[22] M. Marin, T. Ida, and T. Suzuki. On reducing the search space of higer-order lazy narrowing. In *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming*, volume 1722 of *LNCS*, pages 319–334, 1999.

[23] A. Martelli, C. Moiso, and G.F. Rossi. Lazy unification algorithms for canonical rewrite systems. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures, Vol. II, Rewriting Techniques*, pages 245–274. Academic Press Press, 1989.

[24] A. Middeldorp and E. Hamoen. Completeness results for basic narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5:213–253, 1994.

[25] A. Middeldorp and S. Okui. A deterministic lazy narrowing calculus. *Journal of Symbolic Computation*, 25(6):733–757, 1998.

[26] A. Middeldorp, S. Okui, and T. Ida. Lazy narrowing: Strong completeness and eager variable elimination. *Theoretical Computer Science*, 167(1,2):95–130, 1996.

[27] C. Prehofer. A call-by-need strategy for higher-order functional-logic programming. In *Proceedings of the International Symposium on Logic Programming*, pages 147–161. MIT Press, 1995.

[28] C. Prehofer. *Solving Higher-Order Equations: From Logic to Programming*. PhD thesis, Technische Universität München, 1995. Appeared as Technical Report 19508.

[29] C. Prehofer. *Solving Higher-Order Equations: From Logic to Programming.* Progress in Theoretical Computer Science. Birkäuser, 1998.

[30] W. Snyder. *A Proof Theory for General Unification*, volume 11 of *Progress in Computer Science and Applied Logic.* Birkäuser, 1991.

[31] T. Suzuki and A. Middeldorp. A complete selection function for lazy conditional narrowing. In *Proceedings of the 5th International Symposium on Functional and Logic Programming*, volume 2024 of *Lecture Notes in Computer Science*, pages 201–215, Tokyo, 2001. Springer-Verlag.

[32] T. Suzuki, A. Middeldorp, and T. Ida. Level-confluence of conditional rewrite systems with extra variables in right-hand sides. In *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Computer Science*, pages 179–193, Kaiserslautern, 1995. Springer-Verlag.

[33] A. Werner. *Untersuchung von Strategien für das logisch-funktionale Programmieren.* Shaker-Verlag Aachen, March 1998. In German.

A Appendix

In this appendix we prove Lemmata 4.4 and 4.13. Most of the work for the former is done in the following preliminary lemma.

Lemma A.1 *Let* $\Pi\colon G \rightsquigarrow_{\theta}^{*} \top$ *be a* CNC*-refutation,* W *a set of variables, and* γ *a substitution such that* $\mathcal{V}\mathrm{ar}(G) \subseteq W$, $\gamma \leqslant \theta\ [W]$, *and the variables in* $\mathcal{D}(\gamma) \cup \mathcal{I}(\gamma)$ *are different from the variables in the employed rewrite rules. There exists a* CNC*-refutation* $\Pi'\colon G\gamma \rightsquigarrow_{\theta'}^{*} \top$ *such that* Π *subsumes* Π', $\Pi\theta = \Pi'\theta'$, *and* $\gamma\theta' = \theta\ [W]$.

Proof We use induction on the length n of Π. The case $n = 0$ is obvious. Suppose $n > 0$. Let the first step of Π be

$$G = (G', e, G'') \rightsquigarrow_{\sigma_1, p_1, l_1 \to r_1 \Leftarrow c_1, e} (G', e[r_1]_{p_1}, c_1, G'')\sigma_1 = G_1$$

and let $\Pi_1\colon G_1 \rightsquigarrow_{\theta_1}^{*} \top$ be the remainder of Π, so $\sigma_1\theta_1 = \theta$. Let X be the set of variables in the rewrite rules employed in Π. Without loss of generality we assume that $\mathcal{V}\mathrm{ar}(G) \cap X = \varnothing$ and $X' \cap (\mathcal{D}(\sigma_1) \cup \mathcal{I}(\sigma_1)) = \varnothing$. Here X'

is defined as the set $X \setminus \mathcal{V}\mathrm{ar}(l_1 \to r_1 \Leftarrow c_1)$. These two assumptions simply state that the variables in the rewrite rules are sufficiently fresh.

First we show that we can mimic the first step of Π starting from the goal $G\gamma$. From $\gamma \leqslant \theta$ $[W]$ we obtain a substitution δ such that $\gamma\delta = \theta$ $[W]$. Without loss of generality we may assume that $\mathcal{D}(\delta) \subseteq \mathcal{V}\mathrm{ar}_W(\gamma)$. Hence the substitution δ' defined by

$$\delta'(x) = \begin{cases} x\delta & \text{if } x \in \mathcal{V}\mathrm{ar}_W(\gamma), \\ x\theta & \text{otherwise} \end{cases}$$

satisfies $\gamma\delta' = \theta$ $[W \cup X]$ because $(\mathcal{D}(\gamma) \cup \mathcal{I}(\gamma)) \cap X = \varnothing$. Since $\mathcal{V}\mathrm{ar}(e) \subseteq \mathcal{V}\mathrm{ar}(G) \subseteq W$ and $\mathcal{D}(\gamma) \cap X = \varnothing$ we have $e\gamma_{|p_1}\delta' = e_{|p_1}\gamma\delta' = e_{|p_1}\theta = e_{|p_1}\sigma_1\theta_1 = l_1\sigma_1\theta_1 = l_1\theta = l_1\gamma\delta' = l_1\delta'$, so $e\gamma_{|p_1}$ and l_1 are unifiable. Hence we obtain the CNC-step

$$G\gamma \rightsquigarrow_{p_1, \sigma_1', l_1 \to r_1 \Leftarrow c_1, e\gamma} (G'\gamma, e\gamma[r_1]_{p_1}, c_1, G''\gamma)\sigma_1'$$

where σ_1' is any idempotent most general unifier of $e\gamma_{|p_1}$ and l_1.

Next we show that $(G'\gamma, e\gamma[r_1]_{p_1}, c_1, G''\gamma)\sigma_1' = G_1\gamma_1$ for a substitution γ_1 satisfying $\gamma_1 \leqslant \theta_1$ $[W']$ for a set of variables W' that includes $\mathcal{V}\mathrm{ar}(G_1)$. We have $e_{|p_1}\gamma\sigma_1' = e\gamma_{|p_1}\sigma_1' = l_1\sigma_1' = l_1\gamma\sigma_1'$, so $\gamma\sigma_1'$ is a unifier of $e_{|p_1}$ and l_1. Since σ_1 is a most general unifier of $e_{|p_1}$ and l_1 there exists a substitution γ' such that $\sigma_1\gamma' = \gamma\sigma_1'$. Let $W' = \mathcal{V}\mathrm{ar}_{W \cup X}(\sigma_1)$ and $\gamma_1 = \gamma'\restriction_{W'}$. It is easy to show that $\mathcal{V}\mathrm{ar}(G_1) \subseteq W'$. We have $\sigma_1\gamma_1 = \gamma\sigma_1'$ $[W \cup X]$ and because $\mathcal{D}(\gamma) \cap X = \varnothing$ we obtain $(G'\gamma, e\gamma[r_1]_{p_1}, c_1, G''\gamma)\sigma_1' = (G'\gamma, e\gamma[r_1\gamma]_{p_1}, c_1\gamma, G''\gamma)\sigma_1' = G_1\gamma_1$. Because δ' is a unifier of $e\gamma_{|p_1}$ and l_1, there exists a substitution δ'' such that $\sigma_1'\delta'' = \delta'$. Using $\sigma_1\gamma_1 = \gamma\sigma_1'$ $[W \cup X]$ we obtain $\sigma_1\gamma_1\delta'' = \gamma\sigma_1'\delta'' = \gamma\delta' = \sigma_1\theta_1$ $[W \cup X]$ and thus $\gamma_1 \leqslant \theta_1$ $[W']$.

We still have to show that $(\mathcal{D}(\gamma_1) \cup \mathcal{I}(\gamma_1)) \cap X' = \varnothing$ before we can apply the induction hypothesis to Π_1. Because σ_1' is an idempotent most general unifier of $e\gamma_{|p_1}$ and l_1, we have $\mathcal{D}(\sigma_1') \cup \mathcal{I}(\sigma_1') = \mathcal{V}\mathrm{ar}(e\gamma_{|p_1}) \cup \mathcal{V}\mathrm{ar}(l_1)$. Clearly $\mathcal{V}\mathrm{ar}(e\gamma_{|p_1}) \subseteq \mathcal{V}\mathrm{ar}(G) \cup \mathcal{I}(\gamma)$ and thus $\mathcal{V}\mathrm{ar}(e\gamma_{|p_1}) \cap X = \varnothing$ by assumption. Also $\mathcal{V}\mathrm{ar}(l_1) \subseteq X \setminus X'$. Hence $(\mathcal{D}(\sigma_1') \cup \mathcal{I}(\sigma_1')) \cap X' = \varnothing$. Together with $\mathcal{D}(\sigma_1) \cap X' = \varnothing$, $\mathcal{D}(\gamma) \cap X = \varnothing$, and $\sigma_1\gamma_1 = \gamma\sigma_1'$ $[W \cup X]$ this implies $\gamma_1 = \varepsilon$ $[X']$, i.e., $\mathcal{D}(\gamma_1) \cap X' = \varnothing$. It remains to show that $\mathcal{I}(\gamma_1) \cap X' = \varnothing$. First observe that from $\mathcal{I}(\gamma\sigma_1') \subseteq \mathcal{I}(\gamma) \cup \mathcal{I}(\sigma_1')$, $\mathcal{I}(\gamma) \cap X = \varnothing$, and $\mathcal{I}(\sigma_1') \cap X' = \varnothing$ it follows that $\mathcal{I}(\gamma\sigma_1') \cap X' = \varnothing$. Suppose to the contrary that $x \in \mathcal{I}(\gamma_1) \cap X'$ for some variable x. So there exists a variable $y \in \mathcal{D}(\gamma_1)$ such that $x \in \mathcal{V}\mathrm{ar}(y\gamma_1)$. We have $\mathcal{D}(\gamma_1) \subseteq ((W \cup X) \setminus \mathcal{D}(\sigma_1)) \cup \mathcal{I}(\sigma_1\restriction_{W \cup X})$. If $y \in (W \cup X) \setminus \mathcal{D}(\sigma_1)$ then $y\gamma_1 = y\sigma_1\gamma_1 = y\gamma\sigma_1'$, so $x \in \mathcal{I}(\gamma\sigma_1')$. This contradicts $\mathcal{I}(\gamma\sigma_1') \cap X' = \varnothing$. If $y \in \mathcal{I}(\sigma_1\restriction_{W \cup X})$ then there exists a variable $z \in W \cup X$ with $y \in \mathcal{V}\mathrm{ar}(z\sigma_1)$ and thus $x \in \mathcal{V}\mathrm{ar}(z\sigma_1\gamma_1) = \mathcal{V}\mathrm{ar}(z\gamma\sigma_1')$, again contradicting $\mathcal{I}(\gamma\sigma_1') \cap X' = \varnothing$.

Now we are in a position to apply the induction hypothesis to Π_1. This yields a CNC-refutation $G_1\gamma_1 \rightsquigarrow^*_{\theta_1'} \top$ such that $\gamma_1\theta_1' = \theta_1\ [W']$. Concatenating this CNC-refutation with the CNC-step $G\gamma \rightsquigarrow_{\sigma_1'} G_1\gamma_1$ yields the CNC-refutation Π': $G\gamma \rightsquigarrow^*_{\theta'} \top$. Here $\theta' = \sigma_1'\theta_1'$. From $\gamma_1\theta_1' = \theta_1\ [W']$ we infer $\sigma_1\gamma_1\theta_1' = \sigma_1\theta_1 = \theta\ [W \cup X]$ and thus $\gamma\theta' = \gamma\sigma_1'\theta_1' = \sigma_1\gamma_1\theta_1' = \theta\ [W \cup X]$. Hence also $\gamma\theta' = \theta\ [W]$. From the construction of Π' it follows that Π subsumes Π' and $\Pi\theta = \Pi'\theta'$. □

Proof of Lemma 4.4. Let $t\theta = f(t_1, \dots, t_n)$ and define the substitution σ' as follows:

$$\sigma'(x) = \begin{cases} t_i & \text{if } x = x_i \text{ for some } 1 \leqslant i \leqslant n, \\ x\theta & \text{otherwise.} \end{cases}$$

We clearly have $\sigma_1\sigma' = \theta\ [W]$ and thus $\sigma_1 \leqslant \theta\ [W]$. An application of Lemma A.1 yields the desired result. □

Proof of Lemma 4.13. Let $G_1 = G', e_1, G'', e_2, G'''$. Since we may assume that the variables in $l_2 \to r_2 \Leftarrow c_2$ are fresh, we have $\mathcal{D}(\tau_1) \cap \mathcal{V}\mathrm{ar}(l_2 \to r_2 \Leftarrow c_2) = \varnothing$. Hence $r_2\tau_1 = r_2$, $c_2\tau_1 = c_2$, and thus

$$\begin{aligned} G_3 &= ((G', e_1[r_1]_{p_1}, c_1, G'')\tau_1, e_2\tau_1[r_2]_{p_2}, c_2, G'''\tau_1)\tau_2 \\ &= (G', e_1[r_1]_{p_1}, c_1, G'', e_2[r_2]_{p_2}, c_2, G''')\tau_1\tau_2. \end{aligned}$$

From $\mathcal{D}(\tau_1) \cap \mathcal{V}\mathrm{ar}(l_2) = \varnothing$ we also infer that $e_{2|p_2}\tau_1\tau_2 = e_2\tau_{1|p_2}\tau_2 = l_2\tau_2 = l_2\tau_1\tau_2$, so $e_{2|p_2}$ and l_2 are unifiable. Let υ_2 be an idempotent most general unifier of these two terms. By definition of most general unifier there exists a substitution ρ such that $\upsilon_2\rho = \tau_1\tau_2$. We have $H_2 = (G', e_1, G'', e_2[r_2]_{p_2}, c_2, G''')\upsilon_2$ and $\mathcal{D}(\upsilon_2) \subseteq \mathcal{V}\mathrm{ar}(e_{2|p_2}) \cup \mathcal{V}\mathrm{ar}(l_2)$ by idempotency of υ_2. Because we may assume that $\mathcal{V}\mathrm{ar}(l_1 \to r_1 \Leftarrow c_1) \cap (\mathcal{V}\mathrm{ar}(e_2) \cup \mathcal{V}\mathrm{ar}(l_2 \to r_2 \Leftarrow c_2)) = \varnothing$, we obtain $\mathcal{D}(\upsilon_2) \cap \mathcal{V}\mathrm{ar}(l_1 \to r_1 \Leftarrow c_1) = \varnothing$. Hence $e_1\upsilon_{2|p_1}\rho = e_{1|p_1}\upsilon_2\rho = e_{1|p_1}\tau_1\tau_2 = l_1\tau_1\tau_2 = l_1\upsilon_2\rho = l_1\rho$. So the terms $e_1\upsilon_{2|p_1}$ and l_1 are unifiable. Let σ be a most general unifier. We have $\sigma \leqslant \rho$. It follows that $\upsilon_2\sigma \leqslant \tau_1\tau_2$. Using $\mathcal{D}(\upsilon_2) \cap \mathcal{V}\mathrm{ar}(l_1) = \varnothing$ we obtain $e_{1|p_1}\upsilon_2\sigma = e_1\upsilon_{2|p_1}\sigma = l_1\sigma = l_1\upsilon_2\sigma$, so $\upsilon_2\sigma$ is a unifier of $e_{1|p_1}$ and l_1. Because τ_1 is a most general unifier of these two terms, we must have $\tau_1 \leqslant \upsilon_2\sigma$. Let γ be any substitution satisfying $\tau_1\gamma = \upsilon_2\sigma$. With help of $\mathcal{D}(\tau_1) \cap \mathcal{V}\mathrm{ar}(l_2) = \varnothing$ we obtain $e_2\tau_{1|p_2}\gamma = e_{2|p_2}\tau_1\gamma = e_{2|p_2}\upsilon_2\sigma = l_2\upsilon_2\sigma = l_2\tau_1\gamma = l_2\gamma$. Hence we obtain $\tau_2 \leqslant \gamma$ from the fact that τ_2 is a most general unifier of $e_2\tau_{1|p_2}$ and l_2. Therefore $\tau_1\tau_2 \leqslant \tau_1\gamma = \upsilon_2\sigma$. Since we also have $\upsilon_2\sigma \leqslant \tau_1\tau_2$, there is a variable renaming δ such that $\upsilon_2\sigma\delta = \tau_1\tau_2$. Now define $\upsilon_1 = \sigma\delta$. Since most general unifiers are closed under variable renaming, υ_1 is a most general unifier of $e_1\upsilon_{2|p_1}$ and l_1. From $\mathcal{D}(\upsilon_2) \cap \mathcal{V}\mathrm{ar}(l_1 \to r_1 \Leftarrow c_1) = \varnothing$ we infer $r_1\upsilon_2 = r_1$, $c_1\upsilon_2 = c_1$, and thus

$$\begin{aligned} H_3 &= (G'\upsilon_2, e_1\upsilon_2[r_1]_{p_1}, c_1, (G'', e_2[r_2]_{p_2}, c_2, G''')\upsilon_2)\upsilon_1 \\ &= (G', e_1[r_1]_{p_1}, c_1, G'', e_2[r_2]_{p_2}, c_2, G''')\upsilon_2\upsilon_1. \end{aligned}$$

Since $\tau_1\tau_2 = \upsilon_2\upsilon_1$ we conclude that $G_3 = H_3$. □

Confluence and Termination of Simply Typed Term Rewriting Systems

Toshiyuki Yamada

Institute of Information Sciences and Electronics
University of Tsukuba, Tsukuba 305–8573, Japan

toshi@score.is.tsukuba.ac.jp

Abstract. We propose simply typed term rewriting systems (STTRSs), which extend first-order rewriting by allowing higher-order functions. We study a simple proof method for confluence which employs a characterization of the diamond property of a parallel reduction. By an application of the proof method, we obtain a new confluence result for orthogonal conditional STTRSs. We also discuss a semantic method for proving termination of STTRSs based on monotone interpretation.

1 Introduction

Higher-order function is one of the useful features in functional programming. The well-known higher-order function *map* takes a function as an argument and applies it to all elements of a list:

$$\begin{array}{lcl} map\ f\ [\,] & = & [\,] \\ map\ f\ (x : xs) & = & f\ x\ :\ map\ f\ xs \end{array}$$

It is not possible to directly express this definition by using a first-order term rewriting system, because the variable f is also used as a function. In order to deal with higher-order functions, one can use higher-order rewriting (e.g. Combinatory Reduction Systems [Klo80], Higher-Order Rewrite Systems [NP98]). Higher-order rewriting is a computation model which deals with higher-order terms. Higher-order functions and bound variables are usually used for constructing the set of higher-order terms. The use of bound variables enriches the descriptive power of higher-order rewrite systems. However, it makes the theory complicated. The aim of this paper is to give a simple definition of rewrite systems which conservatively extend the ordinary (first-order) term rewriting systems in such a way that they can naturally express equational specifications containing higher-order functions. We propose higher-order rewrite systems which are close to the format of functional programming languages with pattern matching. Based on the new definition of higher-order rewrite systems (given in the next section) we investigate the confluence property of the rewrite systems. We first study a method for proving confluence using parallel reduction (in Section 3). Based on the proof method introduced, we prove confluence of orthogonal higher-order rewrite systems (in Section 4). This confluence result is further extended to the case of conditional higher-order rewriting (in Section 5). We also discuss a semantic method for proving termination of the newly proposed rewrite systems (in Section 6).

2 Simply Typed Term Rewriting Systems

We assume the reader is familiar with abstract rewrite systems (ARSs) and (first-order) term rewrite systems (TRSs). In this section we propose a simple extension of first-order rewrite systems (cf. [DJ90], [Klo92], [BN98]) to the higher-order case. The basic idea behind the following construction of terms is the same as typed combinatory logic (see e.g. [HS86]): variables may take arguments, and types are introduced in order to apply a function to its arguments correctly.

Definition 1 (simply typed term, position, substitution)
The set of *simple types* is the smallest set ST which contains the *base type* o and satisfies the property that $\tau_1 \times \cdots \times \tau_n \to \tau_0 \in \mathsf{ST}$ whenever $\tau_0, \tau_1, \ldots, \tau_n \in \mathsf{ST}$ with $n \geq 1$. We call a non-base type a *function type*. Let V^τ be a set of *variable symbols* of type τ and C^τ be a set of *constant symbols* of type τ, for every type τ. The set $\mathsf{T}(V, C)^\tau$ of *(simply typed) terms* of type τ is the smallest set satisfying the following two properties: (1) if $t \in V^\tau \cup C^\tau$ then $t \in \mathsf{T}(V, C)^\tau$, and (2) if $n \geq 1$, $t_0 \in \mathsf{T}(V, C)^{\tau_1 \times \cdots \times \tau_n \to \tau}$, and $t_i \in \mathsf{T}(V, C)^{\tau_i}$ for $i = 1, \ldots, n$, then $(t_0\, t_1 \cdots t_n) \in \mathsf{T}(V, C)^\tau$. Note that τ is not necessarily the base type. We also define $V = \bigcup_{\tau \in \mathsf{ST}} V^\tau, C = \bigcup_{\tau \in \mathsf{ST}} C^\tau$, and $\mathsf{T}(V, C) = \bigcup_{\tau \in \mathsf{ST}} \mathsf{T}(V, C)^\tau$. In order to be consistent with the standard definition of first-order terms, we use $t_0(t_1, \ldots, t_n)$ as an alternative notation for $(t_0\, t_1 \cdots t_n)$. The outermost parentheses of a term can be omitted. To enhance readability, infix notation is allowed. We use the notation t^τ to make the type τ of a term t explicit.

We do not confuse non-variable symbols with function symbols as in the first-order case, because a variable symbol of non-base type expresses a function. The term t_0 in a term of the form $(t_0\, t_1 \cdots t_n)$ expresses a function which is applied to the arguments $t_1, \cdots, t_n$. The construction of the simply typed terms allows arbitrary terms, including variables, at the position of t_0, while only constant symbols are allowed in the first-order case. Thus a set of first-order terms is obtained as a subset of simply typed terms which satisfies both $V^\tau = \varnothing$ whenever $\tau \neq o$, and $C^{\tau_1 \times \cdots \times \tau_n \to \tau} = \varnothing$ whenever $\tau_i \neq o$ for some i or $\tau \neq o$.

Let t be a term in $\mathsf{T}(V, C)$. The head symbol $\mathsf{head}(t)$ of t is defined as follows: (1) $\mathsf{head}(t) = t$ if $t \in V \cup C$, and (2) $\mathsf{head}(t) = t_0$ if $t = (t_0\, t_1 \cdots t_n)$. The set of variable symbols occurring in t is denoted by $\mathsf{Var}(t)$.

A *position* is a sequence of natural numbers. The empty sequence ϵ is called the *root position*. Positions are partially ordered by $\leq$ as follows: $p \leq q$ if there exists a position r such that $pr = q$. The set of positions in a term t is denoted by $\mathsf{Pos}(t)$. The *subterm* $t_{|p}$ of t at position p is defined as follows: (1) $t_{|p} = t$ if $p = \epsilon$, and (2) $t_{|p} = t_{i|q}$ if $t = (t_0\, t_1 \cdots t_n)$ and $p = iq$. The term obtained from a term s by replacing its subterm at position p with a term t is denoted by $s[t]_p$. The set $\mathsf{Pos}(t)$ is divided into three parts. We say $p \in \mathsf{Pos}(t)$ is at a *variable position* of t if $t_{|p} \in V$, at a *constant position* if $t_{|p} \in C$, otherwise p is at an *application position*. We denote the set of variable positions, constant positions, and application positions in a term t by $\mathsf{Pos_v}(t)$, $\mathsf{Pos_c}(t)$, and $\mathsf{Pos_a}(t)$, respectively.

A *substitution* σ is a function from V to $\mathsf{T}(V, C)$ such that its *domain*, defined as the set $\{ x \in V \mid \sigma(x) \neq x \}$, is finite and $\sigma(x) \in \mathsf{T}(V, C)^\tau$ whenever $x \in V^\tau$. A substitution $\sigma : V \to \mathsf{T}(V, C)$ is extended to the function $\overline{\sigma} : \mathsf{T}(V, C) \to \mathsf{T}(V, C)$ as follows: (1) $\overline{\sigma}(t) = \sigma(t)$ if $t \in V$, (2) $\overline{\sigma}(t) = t$ if $t \in C$, and (3) $\overline{\sigma}(t) = (\overline{\sigma}(t_0)\,\overline{\sigma}(t_1) \cdots \overline{\sigma}(t_n))$ if $t = (t_0\, t_1 \cdots t_n)$. We will write $t\sigma$ instead of $\overline{\sigma}(t)$. A *renaming* is a bijective substitution from V to V. Two terms t and s are *unifiable* by a *unifier* σ if $s\sigma = t\sigma$.

Definition 2 (rewrite rule and rewrite relation)
Let $\mathsf{T}(V, C)$ be a set of simply typed terms. A *rewrite rule* is a pair of terms, written as $l \to r$, such that $\mathsf{Var}(r) \subseteq \mathsf{Var}(l)$, $\mathsf{head}(l) \in C$, and l and r are of the same type. The terms l and r are called the *left-hand side* and the *right-hand side* of the rewrite rule, respectively. Let R be a set of rewrite rules. We call $\mathcal{R} = (R, V, C)$ a *simply typed term rewriting system* (STTRS for short).

Let $\mathcal{R} = (R, V, C)$ be an STTRS. We say a term s *rewrites to* t, and write $s \to_{\mathcal{R}} t$, if there exists a rewrite rule $l \to r \in R$, a position $p \in \mathsf{Pos}(s)$ and a substitution σ such that $s_{|p} = l\sigma$ and $t = s[r\sigma]_p$. We call the subterm $s_{|p}$ a *redex* of s. In order to make the position p of a rewrite step explicit, we also use the notation $s \xrightarrow{p}_{\mathcal{R}} t$. Especially, a rewrite step at root position is denoted by $\xrightarrow{\epsilon}_{\mathcal{R}}$ and a rewrite step at non-root position is denoted by $\xrightarrow{>\epsilon}_{\mathcal{R}}$. When the underlying STTRS $\mathcal{R}$ is clear from the context, we may omit the subscript $\mathcal{R}$ in $\to_{\mathcal{R}}$.

The definition of STTRS is close to the algebraic functional systems defined by Jouannaud and Okada [JO91]. The main difference is that we dispenses with bound variables since our focus is on higher-order functions and not on quantification. It is easy to see that every variable symbol is in normal form because of the second restriction imposed on the rewrite rules. The following example shows that a simple functional programming language with pattern matching can be modeled by STTRS.

Example 3 (functional programming by STTRS)
Let C be the set which consists of constant symbols $\mathsf{0}^o$, $\mathsf{S}^{o \to o}$, $[]^o$, $:^{o \times o \to o}$, $\mathsf{map}^{(o \to o) \times o \to o}$, $\circ^{(o \to o) \times (o \to o) \to (o \to o)}$, and $\mathsf{twice}^{(o \to o) \to (o \to o)}$, where $:$ and $\circ$ are infix symbols, and the set of variable symbols V contains x^o, xs^o, $F^{o \to o}$, and $G^{o \to o}$. We define the set of rewrite rules R as follows:

$$R = \left\{ \begin{array}{ll} \mathsf{map}\ F\ [] & \to [] \\ \mathsf{map}\ F\ (x : xs) & \to F\,x\ :\ \mathsf{map}\ F\ xs \\ (F \circ G)\ x & \to F\ (G\,x) \\ \mathsf{twice}\ F & \to F \circ F \end{array} \right\}.$$

Examples of rewrite sequences of the TRS $\mathcal{R} = (R, V, C)$ are:

$$\begin{aligned}
\mathsf{map}\ \underline{(\mathsf{twice\ S})}\ (0 : [\,]) &\rightarrow_{\mathcal{R}} \underline{\mathsf{map}\ (\mathsf{S} \circ \mathsf{S})\ (0 : [\,])} \\
&\rightarrow_{\mathcal{R}} \underline{(\mathsf{S} \circ \mathsf{S})\ 0}\ :\ \mathsf{map}\ (\mathsf{S} \circ \mathsf{S})\ [\,] \\
&\rightarrow_{\mathcal{R}} \mathsf{S(S0)}\ :\ \underline{\mathsf{map}\ (\mathsf{S} \circ \mathsf{S})\ [\,]} \\
&\rightarrow_{\mathcal{R}} \mathsf{S(S0)}\ :\ [\,] \\
\underline{\mathsf{map}\ (\mathsf{twice\ S})\ (0 : [\,])} &\rightarrow_{\mathcal{R}} \underline{(\mathsf{twice\ S})}\ 0\ :\ \mathsf{map}\ (\mathsf{twice\ S})\ [\,] \\
&\rightarrow_{\mathcal{R}} \underline{(\mathsf{S} \circ \mathsf{S})\ 0}\ :\ \mathsf{map}\ (\mathsf{twice\ S})\ [\,] \\
&\rightarrow_{\mathcal{R}} \mathsf{S(S0)}\ :\ \underline{\mathsf{map}\ (\mathsf{twice\ S})\ [\,]} \\
&\rightarrow_{\mathcal{R}} \mathsf{S(S0)}\ :\ [\,]
\end{aligned}$$

where underlined redexes are rewritten.

3 Confluence by Parallel Moves

In this section, we develop a method for proving confluence using parallel reduction. We assume the reader is familiar with the basic notions of abstract rewrite systems (ARSs). For more detailed descriptions on abstract rewriting, see, for example, Klop's survey [Klo92].

Definition 4 (properties of an ARS)
Let $\mathcal{A} = (A, \rightarrow)$ be an ARS. We use the following abbreviations:

property	definition	abbreviation
$\mathcal{A}$ has the *diamond property*	$\leftarrow \cdot \rightarrow\ \subseteq\ \rightarrow \cdot \leftarrow$	$\diamond(\rightarrow)$
$\mathcal{A}$ is *confluent*	${}^{*}\!\!\leftarrow \cdot \rightarrow^{*}\ \subseteq\ \rightarrow^{*} \cdot {}^{*}\!\!\leftarrow$	$\mathrm{CR}(\rightarrow)$

Lemma 5 (confluence by simultaneous reduction)
Let $(A, \rightarrow)$ and $(A, \hookrightarrow)$ be ARSs.
(1) $\diamond(\rightarrow) \Longrightarrow \mathrm{CR}(\rightarrow)$.
(2) If $\hookrightarrow^{*} = \rightarrow^{*}$ then $\mathrm{CR}(\hookrightarrow) \Longleftrightarrow \mathrm{CR}(\rightarrow)$.

Proof. Straightforward. □

Definition 6 (parallel reduction)
Let $\mathcal{R} = (R, V, C)$ be an STTRS. The *parallel reduction relation* induced by $\mathcal{R}$ is the smallest relation $\mathrel{+\!\!\!\!\rightarrow}_{\mathcal{R}}$ such that
(1) if $t \in V \cup C$ then $t \mathrel{+\!\!\!\!\rightarrow}_{\mathcal{R}} t$,
(2) if $s \xrightarrow{\epsilon}_{\mathcal{R}} t$ then $s \mathrel{+\!\!\!\!\rightarrow}_{\mathcal{R}} t$, and
(3) if $n \geq 1$ and $s_i \mathrel{+\!\!\!\!\rightarrow}_{\mathcal{R}} t_i$ for $i = 0, \ldots, n$, then $(s_0\, s_1 \cdots s_n) \mathrel{+\!\!\!\!\rightarrow}_{\mathcal{R}} (t_0\, t_1 \cdots t_n)$.
We may omit the underlying TRS $\mathcal{R}$ in $\mathrel{+\!\!\!\!\rightarrow}_{\mathcal{R}}$ if it is not important.

One can easily verify that every parallel reduction relation is reflexive. Note also that $\rightarrow\ \subseteq\ \mathrel{+\!\!\!\!\rightarrow}\ \subseteq\ \rightarrow^{*}$, hence $\mathrel{+\!\!\!\!\rightarrow}^{*} = \rightarrow^{*}$. From Lemma 5 we know that the diamond property of a parallel reduction relation is a sufficient condition for the confluence of the underlying TRS: $\diamond(\mathrel{+\!\!\!\!\rightarrow}) \Longrightarrow \mathrm{CR}(\mathrel{+\!\!\!\!\rightarrow}) \Longleftrightarrow \mathrm{CR}(\rightarrow)$. The following lemma gives a characterization of the diamond property of a parallel reduction, which is inspired by Gramlich's characterization of the strong confluence of a parallel reduction [Gra96].

Lemma 7 (parallel moves)
We have $\mathrel{\leftarrow\!\!\!\|} \cdot \xrightarrow{\epsilon} \subseteq \mathrel{\|\!\!\!\rightarrow} \cdot \mathrel{\leftarrow\!\!\!\|} \iff \mathrel{\leftarrow\!\!\!\|} \cdot \mathrel{\|\!\!\!\rightarrow} \subseteq \mathrel{\|\!\!\!\rightarrow} \cdot \mathrel{\leftarrow\!\!\!\|}$.

Proof. The implication from right to left is obvious by $\xrightarrow{\epsilon} \subseteq \mathrel{\|\!\!\!\rightarrow}$. For the reverse implication, suppose $\mathrel{\leftarrow\!\!\!\|} \cdot \xrightarrow{\epsilon} \subseteq \mathrel{\|\!\!\!\rightarrow} \cdot \mathrel{\leftarrow\!\!\!\|}$. We show that if $t \mathrel{\leftarrow\!\!\!\|} s \mathrel{\|\!\!\!\rightarrow} u$ then there exists a term v such that $t \mathrel{\|\!\!\!\rightarrow} v \mathrel{\leftarrow\!\!\!\|} u$, for all terms s, t, and u. The proof is by induction on the structure of s. We distinguish three cases according to the parallel reduction $s \mathrel{\|\!\!\!\rightarrow} t$. If $s = t \in V \cup C$ then $t \mathrel{\|\!\!\!\rightarrow} v = u$ by taking $v = u$. If $s \xrightarrow{\epsilon} t$ or $s \xrightarrow{\epsilon} u$ then we use the assumption. Otherwise, we have $s = (s_0\, s_1 \cdots s_n)$, $t = (t_0\, t_1 \cdots t_n)$, and $u = (u_0\, u_1 \ldots u_n)$ for some $n \geq 1$ with $t_i \mathrel{\leftarrow\!\!\!\|} s_i \mathrel{\|\!\!\!\rightarrow} u_i\, (i = 0, \ldots, n)$. By the induction hypothesis, we know the existence of terms v_i $(i = 0, \ldots, n)$ such that $t_i \mathrel{\|\!\!\!\rightarrow} v_i \mathrel{\leftarrow\!\!\!\|} u_i$. Let $v = (v_0\, v_1 \ldots v_n)$. Then we have $t \mathrel{\|\!\!\!\rightarrow} v \mathrel{\leftarrow\!\!\!\|} u$ by definition. □

This lemma allows us to partially localize the test for the diamond property of a parallel reduction, though the complete localization is impossible as shown in the following example.

Example 8 (complete localization of parallel moves)
In this example we show that the implication $\leftarrow \cdot \xrightarrow{\epsilon} \subseteq \mathrel{\|\!\!\!\rightarrow} \cdot \mathrel{\leftarrow\!\!\!\|} \implies \mathrel{\leftarrow\!\!\!\|} \cdot \xrightarrow{\epsilon} \subseteq \mathrel{\|\!\!\!\rightarrow} \cdot \mathrel{\leftarrow\!\!\!\|}$ does not hold in general. Let $V = \varnothing$ and C be the set consisting of the constant symbol f of type $o \times o \to o$ and constant symbols $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}$ of type o. Consider the set R of rewrite rules defined by

$$R = \left\{ \begin{array}{ll} \mathsf{f\ a\ a} \to \mathsf{c} & \mathsf{a} \to \mathsf{b} \\ \mathsf{f\ a\ b} \to \mathsf{d} & \mathsf{c} \to \mathsf{d} \\ \mathsf{f\ b\ a} \to \mathsf{d} & \mathsf{d} \to \mathsf{e} \\ \mathsf{f\ b\ b} \to \mathsf{e} & \end{array} \right\}.$$

It is easy to see that the inclusion $\leftarrow \cdot \xrightarrow{\epsilon} \subseteq \mathrel{\|\!\!\!\rightarrow} \cdot \mathrel{\leftarrow\!\!\!\|}$ holds. We have $\mathsf{f\ b\ b} \mathrel{\leftarrow\!\!\!\|} \mathsf{f\ a\ a} \xrightarrow{\epsilon} \mathsf{c}$ but $\mathsf{f\ b\ b} \mathrel{\|\!\!\!\rightarrow} \cdot \mathrel{\leftarrow\!\!\!\|} \mathsf{c}$ is not satisfied.

Definition 9 (parallel moves property)
We say an STTRS satisfies the *parallel moves property*, and write $\mathrm{PM}(\to)$, if the inclusion $\mathrel{\leftarrow\!\!\!\|} \cdot \xrightarrow{\epsilon} \subseteq \mathrel{\|\!\!\!\rightarrow} \cdot \mathrel{\leftarrow\!\!\!\|}$ holds.

Lemma 7 states that the parallel moves property is equivalent to the diamond property of a parallel reduction. The parallel moves property is a useful sufficient condition for proving the confluence of orthogonal rewrite systems.

Lemma 10 (confluence by parallel moves)
Every STTRS with the parallel moves property is confluent.

Proof. We have $\mathrm{PM}(\to) \iff \Diamond(\mathrel{\|\!\!\!\rightarrow}) \implies \mathrm{CR}(\to)$ by Lemmata 7 and 5 with $\mathrel{\|\!\!\!\rightarrow}^* = \to^*$, see Fig.1. □

$$\begin{tikzcd}
\cdot \arrow[r, "\epsilon"] \arrow[d, "\parallel"'] \arrow[dr, phantom, "\mathrm{PM}(\to)"] & \cdot \arrow[d, dashed, "\parallel"] \\
\cdot \arrow[r, dashed, "\parallel"'] & \cdot
\end{tikzcd}
\iff
\begin{tikzcd}
\cdot \arrow[r, "\parallel"] \arrow[d, "\parallel"'] \arrow[dr, phantom, "\diamond(\nrightarrow\!\!\!\!\parallel)"] & \cdot \arrow[d, dashed, "\parallel"] \\
\cdot \arrow[r, dashed, "\parallel"'] & \cdot
\end{tikzcd}
\implies
\begin{tikzcd}
\cdot \arrow[r, "*"] \arrow[d, "*"'] \arrow[dr, phantom, "\mathrm{CR}(\to)"] & \cdot \arrow[d, dashed, "*"] \\
\cdot \arrow[r, dashed, "*"'] & \cdot
\end{tikzcd}$$

Fig. 1. confluence by parallel moves

4 Confluence of STTRSs

In this section, we give a simple proof of the confluence of orthogonal STTRSs based on the parallel moves property. We are interested in orthogonal STTRSs, which are of practical importance, although there are confluence results obtained by weakening orthogonality, see, for example, [Hue80]. For the definition of orthogonality, we need the notions of left-linearity and overlap.

Definition 11 (left-linearity)
An STTRS is *left-linear* if none of left-hand sides of rewrite rules contain multiple occurrences of a variable symbol.

Definition 12 (overlap)
An STTRS is *overlapping* if there exist rewrite rules $l \to r$ and $l' \to r'$ without common variable symbols (after renaming) and a non-variable position $p \in \mathsf{Pos}_\mathsf{c}(l) \cup \mathsf{Pos}_\mathsf{a}(l)$ such that
- $l|_p$ and l' are unifiable, and
- if $p = \epsilon$ then $l \to r$ is not obtained from $l' \to r'$ by renaming variable symbols.

Example 13 (overlapping STTRS)
Let C be the set consisting of constant symbols $\mathsf{f}, \mathsf{g}, \mathsf{h}$ of type $o \to o$ and a, b of type o. Let V contain variable symbols F of type $o \to o$ and x of type o. We define

$$R = \left\{ \begin{array}{r} \mathsf{f}\ (F\ x) \to F\ \mathsf{b} \\ \mathsf{g}\ \mathsf{a} \to \mathsf{h}\ \mathsf{b} \end{array} \right\}.$$

The STTRS $\mathcal{R} = (R, V, C)$ is overlapping because the left-hand sides of the rewrite rules are unifiable at an application-position. In this STTRS the term $\mathsf{f}\ (\mathsf{g}\ \mathsf{a})$ has two different normal forms: $\mathsf{g}\ \mathsf{b}\ {}_{\mathcal{R}}\!\leftarrow \mathsf{f}\ (\mathsf{g}\ \mathsf{a}) \to_{\mathcal{R}} \mathsf{f}\ (\mathsf{h}\ \mathsf{b}) \to_{\mathcal{R}} \mathsf{h}\ \mathsf{b}$. Note that the redex $(\mathsf{g}\ \mathsf{a})$ in the initial term is destroyed by the application of the first rewrite rule.

Definition 14 (orthogonality)
An STTRS is *orthogonal* if it is left-linear and non-overlapping.

Now we are ready to give a proof that orthogonal STTRSs are confluent. We first extend the use of parallel reduction to substitutions.

Definition 15 (parallel reduction of a substitution)
Let σ and τ be substitutions and X be a set of variable symbols. We write $\sigma \mathrel{-\!\!\|\!\!\!\!\rightarrow}_{[X]} \tau$ if $\sigma(x) \mathrel{-\!\!\|\!\!\!\!\rightarrow} \tau(x)$ for all $x \in X$.

Lemma 16 (parallel reduction of a substitution)
Let σ and τ be substitutions and t be a term. If $\sigma \mathrel{-\!\!\|\!\!\!\!\rightarrow}_{[X]} \tau$ and $\mathsf{Var}(t) \subseteq X$ then $t\sigma \mathrel{-\!\!\|\!\!\!\!\rightarrow} t\tau$.

Proof. An easy induction on the structure of t. □

Lemma 17 (key properties for confluence)
Let $\mathcal{R} = (R, V, C)$ be an orthogonal STTRS.

(1) $\xleftarrow{\epsilon} \cdot \xrightarrow{\epsilon} \subseteq =$.

(2) For all rewrite rules $l \to r \in R$, substitutions σ, and terms t, if $t \mathrel{\leftarrow\!\!\!\!\|\!\!-} l\sigma \xrightarrow{\epsilon} r\sigma$ and not $l\sigma \xrightarrow{\epsilon} t$ then there exists a substitution τ such that $t = l\tau$ with $\sigma \mathrel{-\!\!\|\!\!\!\!\rightarrow}_{[\mathsf{Var}(l)]} \tau$.

Proof.

(1) Since $\mathcal{R}$ is non-overlapping, we can only use (two renamed versions of) the same rewrite rule in R for rewriting a term at the same position. Hence we always obtain the same term.

(2) Since $\mathcal{R}$ is non-overlapping, there is no term s' with $l_{|p}\sigma \xrightarrow{\epsilon} s'$, for all non-variable position p in l. Hence we have $l_{|p}\sigma \mathrel{-\!\!\|\!\!\!\!\rightarrow} t_{|p}$ for all variable positions p in l. Define the substitution τ by $\tau(x) = t_{|p}$ if $l_{|p} = x$ and $\tau(x) = x$ otherwise. This substitution is well-defined because there are no multiple occurrences of x in l by left-linearity. It is easy to see that $\sigma \mathrel{-\!\!\|\!\!\!\!\rightarrow}_{[\mathsf{Var}(l)]} \tau$ and $t = l\tau$ by construction.

□

Lemma 18 (Parallel Moves Lemma)
Every orthogonal TRS $\mathcal{R} = (R, V, C)$ has the parallel moves property, i.e., $\mathrel{\leftarrow\!\!\!\!\|\!\!-} \cdot \xrightarrow{\epsilon} \subseteq \mathrel{-\!\!\|\!\!\!\!\rightarrow} \cdot \mathrel{\leftarrow\!\!\!\!\|\!\!-}$.

Proof. Suppose $t \mathrel{\leftarrow\!\!\!\!\|\!\!-} s \xrightarrow{\epsilon} u$. We show $t \mathrel{-\!\!\|\!\!\!\!\rightarrow} \cdot \mathrel{\leftarrow\!\!\!\!\|\!\!-} u$. If $s \xrightarrow{\epsilon} t$ then the desired result follows from Lemma 17(1). Otherwise, we do not have $s \xrightarrow{\epsilon} t$. Since there exists a rewrite rule $l \to r \in R$ and a substitution σ such that $s = l\sigma$ and $u = r\sigma$, we know the existence of a substitution τ such that $t = l\tau$ with $\sigma \mathrel{-\!\!\|\!\!\!\!\rightarrow}_{[\mathsf{Var}(l)]} \tau$ by Lemma 17(2). Therefore, $t = l\tau \xrightarrow{\epsilon} r\tau \mathrel{\leftarrow\!\!\!\!\|\!\!-} r\sigma = u$ by Lemma 16 and $\mathsf{Var}(r) \subseteq \mathsf{Var}(l)$. Note that the case $s = t \in V$ is impossible because every variable symbol is in normal form and that the case $s = t \in C$ is contained in the case $s \xrightarrow{\epsilon} u$. □

Note that the Parallel Moves Lemma does not hold for orthogonal higher-order rewrite systems with bound variables as observed in the literature, see

[vO97] and [vR99]. A proof of confluence in such rewrite systems can be found in [MN98]. Now we conclude this section by the main result of this section and an example of its application.

Theorem 19 (confluence by orthogonality)
Every orthogonal TRS is confluent.

Proof. By Lemma 10 and the Parallel Moves Lemma (Lemma 18). □

Example 20 (confluence by orthogonality)
The example STTRS given in Example 3 is confluent because it is orthogonal.

5 Confluence of Conditional STTRSs

In this section, we generalize the confluence result presented in the previous section to the case of conditional rewriting. Bergstra and Klop proved the confluence of first-order orthogonal CTRSs in [BK86]. Their proof depends on the notion of development and the fact that every development is finite. Our result in this section generalizes their result to STTRSs and also simplifies their confluence proof, based on the parallel moves property.

Definition 21 (conditional rewrite rule)
Let $\mathsf{T}(V, C)$ be a set of terms. A *conditional rewrite rule* $l \rightarrow r \Leftarrow c$ consists of a rewrite rule $l \rightarrow r$ and the *conditional part* c. Here c is a possibly empty finite sequence $c = l_1 \approx r_1, \ldots, l_n \approx r_n$ of equations such that every pair of terms l_i and r_i are of the same type and $\mathsf{Var}(r) \subseteq \mathsf{Var}(c)$. If the conditional part is empty, we may simply write $l \rightarrow r$. Let R be a set of conditional rewrite rules. We call $\mathcal{R} = (R, V, C)$ a *conditional* STTRS.

A variable in the right-hand side or in the conditional part of a rewrite rule which does not appear in the corresponding left-hand side is called an extra variable. In this paper, we allow extra variables only in the conditional part but not in the right-hand sides of rewrite rules.

Definition 22 (rewrite relation)
The rewrite relation $\rightarrow_{\mathcal{R}}$ of a conditional STTRS $\mathcal{R} = (R, V, C)$ is defined as follows: $s \rightarrow_{\mathcal{R}} t$ if and only if $s \rightarrow_{\mathcal{R}_k} t$ for some $k \geq 0$. The minimum such k is called the *level* of the rewrite step. Here the relations $\rightarrow_{\mathcal{R}_k}$ are inductively defined:

$$\begin{aligned} \rightarrow_{\mathcal{R}_0} &= \varnothing, \\ \rightarrow_{\mathcal{R}_{k+1}} &= \{\, (t[l\sigma]_p, t[r\sigma]_p) \mid l \rightarrow r \Leftarrow c \in R,\ c\sigma \subseteq \rightarrow^*_{\mathcal{R}_k} \,\}. \end{aligned}$$

Here $c\sigma$ denotes the set $\{\, l'\sigma \approx r'\sigma \mid l' \approx r'$ belongs to $c \,\}$. Therefore $c\sigma \subseteq \rightarrow^*_{\mathcal{R}_k}$ with $c = l_1 \approx r_1, \ldots, l_n \approx r_n$ is a shorthand for $l_1 \rightarrow^*_{\mathcal{R}_k} r_1, \ldots, l_n \rightarrow^*_{\mathcal{R}_k} r_n$. We may abbreviate $\rightarrow_{\mathcal{R}_k}$ to $\rightarrow_k$ if there is no need to make the underlying conditional STTRS explicit.

Properties of conditional STTRSs are often proved by induction on the level of a rewrite step. So, it is useful for proving confluence of orthogonal conditional STTRSs to introduce the parallel reduction relations which are indexed by levels.

Definition 23 (parallel reduction relations indexed by levels)
Let $\mathcal{R}$ be a conditional STTRS. We define $\mathrel{+\!\!\!\!\!\rightarrow}_{\mathcal{R}_k}$ as the smallest relation such that
(1) $t \mathrel{+\!\!\!\!\!\rightarrow}_{\mathcal{R}_k} t$ for all terms t,
(2) if $s \xrightarrow{\epsilon}_{\mathcal{R}_k} t$ then $s \mathrel{+\!\!\!\!\!\rightarrow}_{\mathcal{R}_k} t$, and
(3) if $k \geq j_i$ and $s_i \mathrel{+\!\!\!\!\!\rightarrow}_{\mathcal{R}_{j_i}} t_i$ for $i = 0, \ldots, n$, then $(s_0\, s_1 \cdots s_n) \mathrel{+\!\!\!\!\!\rightarrow}_{\mathcal{R}_k} (t_0\, t_1 \cdots t_n)$.
We may abbreviate $\mathrel{+\!\!\!\!\!\rightarrow}_{\mathcal{R}_k}$ to $\mathrel{+\!\!\!\!\!\rightarrow}_k$ when no confusion can arise.

Observe that $s \mathrel{+\!\!\!\!\!\rightarrow}_{\mathcal{R}} t$ if and only if $s \mathrel{+\!\!\!\!\!\rightarrow}_{\mathcal{R}_k} t$ for some level $k \geq 0$. It is also easy to verify that $\rightarrow_{\mathcal{R}_k} \subseteq \mathrel{+\!\!\!\!\!\rightarrow}_{\mathcal{R}_k} \subseteq \rightarrow^*_{\mathcal{R}_k}$ for all levels $k \geq 0$.

Definition 24 (properties of an ARS with indexes)
Let $\mathcal{A} = (A, \bigcup_{i \in I} \rightarrow_i)$ be an ARS whose rewrite relations are indexed. We use the following abbreviations:

definition	abbreviation
${}_j\!\leftarrow \cdot \rightarrow_k \subseteq \rightarrow_k \cdot {}_j\!\leftarrow$	$\diamond^j_k(\rightarrow)$
${}^*_j\!\leftarrow \cdot \rightarrow^*_k \subseteq \rightarrow^*_k \cdot {}^*_j\!\leftarrow$	$\mathrm{CR}^j_k(\rightarrow)$

Lemma 25 (confluence by simultaneous reduction)
Let $(A, \bigcup_{i \in I} \rightarrow_i)$ and $(A, \bigcup_{i \in I} \hookrightarrow_i)$ be ARSs such that $\hookrightarrow^*_i = \rightarrow^*_i$ for all $i \in I$. We have $\diamond^j_k(\hookrightarrow) \implies {}_j\!\hookleftarrow \cdot \rightarrow^*_k \subseteq \rightarrow^*_k \cdot {}_j\!\hookleftarrow \implies \mathrm{CR}^j_k(\rightarrow)$.

Proof. Straightforward. □

Definition 26 (parallel moves property for conditional STTRSs)
We say a conditional STTRS satisfies the *parallel moves property* with respect to levels j and k, and write $\mathrm{PM}^j_k(\rightarrow)$, if the inclusion ${}_j\!\mathrel{\leftarrow\!\!\!\!\!+} \cdot \xrightarrow{\epsilon}_k \subseteq \mathrel{+\!\!\!\!\!\rightarrow}_k \cdot {}_j\!\mathrel{\leftarrow\!\!\!\!\!+}$ holds.

Lemma 27 (parallel moves for conditional STTRSs)
The following two statements are equivalent, for all $m \geq 0$.
(1) $\mathrm{PM}^j_k(\rightarrow)$ for all j, k with $j + k \leq m$.
(2) $\diamond^j_k(\mathrel{+\!\!\!\!\!\rightarrow})$ for all j, k with $j + k \leq m$.

Proof. The implication (2) $\Rightarrow$ (1) is obvious because $\xrightarrow{\epsilon}_k \subseteq \mathrel{+\!\!\!\!\!\rightarrow}_k$ by definition. For the proof of the implication (1) $\Rightarrow$ (2), suppose statement (1) holds. We show that if $t \;{}_j\!\mathrel{\leftarrow\!\!\!\!\!+}\; s \mathrel{+\!\!\!\!\!\rightarrow}_k u$ and $j + k \leq m$ then there exists a term v such that $t \mathrel{+\!\!\!\!\!\rightarrow}_k v \;{}_j\!\mathrel{\leftarrow\!\!\!\!\!+}\; u$, for all terms s, t, u and levels j, k. The proof is by induction on the structure of s. We distinguish three cases according to the parallel reduction $s \mathrel{+\!\!\!\!\!\rightarrow}_j t$. If $s = t$ then $t \mathrel{+\!\!\!\!\!\rightarrow}_k v = u$ by taking $v = u$. If $s \xrightarrow{\epsilon}_j t$ or $s \xrightarrow{\epsilon}_k u$ then we can use the assumption (1) in both cases because $j + k \leq m$. Otherwise, we have $s = (s_0\, s_1 \cdots s_n)$, $t = (t_0\, t_1 \cdots t_n)$, and $u = (u_0\, u_1 \ldots u_n)$ for some $n \geq 1$ with

$t_i \ {}_{j_i}\!\!\leftarrow\!\!\!\!\|\!\!\!\!\| \ s_i \mathrel{\|\!\!\!\!\|\!\!\!\!\rightarrow_{k_i}} u_i (i = 0, \dots , n)$. Since $j_i+k_i \leq j+k \leq m$, the induction hypothesis yields the existence of terms v_i such that $t_i \mathrel{\|\!\!\!\!\|\!\!\!\!\rightarrow_{k_i}} v_i \ {}_{j_i}\!\!\leftarrow\!\!\!\!\|\!\!\!\!\| \ u_i$, for $i = 0, \dots , n$. Let $v = (v_0\, v_1 \dots v_n)$. Then we have $t \mathrel{\|\!\!\!\!\|\!\!\!\!\rightarrow_k} v \ {}_j\!\!\leftarrow\!\!\!\!\|\!\!\!\!\| \ u$ by definition. □

The following lemma gives a sufficient condition for the confluence of CTRSs.

Lemma 28 (confluence by parallel moves)
If a CTRS satisfies $\mathrm{PM}^j_k(\rightarrow)$ for all levels j and k then $\mathrm{CR}^j_k(\rightarrow)$ holds for all j and k, hence it is confluent.

Proof. By Lemmata 27 and 25 with $\mathrel{\|\!\!\!\!\|\!\!\!\!\rightarrow}^*_i = \rightarrow^*_i$ for all levels i, see Fig.2. □

$$\begin{tikzcd}
\forall j,k\ j+k \leq m & & & \forall j,k\ j+k \leq m & & & \forall j,k\ j+k \leq m & & & \forall j,k\ j+k \leq m & \\
\cdot \arrow[r, "\epsilon", "k"'] \arrow[d, "\|" description, "j"'] \arrow[dr, phantom, "\mathrm{PM}^j_k(\rightarrow)"] & \cdot \arrow[d, dashed, "\|" description, "j"] & \Leftrightarrow & \cdot \arrow[r, "\|", "k"'] \arrow[d, "\|" description, "j"'] \arrow[dr, phantom, "\Diamond^j_k(\mathrel{\|\!\!\!\!\|\!\!\!\!\rightarrow})"] & \cdot \arrow[d, dashed, "\|" description, "j"] & \Rightarrow & \cdot \arrow[r, "*", "k"'] \arrow[d, "\|" description, "j"'] & \cdot \arrow[d, dashed, "\|" description, "*\ j"] & \Rightarrow & \cdot \arrow[r, "*", "k"'] \arrow[d, "*\ j"'] \arrow[dr, phantom, "\mathrm{CR}^j_k(\rightarrow)"] & \cdot \arrow[d, dashed, "*\ j"] \\
\cdot \arrow[r, dashed, "\|", "k"'] & \cdot & & \cdot \arrow[r, dashed, "\|", "k"'] & \cdot & & \cdot \arrow[r, dashed, "k"'] & \cdot & & \cdot \arrow[r, dashed, "k"'] & \cdot
\end{tikzcd}$$

Fig. 2. confluence of CTRSs by parallel moves

Imposing restrictions on reducibility of the right-hand sides of the conditions is important for ensuring the confluence of conditional STTRSs.

Definition 29 (normal conditional STTRS)
Let $\mathcal{R}$ be a conditional STTRS. A term t is called *normal* if it contains no variables and is a normal form with respect to the unconditional version of $\mathcal{R}$. Here the unconditional version is obtained from $\mathcal{R}$ by dropping all conditions. A conditional STTRS is called *normal* if every right-hand side of an equation in the conditional part of a rewrite rule is normal.

We extend the indexed version of a parallel reduction $\mathrel{\|\!\!\!\!\|\!\!\!\!\rightarrow_k}$ to the relation $\mathrel{\|\!\!\!\!\|\!\!\!\!\rightarrow_{k[X]}}$ on substitutions as in the unconditional case (Definition 15). It is easy to verify that the level version of Lemma 16 also holds. Now we are ready for proving the Parallel Moves Lemma for conditional STTRSs. In the conditional case, we must confirm that the conditions are satisfied after the change of the substitution.

Lemma 30 (Parallel Moves Lemma for conditional STTRSs)
Every orthogonal normal conditional STTRS $\mathcal{R} = (R, V, C)$ satisfies $\mathrm{PM}^j_k(\rightarrow)$, i.e., ${}_j\!\!\leftarrow\!\!\!\!\|\!\!\!\!\| \cdot \xrightarrow{\epsilon}_k \subseteq \mathrel{\|\!\!\!\!\|\!\!\!\!\rightarrow_k} \cdot {}_j\!\!\leftarrow\!\!\!\!\|\!\!\!\!\|$, for all levels j and k.

Proof. We show that if $t\ {}_{j}\!\!\mathrel{\leftarrow\!\!\!\!\!+\!\!+}\ s \xrightarrow{\epsilon}_k u$ then there exists a term v such that $t \mathrel{+\!\!+\!\!\!\!\!\rightarrow}_k v\ {}_{j}\!\!\mathrel{\leftarrow\!\!\!\!\!+\!\!+}\ u$. The proof is by induction on $j+k$. The case $j+k=0$ is trivial because $\xrightarrow{\epsilon}_0 = \varnothing$. Suppose $j+k>0$. We distinguish two cases. If $s \xrightarrow{\epsilon}_j t$ then we have $s=u$ because $\mathcal{R}$ is non-overlapping and has no extra variable in the right hand sides of R. Hence we can take $v=s=u$. Consider the case that $s \xrightarrow{\epsilon}_j t$ does not hold. From $s \xrightarrow{\epsilon}_k u$ we know that there exists a conditional rewrite rule $l \rightarrow r \Leftarrow c \in R$ and a substitution σ such that $s = l\sigma$, $u = r\sigma$, and $c\sigma \subseteq \rightarrow^*_{k-1}$. Since $\mathcal{R}$ is non-overlapping, there is no term s' with $l_{|p}\sigma \xrightarrow{\epsilon}_j s'$ for all non-variable positions p in l. Define the substitution τ by $\tau(x) = t_{|p}$ if $l_{|p} = x$ and $\tau(x) = \sigma(x)$ otherwise. This substitution is well-defined by the left-linearity of $\mathcal{R}$. We have $\sigma \mathrel{+\!\!+\!\!\!\!\!\rightarrow}_{j[\mathsf{Var}(l,c)]} \tau$ and $t = l\tau$ by the definition of τ. From $\mathsf{Var}(r) \subseteq \mathsf{Var}(l)$ and the level version of Lemma 16 we obtain $r\sigma \mathrel{+\!\!+\!\!\!\!\!\rightarrow}_j r\tau$. It remains to show that $t = l\tau \mathrel{+\!\!+\!\!\!\!\!\rightarrow}_k r\tau$. So, we will prove $c\tau \subseteq \rightarrow^*_{k-1}$. Let $l' \approx r'$ be an arbitrary condition in c. We have to prove that $l'\tau \rightarrow^*_{k-1} r'\tau$. Since $c\sigma \subseteq \rightarrow^*_{k-1}$, we have $l'\sigma \rightarrow^*_{k-1} r'\sigma$. Moreover, $\sigma \mathrel{+\!\!+\!\!\!\!\!\rightarrow}_{j[\mathsf{Var}(l,c)]} \tau$, $\mathsf{Var}(c) \subseteq \mathsf{Var}(l,c)$, and the level version of Lemma 16 yields that $l'\sigma \mathrel{+\!\!+\!\!\!\!\!\rightarrow}_j l'\tau$. Hence $l'\tau\ {}_{j}\!\!\mathrel{\leftarrow\!\!\!\!\!+\!\!+}\ l'\sigma \rightarrow^*_{k-1} r'\sigma$. From the induction hypothesis and Lemmata 30 and 25, we know the existence of a term v such that $l'\tau \rightarrow^*_{k-1} v\ {}_{j}\!\!\mathrel{\leftarrow\!\!\!\!\!+\!\!+}\ r'\sigma$. Because $\mathcal{R}$ is normal, $r'\sigma = r' = r'\tau = v$. Hence $l'\tau \rightarrow^*_{k-1} r'\tau$. Therefore $l\tau \mathrel{+\!\!+\!\!\!\!\!\rightarrow}_k r\tau$. □

Theorem 31 (confluence by orthogonality)
Every orthogonal normal conditional STTRS is confluent.

Proof. By Lemma 28 and the Parallel Moves Lemma for conditional STTRSs (Lemma 30). □

6 Termination of STTRSs by Interpretation

In this section, we discuss how to prove termination of STTRSs by semantic method. We adapt the semantic proof technique both in the first-order case [Zan94] and in the higher-order case [vdP94]. In a semantic method, terms are interpreted as an element in a well-founded ordered set.

Definition 32 (monotone domain)
Given a non-empty set D_o, called a *base domain*, and a strict partial order (irreflexive transitive relation) $\succ^D_o$ on D_o, we recursively define the set D_τ and the relation $\succ^D_\tau$ on D_τ, for every function type $\tau = \tau_1 \times \cdots \tau_n \rightarrow \tau_0$, as follows:

$$D_\tau = \{\, \varphi : D_{\tau_1} \times \cdots \times D_{\tau_n} \rightarrow D_{\tau_0} \mid \varphi \text{ is monotone} \,\}$$

where a function $\varphi \in D_\tau$ is called *monotone* if

$$\varphi(x_1, \ldots, x_i, \ldots, x_n) \succ^D_{\tau_0} \varphi(x_1, \ldots, y, \ldots, x_n)$$

for all $i \in \{1, \ldots, n\}$ and $x_1 \in D_{\tau_1}, \ldots, x_n \in D_{\tau_n}, y \in D_{\tau_i}$ with $x_i \succ^D_{\tau_i} y$. We write $\varphi \succ^D_\tau \psi$ if $\varphi, \psi \in D_\tau$ and

$$\varphi(x_1, \ldots, x_n) \succ^D_{\tau_0} \psi(x_1, \ldots, x_n)$$

for all $x_1 \in D_{\tau_1}, \ldots, x_n \in D_{\tau_n}$.

Lemma 33 (properties of $\succ_\tau^D$)
Let $\succ_o^D$ be a strict partial order on a base domain D_o.
(1) $\succ_\tau^D$ also is a strict partial order for all types τ.
(2) If $\succ_o^D$ is well-founded, then $\succ_\tau^D$ also is well-founded for all types τ.

Proof. By induction on τ. □

We interpret every constant symbol by an algebra operation on a well-founded ordered set. A valuation gives an interpretation to variables.

Definition 34 (monotone interpretation)
Let $\succ_o^D$ be a strict partial order on a base domain D_o. A *monotone interpretation* I associates every constant symbol c of type τ with its interpretation $c_I \in D_\tau$. A *valuation* is a function which maps a variable symbol of type τ to an element in D_τ.

Let I be a monotone interpretation and α be a valuation. A term t of type τ is interpreted as an element of D_τ as follows: If t is a variable symbol, then $[\![t]\!]_\alpha = \alpha(t)$. If t is a constant symbol, then $[\![t]\!]_\alpha = t_I$. If $t = (t_0\, t_1 \cdots t_n)$ then $[\![t]\!]_\alpha = [\![t_0]\!]_\alpha([\![t_1]\!]_\alpha, \ldots, [\![t_n]\!]_\alpha)$. We define the relation $\succ_I$ on the set of terms as follows: $s \succ_I t$ if and only if s and t are of the same type, say τ, and $[\![s]\!]_\alpha \succ_\tau^D [\![t]\!]_\alpha$ for all valuations α.

The order $\succ_I$ compares two terms by interpretation. If a well-founded order $\succ$ on terms is closed under contexts and substitutions, then $l \succ r$ for all rewrite rules $l \to r \in \mathcal{R}$ suffices to show the termination of $\mathcal{R}$. It turns out that $\succ_I$ is closed under contexts and substitutions.

Lemma 35 (properties of $\succ_I$)
Let $\succ_o^D$ be a strict partial order on a base domain D_o and I be a monotone interpretation.
(1) $\succ_I$ also is a strict partial order.
(2) If $\succ_o^D$ is well-founded, then $\succ_I$ also is well-founded.

Proof. Easy consequences of Lemma 33. □

Lemma 36 (closure under contexts of $\succ_I$)
Let $\succ_o^D$ be a strict partial order on a base domain D_o and I be a monotone interpretation. Then, $\succ_I$ is closed under contexts, i.e., if $s \succ_I t$ then $u[s]_p \succ_I u[t]_p$ for all possible terms u and positions p in u.

Proof. The proof is by induction on p. The case $p = \epsilon$ is trivial. If $p = iq$, then u should have the form $(u_0\, u_1 \cdots u_n)$ and u_0 has function type, say $\tau = \tau_1 \times \cdots \times \tau_n \to \tau_0$. From the induction hypothesis, we know $u_i[s]_q \succ_I u_i[t]_q$. We distinguish two cases. If $i = 0$, then $[\![u_0[s]_q]\!]_\alpha \succ_\tau^D [\![u_0[t]_q]\!]_\alpha$ for all valuations α, by the induction hypothesis. Hence $[\![u[s]_p]\!]_\alpha = [\![(u_0[s]_q\, u_1 \cdots u_n)]\!]_\alpha = [\![u_0[s]_q]\!]_\alpha([\![u_1]\!]_\alpha, \ldots, [\![u_n]\!]_\alpha) \succ_{\tau_0}^D [\![u_0[t]_q]\!]_\alpha([\![u_1]\!]_\alpha, \ldots, [\![u_n]\!]_\alpha) = [\![u_0[t]_q\, u_1 \cdots u_n]\!]_\alpha =$

$[\![u[t]_p]\!]_\alpha$ for all α. Therefore $u[s]_p \succ_I u[t]_p$. Consider the case $i > 0$. By the induction hypothesis, we have $[\![u_i[s]_q]\!]_\alpha \succ^D_\tau [\![u_i[t]_q]\!]_\alpha$ for all valuations α. Hence, $[\![u[s]_p]\!]_\alpha = [\![(u_0 u_1 \cdots u_i[s]_q \cdots u_n)]\!]_\alpha = [\![u_0]\!]_\alpha([\![u_1]\!]_\alpha, \ldots, [\![u_i[s]_q]\!]_\alpha, \ldots [\![u_n]\!]_\alpha) \succ^D_{\tau_0}$ $[\![u_0]\!]_\alpha([\![u_1]\!]_\alpha, \ldots, [\![u_i[t]_q]\!]_\alpha, \ldots, [\![u_n]\!]_\alpha) = [\![(u_0 u_1 \cdots u_i[t]_q \cdots u_n)]\!]_\alpha = [\![u[t]_p]\!]_\alpha$, for all valuations α, by monotonicity. Therefore $u[s]_p \succ_I u[t]_p$. □

Lemma 37 (closure under substitutions of $\succ_I$)
Let $\succ^D_o$ be a strict partial order on a base domain D_o and I be a monotone interpretation.
(1) $[\![t\sigma]\!]_\alpha = [\![t]\!]_{\alpha * \sigma}$ for all terms t, substitutions σ, and valuations α. Here the valuation $\alpha * \sigma$ is defined by $(\alpha * \sigma)(x) = [\![\sigma(x)]\!]_\alpha$.
(2) $\succ_I$ is closed under substitutions, i.e., if $s \succ_I t$ then $s\sigma \succ_I t\sigma$ for all substitutions σ.

Proof. (1) is by structural induction on t. (2) is an easy consequence of (1). □

The following theorem provides a semantic proof method for termination of STTRSs.

Theorem 38 (termination of STTRSs by interpretation)
Let $\mathcal{R} = (R, V, C)$ be an STTRS. If there exists a base domain D_o, a well-founded strict partial order $\succ^D_o$ on D_o, and a monotone interpretation I, such that $[\![l]\!]_\alpha \succ_I [\![r]\!]_\alpha$ for all rewrite rules $l \to r \in R$ and valuations α, then $\mathcal{R}$ is terminating.

Proof. It is easy to see that $\to_\mathcal{R} \subseteq \succ_I$ because we have $[\![l]\!]_\alpha \succ_I [\![r]\!]_\alpha$ for all rewrite rules $l \to r$ and valuations α by assumption, and the relation $\succ_I$ is closed under contexts and substitutions by Lemmata 36 and 37. For the proof by contradiction, suppose $\to_\mathcal{R}$ is not well-founded. So, there is an infinite rewrite sequence $t_0 \to_\mathcal{R} t_1 \to_\mathcal{R} \cdots$. From this sequence and the inclusion $\to_\mathcal{R} \subseteq \succ_I$, we obtain an infinite descending sequence $t_0 \succ_I t_1 \succ_I \cdots$. This contradicts the well-foundedness of the relation $\succ_I$, which follows from Lemma 35. □

Example 39 (termination of STTRSs by interpretation)
Consider again the STTRS $\mathcal{R}$ of Example 3. In order to prove the termination of $\mathcal{R}$, define the base domain as $D_o = \mathbb{N} - \{0, 1\}$, and the order $\succ^D_o$ by using the standard order $>$ on $\mathbb{N}$ as follows: $m \succ^D_o n$ if and only if $m > n$. Constant symbols are interpreted as follows:

$$\begin{array}{ll}
[\,]_I & = 2 \\
:_I(m, n) & = m + n \\
\mathsf{map}_I(\varphi, n) & = n \times \varphi(n) \\
\circ_I(\varphi, \psi)(n) & = \varphi(\psi(n)) + 1 \\
\mathsf{twice}_I(\varphi)(n) & = \varphi(\varphi(n)) + 2
\end{array}$$

Note that $[\,]_I \in D_o$, $:_I \in D_{o \times o \to o}$, $\mathsf{map}_I \in D_{(o \to o) \times o \to o}$, $\circ_I \in D_{(o \to o) \times (o \to o) \to (o \to o)}$, and $\mathsf{twice}_I \in D_{(o \to o) \to (o \to o)}$ are satisfied. It is easy to see that $[\![l]\!]_\alpha \succ_I [\![r]\!]_\alpha$ for all rewrite rules $l \to r \in R$ and valuations α. Therefore $\mathcal{R}$ is terminating.

The converse of Theorem 38 holds for the first order case [Zan94]. However, for STTRSs, the converse does not hold as shown in the following example.

Example 40 (incompleteness of the semantic proof method)
Let $C = \{0^o, 1^o, \mathsf{g}^{o \to o}\}$, the set of variable symbols V contain $F^{o \to o}$, and $R = \{\mathsf{g}\,(F\,0\,1) \to \mathsf{g}\,(F\,1\,0)\}$. It is not difficult to see that the STTRS (R, V, C) is terminating.

Suppose there is a base domain D_o, a well-founded strict partial order $\succ_o^D$ on D_o, and a monotone interpretation I, such that $[\![l]\!]_\alpha \succ_I [\![r]\!]_\alpha$ for all rewrite rules $l \to r \in R$ and valuations α. We have $\mathsf{g}_I(f(0_I, 1_I)) \succ_o^D \mathsf{g}_I(f(1_I, 0_I))$ for all monotone functions $f \in D_{o \times o \to o}$. Let h be an arbitrary monotone function in $D_{o \times o \to o}$. Then, the function h' defined by $h'(x, y) = h(y, x)$ is also monotone and hence $h' \in D_{o \times o \to o}$. Therefore, $\mathsf{g}_I(h(0_I, 1_I)) \succ_o^D \mathsf{g}_I(h(1_I, 0_I)) = \mathsf{g}_I(h'(0_I, 1_I)) \succ_o^D \mathsf{g}_I(h'(1_I, 0_I)) = \mathsf{g}_I(h(0_I, 1_I))$. This contradicts the irreflexivity of $\succ_o^D$.

At present, it is not known whether there is a complete proof method for termination of STTRSs based on monotone interpretation.

7 Concluding Remarks

We have proposed simply typed term rewriting systems (STTRSs), which is close to the format of functional programming languages with pattern matching. For proving the confluence of orthogonal rewrite systems, we introduced the parallel moves property, which is a useful sufficient condition obtained by localizing the test for the diamond property of a parallel reduction. We proved the confluence of orthogonal STTRSs and orthogonal normal conditional STTRSs. We also provided a semantic method for proving termination of STTRSs based on monotone interpretation.

Since the class of (conditional) STTRSs is a proper extension of the first-order case, all known results for the first-order TRSs can be applied to the subclass of our (conditional) STTRSs. We can also expect that many known results for the first-order TRSs can be lifted to the higher-order case without difficulty, because the behaviour of our higher-order extension is very close to that of the first-order (conditional) TRSs.

Suzuki et al. gave a sufficient condition for the confluence of orthogonal first-order conditional TRSs possibly with extra variables in the right-hand sides of the rewrite rules [SMI95]. The author conjectures that their result can be extended to the higher-order case.

Acknowledgments

I wish to thank Aart Middeldorp, Femke van Raamsdonk, and Fer-Jan de Vries for their comments to the preliminary version of this paper. I benefitted from a discussion with Jaco van de Pol and Hans Zantema, which results in Example 40. My thanks are also due to the anonymous referees for their helpful comments.

References

[BK86] J.A. Bergstra and J.W. Klop. Conditional rewrite rules: Confluence and termination. *Journal of Computer and System Science*, 32:323–362, 1986.

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

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 243–320. The MIT Press, 1990.

[Gra96] B. Gramlich. Confluence without termination via parallel critical pairs. In *Proceedings of the 21st International Colloquium on Trees in Algebra and Programming*, 1996. Lecture Notes in Computer Science 1059, pp. 211-225.

[HS86] J.R. Hindley and J.P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.

[Hue80] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the Association for Computing Machinery*, 27(4):797–821, 1980.

[JO91] J.-P. Jouannaud and M. Okada. Executable higher-order algebraic specification languages. In *Proceedings of the 6th IEEE Symposium on Logic in Computer Science*, pages 350–361, 1991.

[Klo80] J.W. Klop. *Combinatory Reduction Systems*. PhD thesis, Rijks-universiteit, Utrecht, 1980.

[Klo92] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 1, pages 1–116. Oxford University Press, 1992.

[MN98] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192:3–29, 1998.

[NP98] T. Nipkow and C. Prehofer. *Higher-Order Rewriting and Equational Reasoning*, volume I, pages 399–430. Kluwer, 1998.

[SMI95] T. Suzuki, A. Middeldorp, and T. Ida. Level-confluence of conditional rewrite systems with extra variables in right-hand sides. In *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, 1995. Lecture Notes in Computer Science 914, pp. 179–193.

[vdP94] J. van de Pol. Termination proofs for higher-order rewrite systems. In *Proceedings of the 1st International Workshop on Higher-Order Algebra, Logic and Term Rewriting*, 1994. Lecture Notes in Computer Science 816, pp. 305–325.

[vO97] V. van Oostrom. Developing developments. *Theoretical Computer Science*, 175:159–181, 1997.

[vR99] F. van Raamsdonk. Higher-order rewriting. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications*, 1999. Lecture Notes in Computer Science 1631, pp. 220–239.

[Zan94] H. Zantema. Termination of term rewriting: Interpretation and type elimination. *Journal of Symbolic Computation*, 17:23–50, 1994.

Refinements of Lazy Narrowing for Left-Linear Fully-Extended Pattern Rewrite Systems

Mircea Marin[1], Taro Suzuki[2], and Tetsuo Ida[1]

[1] Institute of Information Sciences and Electronics
University of Tsukuba, Tsukuba 305-8573, Japan
{mmarin,ida}@score.is.tsukuba.ac.jp
[2] Department of Computer Software
University of Aizu, Aizu Wakamatsu 965-8580, Japan
taro@u-aizu.ac.jp

ISE-TR-01-180

Abstract. Lazy narrowing is a general E-unification procedure for equational theories presented by confluent term rewriting systems. It has been deeply studied in the first order case and various higher-order extensions have been proposed in an attempt to improve its expressive power. Such extensions suffer from huge search space in guessing the solutions of variables of functional type. For practical purposes, the need to reduce the search space of solutions is of paramount importance.
In this paper we introduce HOLN, a higher-order lazy narrowing calculus for E-unification in theories presented by pattern rewrite systems. The calculus is designed to deal with both oriented and unoriented equations, and keeps track of the variables which are to be bound to normalized solutions. We discuss the operating principle of HOLN, its main properties, and propose refinements to reduce its search space for solutions. Our refinements are defined for classes of left-linear fully-extended pattern rewrite systems which are widely used in higher-order functional logic programming.

1 Introduction

Lazy narrowing is a general E-unification procedure for equational theories that are presented by confluent term rewriting systems. It has been extensively studied in the first-order case (see, e.g., [10, 11]) and serves as computational model of many functional logic programming (FLP) languages. Motivated by functional programming, many higher-order extensions of the FLP programming style have been proposed. Naturally, these extensions impose suitable generalizations of the underlying computational model to the higher-order case. Since higher-order constructs provide support for concise and natural formulations of many real-world problems, this research field has attracted much interest in recent years (see, e.g., [4, 7, 8, 13, 15]). Of particular interest is the framework suggested in [15] for E-unification in theories presented by pattern rewrite systems. Among the main benefits of adopting this framework, we mention:

1. the expressive power of FLP is extended with lambda abstractions and variables of functional type,
2. many higher-order generalizations of the properties of first-order lazy narrowing depend on the properties (confluence, determinism, termination, etc.) of the underlying rewrite relation. Rewriting with pattern rewrite systems preserves properties of first-order term rewriting which are crucial in lifting the essential properties of first-order lazy narrowing to the higher-order case.

The main difficulty in the design of a suitable computational model for higher-order FLP is harnessing the high nondeterminism of guessing the solutions of variables of functional type without loosing completeness. There are at least two ways to overcome this difficulty: (a) we restrict to certain classes of rewrite systems, and (b) we restrict to certain classes of goals to be solved. It is important to identify restrictions which preserve the possibility to formulate large classes of problems in an easy and convenient way.

In this paper we are concerned with a calculus inspired by the calculus LN proposed by Prehofer [15]. We call this calculus *higher-order lazy narrowing calculus* (HOLN for short). HOLN differs from LN in the following respects:

1. LN is designed to solve goals consisting of oriented equations. HOLN can solve goals made of both oriented and unoriented equations.
2. LN regards all equations between λ-terms with free variable at head position as constraints. We call these equations *flex/flex equations*. Since solving flex/flex equations is highly nondeterministic, LN doesn't solve them. This decision is motivated by the fact that flex/flex equations are always solvable, and this information is often sufficient (e.g., in theorem proving). By contrast, HOLN solves certain flex/flex equations without increasing the nondeterminism of computation. As a consequence, HOLN can compute more detailed answers.
3. The inference rules of LN depend only on the syntactic structure of goals and rewrite rules. In addition, HOLN takes into account the fact that certain goal variables must be bound to normalized solutions. This additional

information allows us to reduce the high nondeterminism of guessing bindings of normalized variables. We claim that the runtime overhead of keeping track of the normalized variables in the goal pays off in comparison with the reduction of nondeterminism enabled by this information.

The structure of the paper is as follows. In Section 2 we introduce our main notions and notations. In Section 3 we formally introduce HOLN together with its main properties, i.e. soundness and completeness. In Section 4 we introduce four refinements of HOLN for left-linear fully-extended PRSs (LEPRSs for short), which reduce the *don't know* nondeterminism due to the selection of the inference rule to be applied next. Finally, in Section 5 we draw some conclusions and directions of future work.

2 Preliminaries

We first describe the meta-language of simply-typed λ-calculus. The notation is roughly consistent with [2, 9, 15].

2.1 The Simply-Typed λ-Calculus

Starting with a fixed set of base types $\mathcal{B}$, the set of all types $\mathcal{T}$ is the closure of $\mathcal{B}$ under the function space constructor $\rightarrow$. The letter τ ranges over types. Function types associate to the right, i.e., we parse $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ as $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$. We write $\overline{\tau_n} \rightarrow \tau$ instead of $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$, if τ is a base type.

Terms are generated as usual, by λ-abstraction and application, from a set of *typed variables* $\mathcal{V} = \bigcup_{\tau \in \mathcal{T}} \mathcal{V}_\tau$ and a set of *typed function symbols* $\mathcal{F} = \bigcup_{\tau \in \mathcal{T}} \mathcal{F}_\tau$, where $\mathcal{V}_\tau \cap \mathcal{V}_{\tau'} = \mathcal{F}_\tau \cap \mathcal{F}_{\tau'} = \emptyset$ if $\tau \neq \tau'$. We assume that $\mathcal{V}_\tau$ is countable for any type τ. We denote the *application* of two terms s, t by $(s\ t)$, and the *abstraction* of a term t over a variable x by $\lambda x.t$. An occurrence of a variable x in a term t is *bound* if it occurs below a binder for x, i.e., the occurrence of x is in a subterm $\lambda x.t'$ of t. Otherwise it is *free*. Variables with free and bound occurrences in a term t will be denoted by $\mathcal{FV}(t)$ and $\mathcal{BV}(t)$ respectively.

A *type judgement* that a term t is of type τ is written as $t : \tau$. The following inference rules inductively define the set of simply-typed λ-terms:

$$\frac{a \in \mathcal{F}_\tau \cup \mathcal{V}_\tau}{a : \tau} \quad \frac{s : \tau \rightarrow \tau' \quad t : \tau}{(s\ t) : \tau'} \quad \frac{x : \tau \quad s : \tau'}{(\lambda x.s) : \tau \rightarrow \tau'}.$$

In the sequel we consider only simply-typed λ-terms. We denote by $\mathcal{T}(\mathcal{F}, \mathcal{V})$ the set of simply-typed λ-terms, and by *type*(t) the type of a simply-typed λ-term t.

The following naming conventions are used in the sequel:

sets of finite variables in $\mathcal{V}$:	V, W
variables or function symbols:	a
bound variables or function symbols:	v
simply-typed λ-terms:	l, r, s, t, u
constants in $\mathcal{F}$:	f, g
bound variables:	x, y, z
free variables:	X, Y, Z, H
non-negative integers:	i, j, k, m, n, N

To ease the notation, we also adopt the following abbreviations:

$\overline{ob_n}$ for a sequence of syntactic objects $ob_1, \ldots, ob_n$ where $n \geq 0$; the symbol $\Box$ denotes the empty sequence
$\lambda\overline{x_n}.s$ for $\lambda x_1.\ldots.\lambda x_n.s$
$a(\overline{s_n})$ for $((\cdots(a\ s_1)\cdots)\ s_n)$

For instance, $\lambda\overline{x_m}.f(\overline{s_n})$ stands for $\lambda x_1 \ldots \lambda x_m.((\cdots(f\ s_1)\cdots)\ s_n)$. The subscripts m and n will be omitted when irrelevant or understood from the context.

Let $s[t/X]$ denote the result of replacing each free occurrence of X in s by t. The *conversion rules* in λ-calculus are defined as follows:

(α-conversion) If $y \notin \mathcal{FV}(t) \cap \mathcal{BV}(t)$ and $type(y) = type(x)$ then $\lambda x.t \succ_\alpha \lambda y.(t[y/x])$,
(β-conversion) $(\lambda x.s)\ t \succ_\beta s[t/x]$,
(η-conversion) If $x \notin \mathcal{FV}(t)$ then $(\lambda x.(t\ x)) \succ_\eta t$.

If we denote by $t[l]$ a λ-term with a distinguished occurrence of a subterm l, then let $t[r]$ denote the result of replacing the single subterm l by the term r, where $type(l) = type(r)$. We define the α-*reduction* relation $\longrightarrow_\alpha$ as

$$t[l] \longrightarrow_\alpha t[r] \ \text{ iff } \ l \succ_\alpha r.$$

The β-*reduction relation* $\longrightarrow_\beta$ and η-*reduction relation* $\longrightarrow_\eta$ are defined similarly. We define $\longrightarrow_{\beta\eta}$ as $\longrightarrow_\beta \cup \longrightarrow_\eta$. For each of these reduction relations, we also define the symmetric closure $\longleftrightarrow_\phi$, the transitive closure $\longrightarrow_\phi^+$, the reflexive-transitive closure $\longrightarrow_\phi^*$, and the reflexive-symmetric-transitive closure $\longleftrightarrow_\phi^*$ in the obvious fashion ($\phi \in \{\alpha, \beta, \eta\}$). The relations $\longleftrightarrow_\beta^*$, $\longleftrightarrow_\eta^*$, and $\longleftrightarrow_{\beta\eta}^*$ are called β-, η-, and $\beta\eta$-*equivalence* respectively. Since the simply-typed λ-calculus is confluent and terminating with respect to β-reduction (respectively η-reduction) [1], every term t has a normal form which is denoted by $t{\downarrow_\beta}$ (respectively $t{\downarrow_\eta}$). The β-*normal form* (respectively η-*normal form*) of a term t is denoted by $t{\downarrow_\beta}$ (respectively $t{\downarrow_\eta}$). Let t be in β-normal form (i.e., $t = t{\downarrow_\beta}$). Then t is of the form $\lambda\overline{x_m}.a(\overline{s_n})$, where $a \in \mathcal{F} \cup \mathcal{V}$ is called the *head* of t, denoted by head(t). The η-*expanded form* of $t = \lambda\overline{x_m}.a(\overline{s_n})$ is recursively defined by

$$t{\uparrow_\eta} = \lambda\overline{x_{m+k}}.a(\overline{s_n{\uparrow_\eta}}, x_{m+1}{\uparrow_\eta}, \ldots, x_{m+k}{\uparrow_\eta})$$

where $t : \overline{\tau_{m+k}} \to \tau$ and $x_{m+1}, \ldots, x_{m+k} \notin \mathcal{FV}(\overline{s_n})$. We call $t{\downarrow_\beta}{\uparrow^\eta}$ the *long $\beta\eta$-normal form* of a term t, also written $t{\updownarrow_\beta^\eta}$. A term t is in *long $\beta\eta$-normal form* if $t = t{\updownarrow_\beta^\eta}$.

A term t in long $\beta\eta$-normal form is called *flex* if head(t) is a free variable, and *rigid* otherwise.

We will in general assume that terms are in long $\beta\eta$-normal form and that the transformation of a term into its long $\beta\eta$-normal form is an implicit operation, e.g., when applying a substitution to a term (see next). We will also identify α-equivalent terms and assume that bound variables with different binders have different names. This identification can be achieved at syntactic level if we adopt the de Bruijn representation of λ-terms [3]. Since $s \leftrightarrow_{\alpha\beta\eta} t$ iff $s\updownarrow_{\beta}^{\eta} \leftrightarrow_{\alpha} t\updownarrow_{\beta}^{\eta}$ [5], we can detect the $\alpha\beta\eta$-equivalence of two terms by comparing the de Bruijn representations of their long $\beta\eta$-normal forms.

The *size* $|t|$ of a term t in long $\beta\eta$-normal form is the number of symbols occurring in t, not counting binders. Formally, $|\lambda x.t| = |t|$ and $|a(\overline{t_n})| = 1+\Sigma_{i=1}^{n}|t_i|$.

A *position* is a sequence of natural numbers identifying a subterm in a term. The set $Pos(t)$ of positions in a term t is defined inductively as follows: $Pos(a) = \{\epsilon\}$ if $a \in \mathcal{V} \cup \mathcal{F}$; $Pos(\lambda x.t) = \{\varepsilon\} \cup \{1{\cdot}q \mid q \in Pos(t)\}$; and $Pos(a(\overline{s_n})) = \{\epsilon\} \cup \bigcup_{i=1}^{n}\{i{\cdot}q \mid q \in Pos(s_i)\}$. Here ε denotes the empty sequence. If $p \in Pos(t)$, then $t_{|p}$ denotes the subterm of s at position p, and $t[s]_p$ denotes the term obtained from t by replacing its subterm at position p by a term s of appropriate type.

Let $p \in Pos(s)$. The sequence $\mathcal{BV}(s,p)$ of λ-abstracted variables on the path to p in s is defined inductively as:

- $\mathcal{BV}(s,\varepsilon) = \Box$,
- $\mathcal{BV}(a(\overline{s_n}), i{\cdot}p) = \mathcal{BV}(s_i,p)$,
- $\mathcal{BV}(\lambda x.t, 1{\cdot}p) = x, \mathcal{BV}(t,p)$.

A *substitution* is a map $\theta : \mathcal{V} \to \mathcal{T}(\mathcal{F},\mathcal{V})$ such that:

(a) $type(\theta(X)) = type(X)$ for all $X \in \mathcal{V}$,
(b) $Dom(\theta) := \{X \in \mathcal{V} \mid X \neq \theta(X)\}$, called the *domain* of θ, is finite.

We frequently identify θ with the set $\{X \mapsto \theta(X) \mid X \in Dom(\theta)\}$ of *variable bindings*. We denote the set $\bigcup_{X \in Dom(\theta)} \mathcal{FV}(\theta(X))$ of free variables *introduced* by θ by $Ran(\theta)$. We also denote the *codomain* $\{\theta(X) \mid X \in Dom(\theta)\}$ of θ by $Cod(\theta)$. The empty substitution is denoted by ϵ, and the set of all substitutions by $Subst(\mathcal{F},\mathcal{V})$. Two substitutions θ_1 and θ_2 are *equal on* V, notation $\theta_1 = \theta_2\ [V]$, if $\theta_1(X) = \theta_2(X)$ for all $X \in V$. The *restriction of a substitution* θ to V, denoted by $\theta\lceil_V$, is defined by $\theta\lceil_V(X) = \theta(X)$ if $X \in V$, and $\theta\lceil_V(X) = X$ otherwise.

The *application* of a substitution $\theta = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ to a term t, denoted by $t\theta$, is defined as $[t_n/X_n]\ldots[t_1/X_1]t$. This notation is extended to other syntactic constructs over λ-terms (e.g., sequences of terms, equations, etc.) in the obvious way. For example, if $\overline{t_n}$ is a sequence of terms, then $\overline{t_n}\theta$ denotes the sequence of terms $t_1\theta, \ldots, t_n\theta$.

The *composition* $\theta_1\theta_2$ of two substitutions θ_1, θ_2 is defined as $\theta_1\theta_2(X) := (X\theta_1)\theta_2$. A substitution θ_1 is *more general than* a substitution θ_2 over a set of variables V, notation $\theta_1 \leq \theta_2\ [V]$, if $\theta_1\gamma = \theta_2\ [V]$ for some substitution γ. θ_1 and θ_2 are incomparable over V if neither $\theta_1 \leq \theta_2\ [V]$ nor $\theta_2 \leq \theta_1\ [V]$.

Two terms s and t are called *unifiable* is there exists a substitution θ such that $s\theta = t\theta$. Such a θ is called a *unifier* of terms s and t.

Let $V \in \mathcal{P}_{fin}(\mathcal{V})$. A *renaming* away from V is a map $\rho : \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ with:

- $Dom(\rho) := \{X \in \mathcal{V} \mid X \neq \rho(X)\} \in \mathcal{P}_{fin}(\mathcal{V})$,
- $\{t\downarrow_\eta \mid t \in Cod(\rho)\} \subset \mathcal{V}$, where $Cod(\rho) := \{\rho(X) \mid X \in Dom(\rho)\}$,
- $\rho(X) \neq \rho(Y)$ for all $X, Y \in Dom(\rho)$ with $X \neq Y$, and
- $Ran(\rho) \cap V = \emptyset$, where $Ran(\rho) := \bigcup_{X \in Dom(\rho)} \mathcal{FV}(\rho(X))$.

E.g., $\rho = \{X \mapsto \lambda x.H_1(x), Y \mapsto H_2\}$ is a renaming away from $V = \{X, Y\}$.

2.2 Pattern Rewrite Systems

The following subclass of simply-typed λ-terms was introduced by Miller [12] and is often called higher-order pattern in the literature.

Definition 1 (Pattern [12]). *A* higher-order pattern *(*pattern *for short) is a term t in which every subterm of the form $X(\overline{u_n})$ with $X \in \mathcal{FV}(t)$ has $\overline{u_n\downarrow_\eta}$ a sequence of distinct bound variables.*

For example, the terms $\lambda x, y, z.X(z, x)$ and $\lambda x, y, z.f(X(x, z), Y(x))$ are patterns. The terms $\lambda x, y.X(x, y, x)$ and $\lambda x, y.f(X(x, g))$ are not patterns.

Patterns have the remarkable property that unification is unitary. Moreover, if two patterns are unifiable then a most general unifier can be computed in linear time [16]. This result shows that unification with patterns behaves similar to the first-order case.

Definition 2. *A* fully extended pattern *is a pattern t such that for all $p \in Pos(t)$, if $t|_p = X(\overline{u_n})$ with $X \in \mathcal{FV}(t)$ then $\overline{u_n\downarrow_\eta}$ is a permutation of $\mathcal{BV}(t, p)$.*

For instance, the pattern $\lambda x, y, z.X(x, z, y)$ is fully-extended, but the pattern $\lambda x, y, z.f(X(x, y))$ is not.

Definition 3 (Pattern rewrite system). *A* pattern rewrite system *(*PRS *for short) is a set $\mathcal{R}$ of pairs $l \to r$ such that*

(c_1) *l and r are λ-terms of the same base type,*
(c_2) *$\mathcal{FV}(r) \subseteq \mathcal{FV}(l)$,*
(c_3) *l is a pattern of the form $f(\overline{l_n})$.*

A fully extended pattern rewrite system *(*FPRS *for short) is a pattern rewrite system $\mathcal{R}$ which satisfies the additional condition:*

(c_4) *$\forall(l \to r) \in \mathcal{R}$, l is a fully extended pattern.*

In the sequel we assume given a PRS $\mathcal{R}$. We regard $\mathcal{F}$ as the disjoint union $\mathcal{F}_d \uplus \mathcal{F}_c$, where $\mathcal{F}_d = \{f \in \mathcal{F} \mid \exists(f(\overline{l_n}) \to r) \in \mathcal{R}\}$, and $\mathcal{F}_c = \mathcal{F} \setminus \mathcal{F}_d$. The elements of $\mathcal{F}_d$ are called *defined symbols*, and the elements of $\mathcal{F}_c$ are called *(data) constructors*.

Definition 4 (Rewriting). *If* $(l \to r) \in \mathcal{R}$ *and* $p \in Pos(s)$, *we define a* rewrite step *from* s *to* t *as*

$$s \to^{l\to r}_{p,\theta} t :\Leftrightarrow s|_p = l\theta \wedge t = s[r\theta]_p.$$

We often omit some of the parameters $p, \theta, l \to r$ *and may write* $s \to_{\mathcal{R}} t$ *instead. The relation* $\to_{\mathcal{R}}$ *is called the* rewrite relation *induced by* $\mathcal{R}$ *on* $\mathcal{T}(\mathcal{F}, \mathcal{V})$.

We mention below an equivalent definition of rewriting which takes into account the free variables in $s|_p$ which were bound in s. This new definition is based on the notion of *lifter*.

Definition 5 (Lifter). *An* $\overline{x_k}$-lifter *of a term* t *(respectively rewrite rule* $l \to r$*)* away from V *is a substitution* $\sigma = \{X \mapsto \rho(X)(\overline{x_k}) \mid X \in \mathcal{FV}(t)\}$ *where* ρ *is a renaming with* $Dom(\rho) = \mathcal{FV}(t)$ *(respectively* $Dom(\rho) = \mathcal{FV}(l)$*),* $Ran(\rho) \cap V = \emptyset$ *and* $\rho(X) : \overline{\tau_{k+m}} \to \tau$ *if* $x_1 : \tau_1, \ldots, x_k : \tau_k$ *and* $X : \tau_{k+1} \to \ldots \to \tau_{k+m} \to \tau$.

For example, $\{X \mapsto Y(x)\}$ is an x-lifter of $f(X)$ away from any set $V \subseteq \mathcal{V}\backslash\{Y\}$.

An $\overline{x}$*-lifted rewrite rule* of a rewrite rule $l \to r$ away from V is an expression of the form $l\sigma \to r\sigma$ where σ is an $\overline{x}$-lifter of $l \to r$.

The following definition of rewriting can be proved to be equivalent to Definition 4.

Definition 6 (Rewriting). *A* rewrite step *from* $\lambda\overline{x_j}.s$ *to* $\lambda\overline{x_j}.t$ *is a relation defined recursively as follows:*

$$\lambda\overline{x_j}.s \to_{\mathcal{R}} \lambda\overline{x_j}.t :\Leftrightarrow \lambda\overline{x_k}.(s|_p) = \lambda\overline{x_k}.l\theta$$

where:

- $p \in Pos(s)$ *and* $\overline{x_k} = \mathcal{BV}(\lambda\overline{x_j}.s, p)$,
- $l \to r$ *is an* $\overline{x_k}$*-lifted rewrite rule away from* $\mathcal{FV}(s)$.

We denote by $\to^*_{\mathcal{R}}$ the reflexive-transitive closure of $\to_{\mathcal{R}}$, and by $\leftrightarrow^*_{\mathcal{R}}$ the reflexive-symmetric-transitive closure of $\to_{\mathcal{R}}$. Two terms s and t are $\mathcal{R}$*-joinable*, notation $s \downarrow_{\mathcal{R}} t$, if there exists a term u such that $s \to^*_{\mathcal{R}} u$ and $t \to^*_{\mathcal{R}} u$. $\mathcal{R}$ is *confluent* if whenever $s \to^*_{\mathcal{R}} l$ and $s \to^*_{\mathcal{R}} r$, we have $l \downarrow_{\mathcal{R}} r$. $\mathcal{R}$ is *left-linear* if there is no rewrite rule $(l \to r) \in \mathcal{R}$ with multiple occurrences of a free variable in l.

A term s is $\mathcal{R}$*-normalized* if there is no rewrite step $s \to_{\mathcal{R}} t$. A substitution θ is $\mathcal{R}$-normalized if $\theta(X)$ is $\mathcal{R}$-normalized for any $X \in Dom(\theta)$.

$\mathcal{R}$ induces an equivalence relation $=_{\mathcal{R}}$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$, which is the least equivalence relation induced by the following axioms and inference rules:

$$\frac{}{t =_{\mathcal{R}} t} \quad \frac{s =_{\mathcal{R}} t}{t =_{\mathcal{R}} s} \quad \frac{s =_{\mathcal{R}} t \quad t =_{\mathcal{R}} u}{s =_{\mathcal{R}} u} \quad \frac{s =_{\mathcal{R}} t}{\lambda x.s =_{\mathcal{R}} \lambda x.t} \quad \frac{s =_{\mathcal{R}} s' \quad t =_{\mathcal{R}} t'}{(s\ t) =_{\mathcal{R}} (s'\ t')}$$

$$\frac{(l \to r) \in \mathcal{R}}{l =_{\mathcal{R}} r} \quad \frac{s \leftrightarrow^*_{\beta\eta} t}{s =_{\mathcal{R}} t}$$

It has been shown [18] that $=_{\mathcal{R}}$ coincides with the model-theoretical semantics for higher-order equational logic. Moreover, we have the following relationship between rewriting and equational logic [9]:

$$s =_{\mathcal{R}} t \Leftrightarrow s\updownarrow^{\eta}_{\beta} \leftrightarrow^*_{\mathcal{R}} t\updownarrow^{\eta}_{\beta}. \tag{1}$$

where α is the label of the inference rule, e is the selected equation, θ is the substitution computed in the inference step, $W' = \mathcal{FV}(W\theta)$, and E is a sequence of equations whose elements are called the *descendants* of e. If e' is an equation in E_1 or E_2, then e' has only one descendant in the inference step, namely the corresponding equation $e'\theta$ in $E_1\theta$ or $E_2\theta$. We often omit some of the subscripts α, e, θ of an inference step when they are irrelevant or understood from the context.

We call *C-step* an inference step of a calculus C. We will denote by *step*(C) the set of inference steps of a calculus C. A *C-derivation* is a (possibly empty) sequence of C-steps

$$E\lfloor_W = E_0\lfloor_{W_0} \Rightarrow_{\alpha_1,\theta_1} E_1\lfloor_{W_1} \Rightarrow_{\alpha_2,\theta_2} \cdots \Rightarrow_{\alpha_n,\theta_n} E_n\lfloor_{W_n}$$

abbreviated $E_0\lfloor_{W_0} \Rightarrow_\theta^n E_n\lfloor_{W_n}$ or simply $E_0\lfloor_{W_0} \Rightarrow_\theta^* E_n\lfloor_{W_n}$, where $\theta = \theta_1 \ldots \theta_n$. A *$C$-refutation* is a C-derivation $E\lfloor_W \Rightarrow_\theta^* F\lfloor_{W'}$ for which there is no C-step starting with $F\lfloor_{W'}$. We define

$$Ans_{\mathcal{R}}^{C}(E\lfloor_W) = \{\langle\theta, F\lfloor_{W'}\rangle \mid \exists\ C\text{-refutation } E\lfloor_W \Rightarrow_\theta^* F\lfloor_{W'}\}.$$

In the sequel se adopt the following naming conventions:

equations:	$e, e', \ldots, e_1, e_2, \ldots$
sequences of equations:	$E, E', \ldots, E_1, E_2, \ldots$
sequences of flex equations:	F
C-steps:	$\pi, \pi', \ldots, \pi_1, \pi_2, \ldots$
C-derivations:	$\Pi, \Pi', \ldots, \Pi_1, \Pi_2, \ldots$

In the sequel we will introduce several higher-order lazy narrowing calculi and analyze their main properties. To simplify their presentation, we adopt the following conventions:

- $s \approx^{-1} t$ stands for $t \approx s$, and $s \rhd^{-1} t$ stands for $t \rhd s$,
- for any binary symbol $\bowtie$, we abbreviate by $\overline{u_n \bowtie v_n}$ a sequence of expressions $u_1 \bowtie v_1, \ldots, u_n \bowtie v_n$. For example, $\overline{X_n \mapsto t_n}$ denotes the sequence of variable bindings $X_1 \mapsto t_1, \ldots, X_n \mapsto t_n$, whereas $\overline{s_n \rhd t_n}$ denotes the sequence of equations $s_1 \rhd t_1, \ldots, s_n \rhd t_n$.
- whenever convenient, we relax the convention of writing terms in long $\beta\eta$-normal form, but keep the convention that all written terms are β-normal forms,
- $H, H_1, H_2, \ldots$ denote distinct fresh variables; also, the sequences $\overline{y_m}$, $\overline{y_n}$, $\overline{y'_n}$ and $\overline{z_p}$ are assumed to consist of distinct bound variables.

3.1 The Calculus HOLN

HOLN consists of three groups of inference rules: *preunification rules, narrowing rules,* and *rules for removal of flex/flex equations.*

Preunification rules

[i] **Imitation.**
If $\simeq \in \{\approx, \approx^{-1}, \rhd, \rhd^{-1}\}$ then

$$(E_1,\ \lambda\overline{x}.X(\overline{s_m}) \simeq \lambda\overline{x}.g(\overline{t_n}), E_2)|_W \Rightarrow_{[i],\theta} (E_1, \overline{\lambda\overline{x}.H_n(\overline{s_m}) \simeq \lambda\overline{x}.t_n}, E_2)\theta|_{W'}$$

where $\theta = \{X \mapsto \lambda\overline{y_m}.g(\overline{H_n(\overline{y_m})})\}$.

[p] **Projection.**
If $\lambda\overline{x}.t$ is rigid and $\simeq \in \{\approx, \approx^{-1}, \rhd, \rhd^{-1}\}$ then

$$(E_1,\ \lambda\overline{x}.X(\overline{s_m}) \simeq \lambda\overline{x}.t, E_2)|_W \Rightarrow_{[p],\theta} (E_1, \lambda\overline{x}.X(\overline{s_m}) \simeq \lambda\overline{x}.t, E_2)\theta|_{W'}$$

where $\theta = \{X \mapsto \lambda\overline{y_m}.y_i(\overline{H_n(\overline{y_m})})\}$

[d] **Decomposition.** If $\simeq \in \{\approx, \rhd\}$ then

$$(E_1,\ \lambda\overline{x}.v(\overline{s_n}) \simeq \lambda\overline{x}.v(\overline{t_n}), E_2)|_W \Rightarrow_{[d],\epsilon} (E_1, \overline{\lambda\overline{x}.s_n \simeq \lambda\overline{x}.t_n}, E_2)\ |_{W'}$$

Lazy narrowing rules

[on] **Outermost narrowing at nonvariable position.**
If $\simeq \in \{\approx, \approx^{-1}, \rhd\}$ and $f(\overline{l_n}) \to r$ is a fresh[1] $\overline{x}$-lifted rewrite rule of $\mathcal{R}$ then

$$(E_1,\ \lambda\overline{x}.f(\overline{s_n}) \simeq \lambda\overline{x}.t, E_2)\ |_W \Rightarrow_{[on],\epsilon} (E_1, \overline{\lambda\overline{x}.s_n \rhd \lambda\overline{x}.l_n}, \lambda\overline{x}.r \simeq \lambda\overline{x}.t, E_2)\ |_{W'}$$

[ov] **Outermost narrowing at variable position.**
If $\simeq \in \{\approx, \approx^{-1}, \rhd\}$, $f(\overline{l_n}) \to r$ is a fresh $\overline{x}$-lifted rewrite rule of $\mathcal{R}$, and either $\lambda\overline{x}.X(\overline{s_m})$ is not a pattern or $X \notin W$ then

$$\begin{aligned}(E_1, \lambda\overline{x}.X(\overline{s_m}) \simeq \lambda\overline{x}.t, E_2)\ |_W \Rightarrow_{[ov],\theta} (&E_1\theta, \overline{\lambda\overline{x}.H_n(\overline{s_m\theta}) \rhd \lambda\overline{x}.l_n},\\ &\lambda\overline{x}.r \simeq \lambda\overline{x}.t\theta, E_2\theta)\ |_{W'}\end{aligned}$$

where $\theta = \{X \mapsto \lambda\overline{y_m}.f(\overline{H_n(\overline{y_m})})\}$.

Rules for removal of flex/flex equations

[t] **Trivial equation.**
If $\simeq \in \{\approx, \rhd\}$ then

$$(E_1, t \simeq t, E_2)|_W \Rightarrow_{[t],\epsilon} (E_1, E_2)|_W$$

[fs] **Flex/flex same.**
If $\simeq \in \{\approx, \rhd\}$ and $X \in W$ then

$$(E_1, \lambda\overline{x}.X(\overline{y_n}) \simeq \lambda\overline{x}.X(\overline{y'_n}), E_2)|_W \Rightarrow_{[fs],\theta} (E_1, E_2)\theta|_{W'}$$

where $\theta = \{X \mapsto \lambda\overline{y_n}.H(\overline{z_p})\}$ with $\{\overline{z_p}\} = \{y_i \mid y_i = y'_i, 1 \le i \le n\}$.

[1] This means that $f(\overline{l_n}) \to r$ is an $\overline{x}$-lifted rewrite rule away from the finite set of free variables which occurred in the preceding part of the computation.

[fd] **Flex/flex different.**
If $X, Y \in W$ then

$$(E_1, \lambda\overline{x}.X(\overline{y_m}) \approx \lambda\overline{x}.Y(\overline{y'_n}), E_2)\lfloor_W \Rightarrow_{[\mathrm{fd}],\theta} (E_1, E_2)\theta\lfloor_{W'}.$$

If $X \in W$ then

$$(E_1, \lambda\overline{x}.X(\overline{y_m}) \rhd \lambda\overline{x}.Y(\overline{y'_n}), E_2)\lfloor_W \Rightarrow_{[\mathrm{fd}],\theta} (E_1, E_2)\theta\lfloor_{W'}.$$

In both situations, $\theta = \{X \mapsto \lambda\overline{y_m}.H(\overline{z_p}), Y \mapsto \lambda\overline{y'_n}.H(\overline{z_p})\}$ with $\{\overline{z_p}\} = \{\overline{y_m}\} \cap \{\overline{y'_n}\}$.

Remark 1. The calculus HOLN restricts the application of inference rule [ov] by taking into account the information that certain variables must be bound to $\mathcal{R}$-normalized values. This information is also used for solving certain flex/flex equations in a deterministic way. Keeping track of the $\mathcal{R}$-normalized variables of a goal and employing this information in the solving process is a novel feature which distinguishes HOLN from the other higher-order lazy narrowing calculi proposed so far in the literature.

3.2 Main Properties of HOLN

It is obvious that $Ans_{\mathcal{R}}^{HOLN}(E\lfloor_W) \subseteq Subst(\mathcal{F},\mathcal{V}) \times Goal_f(\mathcal{F},\mathcal{V})$ whenever $E\lfloor_W \in Goal(\mathcal{F},\mathcal{V})$. In this section we prove that HOLN is a sound and complete calculus. The following trivial lemmata will be instrumental in our proofs.

Lemma 1. *Let $\theta, \gamma_1, \gamma_2 \in Subst(\mathcal{F},\mathcal{V})$ and $V, V' \in \mathcal{P}(\mathcal{V})$ such that $V' \subseteq \mathcal{FV}(V\theta)$. If $\gamma_1 = \gamma_2$ $[V]$ then $\theta\gamma_1 = \theta\gamma_2$ $[V']$.*

Lemma 2. *Let $\theta, \gamma \in Subst(\mathcal{F},\mathcal{V})$ and $V, V' \in \mathcal{V}$ such that $V' \subseteq \mathcal{FV}(V\theta)$. If θ is a pattern substitution and $\theta\gamma\lfloor_V$ is $\mathcal{R}$-normalized, then $\gamma\lfloor_{V'}$ is $\mathcal{R}$-normalized.*

First, we prove that HOLN is sound. The following theorem is instrumental in our proof of soundness.

Theorem 1. *Let $\pi : E\lfloor_W \Rightarrow_{\alpha,e,\theta} E'\lfloor_{W'}$ be an arbitrary HOLN-step. If $\gamma' \in \mathcal{U}_{\mathcal{R}}(E')$ then $\theta\gamma' \in \mathcal{U}_{\mathcal{R}}(E)$.*

Proof. The proof is by case distinction on the label of the inference step.

– If $\alpha = [\mathrm{i}]$ then π is of the form

$$(E_1, \underbrace{\lambda\overline{x}.X(\overline{s_m}) \simeq \lambda\overline{x}.g(\overline{t_n})}_{e}, E_2)\lfloor_W \Rightarrow_{[\mathrm{i}],e,\theta} (E_1, \overline{\lambda\overline{x}.H_n(\overline{s_m}) \simeq \lambda\overline{x}.t_n}, E_2)\theta\lfloor_{W'}$$

where $\theta = \{X \mapsto \lambda\overline{y_m}.g(\overline{H_n(\overline{y_m})})\}$. Obviously, $\theta\gamma' \in \mathcal{U}_{\mathcal{R}}(E_1, E_2)$, and we only have to check that $\theta\gamma' \in \mathcal{U}_{\mathcal{R}}(e)$. From $\gamma' \in \mathcal{U}_{\mathcal{R}}(E')$ we learn that $\gamma' \in \mathcal{U}_{\mathcal{R}}(\overline{\lambda\overline{x}.H_n(\overline{s_m\theta}) \simeq \lambda\overline{x}.t_n})$.
This implies that $\gamma' \in \mathcal{U}_{\mathcal{R}}(\lambda\overline{x}.g(\overline{H_n(\overline{s_m\theta})}) \simeq \lambda\overline{x}.g(\overline{t_n})) = \mathcal{U}_{\mathcal{R}}(e\theta)$, and thus $\theta\gamma' \in \mathcal{U}_{\mathcal{R}}(e)$.

- If $\alpha = [\text{on}]$, there is a fresh $\overline{x}$-lifted rewrite rule $f(\overline{l_n}) \to r$ of $\mathcal{R}$ such that π is of the form

$$(E_1, \underbrace{\lambda\overline{x}.f(\overline{s_n}) \simeq \lambda\overline{x}.t}_{e}, E_2)\downharpoonright_W \Rightarrow_{[\text{on}],e,\epsilon} (E_1, \overline{\lambda\overline{x}.s_n \rhd \lambda\overline{x}.l_n}, \lambda\overline{x}.r \simeq \lambda\overline{x}.t, E_2)\downharpoonright_{W'}$$

and we only have to check that $\gamma' \in \mathcal{U}_\mathcal{R}(e)$. From $\gamma' \in \mathcal{U}_\mathcal{R}(E')$ we learn that

$$\begin{array}{ll} \lambda\overline{x}.s_j\gamma' \to_\mathcal{R}^* \lambda\overline{x}.l_j\gamma' & \text{for } 1 \le j \le n \\ \lambda\overline{x}.r\gamma' \to_\mathcal{R}^* \lambda\overline{x}.t'\gamma' & \text{if } e \text{ is an oriented equation} \\ \lambda\overline{x}.r\gamma' \leftrightarrow_\mathcal{R}^* \lambda\overline{x}.t\gamma' & \text{if } e \text{ is an unoriented equation} \end{array}$$

These relations imply that

$$\begin{array}{l} \lambda\overline{x}.f(\overline{s_n})\gamma' \to_\mathcal{R}^* \lambda\overline{x}.t\gamma' \text{ if } e \text{ is an oriented equation} \\ \lambda\overline{x}.f(\overline{s_n})\gamma' \leftrightarrow_\mathcal{R}^* \lambda\overline{x}.t\gamma' \text{ if } e \text{ is an unoriented equation} \end{array}$$

i.e., γ' is an $\mathcal{R}$-solution of e.
- If $\alpha =[\text{ov}]$ then there is a fresh $\overline{x}$-lifted rewrite rule $f(\overline{l_n}) \to r$ of $\mathcal{R}$ such that

$$\pi : (E_1, \underbrace{\lambda\overline{x}.X(\overline{s_m}) \simeq \lambda\overline{x}.t}_{e}, E_2)\downharpoonright_W \Rightarrow_{[\text{ov}],e,\theta} (E_1\theta, \overline{\lambda\overline{x}.H_n(\overline{s_m\theta}) \rhd \lambda\overline{x}.l_n}, \lambda\overline{x}.r \simeq \lambda\overline{x}.t\theta, E_2\theta)\downharpoonright_{W'}$$

where $\theta = \{X \mapsto \lambda\overline{y_m}.f(\overline{H_n(\overline{y_m})})\}$. In this case we only have to check that $\theta\gamma' \in \mathcal{U}_\mathcal{R}(e)$. From $\gamma' \in \mathcal{U}_\mathcal{R}(E')$ we learn that

$$\begin{array}{ll} \lambda\overline{x}.H_j(\overline{s_m\theta})\gamma' \to_\mathcal{R}^* \lambda\overline{x}.l_j\gamma' & \text{for } 1 \le j \le n \\ \lambda\overline{x}.r\gamma' \to_\mathcal{R}^* \lambda\overline{x}.t\theta\gamma' & \text{if } e \text{ is an oriented equation} \\ \lambda\overline{x}.r\gamma' \leftrightarrow_\mathcal{R}^* \lambda\overline{x}.t\theta\gamma' & \text{if } e \text{ is an unoriented equation} \end{array}$$

As a consequence, we have

$$\lambda\overline{x}.X(\overline{s_m})\theta\gamma' = \lambda\overline{x}.f(\overline{H_n(\overline{s_m\theta})})\gamma' \to_\mathcal{R}^* \lambda\overline{x}.f(\overline{l_n\gamma'}) \to_\mathcal{R} \lambda\overline{x}.r\gamma'$$

and therefore:

$$\begin{array}{l} \lambda\overline{x}.X(\overline{s_m})\theta\gamma' \to_\mathcal{R}^* \lambda\overline{x}.t\theta\gamma' \text{ if } e \text{ is an oriented equation} \\ \lambda\overline{x}.X(\overline{s_m})\theta\gamma' \leftrightarrow_\mathcal{R}^* \lambda\overline{x}.t\theta\gamma' \text{ if } e \text{ is an unoriented equation} \end{array}$$

Hence, $\theta\gamma' \in \mathcal{U}_\mathcal{R}(e)$.
- the cases when $\alpha \in \{[\text{p}],[\text{d}],[\text{t}],[\text{fs}],[\text{fd}]\}$ are straightforward. □

Corollary 1 (Soundness). *HOLN is sound.*

Proof. Let $E\downharpoonright_W$ be an arbitrary goal, $\gamma' \in \mathcal{U}_\mathcal{R}(F)$, and

$$E\downharpoonright_W = E_0\downharpoonright_{W_0} \Rightarrow_{\alpha_1,\theta_1} E_1\downharpoonright_{W_1} \Rightarrow_{\alpha_2,\theta_2} \cdots \Rightarrow_{\alpha_n,\theta_n} E_n\downharpoonright_{W_n} = F\downharpoonright_{W'}$$

an HOLN-refutation, abbreviated $E\downharpoonright_W \Rightarrow_\theta^n F\downharpoonright_{W'}$. We can apply n times Lemma 1 to infer that

$$\gamma' \in \mathcal{U}_{\mathcal{R}}(F) = \mathcal{U}_{\mathcal{R}}(E_n) \Rightarrow \theta_n\gamma' \in \mathcal{U}_{\mathcal{R}}(E_{n-1}),$$

$$\ldots,$$

$$\theta_2 \ldots \theta_n\gamma' \in \mathcal{U}_{\mathcal{R}}(E_2) \Rightarrow \theta_1(\theta_2 \ldots \theta_n\gamma') \in \mathcal{U}_{\mathcal{R}}(E_0) = \mathcal{U}_{\mathcal{R}}(E).$$

Thus, $\theta\gamma' \in \mathcal{U}_{\mathcal{R}}(E)$. □

The following definition will be used in the completeness proof of HOLN and of its further refinements.

Definition 11 (Configuration). *A* configuration *is a tuple* $\langle E\lfloor_W, \gamma, \rho\rangle$ *with* $\gamma \in \mathcal{U}_{\mathcal{R}}(E\lfloor_W)$ *and* $\rho \in \mathit{Proof}_{\mathcal{R}}(E, \gamma)$. *We denote the set of configurations by* Cfg.

Our main idea of proving completeness of a higher-order lazy narrowing calculus $\mathcal{C}$ is to identify a poset $(\mathcal{A}, \succeq)$ with $\mathcal{A} \subseteq \mathit{Cfg}$, $\succ$ a well-founded ordering on $\mathcal{A}$, and a partial function

$$\Phi_{\mathcal{C}} : \mathcal{A} \times \mathcal{P}_{fin}(\mathcal{V}) \times \mathit{Eq}(\mathcal{F}, \mathcal{V}) \to \mathit{step}(\mathcal{C}) \times \mathcal{A}$$

which satisfy the following conditions:

(a) $\forall\gamma \in \mathcal{U}_{\mathcal{R}}(E\lfloor_W), \exists\rho \in \mathit{Proof}_{\mathcal{R}}(E, \gamma).\langle E\lfloor_W, \gamma, \rho\rangle \in \mathcal{A}$,

(b) If $T = \langle E\lfloor_W, \gamma, \rho\rangle \in \mathcal{A}$, $V \in \mathcal{P}_{fin}(\mathcal{V})$ with $\mathcal{FV}(E\lfloor_W) \subseteq V$, and $e \in E$ can be selected in a $\mathcal{C}$-step, then $\Phi_{\mathcal{C}}(T, V, e) = \langle\pi, T'\rangle$ with $\pi : E\lfloor_W \Rightarrow_{e,\theta} E'\lfloor_{W'}$, $T' = \langle E'\lfloor_{W'}, \gamma', \rho'\rangle$, $T \succ T'$, and $\gamma = \theta\gamma'$ $[V]$.

Under these assumptions, proving completeness of $\mathcal{C}$ proceeds as follows. Let $E_0\lfloor_{W_0} \in \mathit{Goal}(\mathcal{F}, \mathcal{V})$ and $\gamma_0 \in \mathcal{U}_{\mathcal{R}}(E_0\lfloor_{W_0})$. By (a), there exists a triple $T_1 = \langle E_0\lfloor_{W_0}\gamma_0, \rho_0\rangle \in \mathcal{A}$. Let $V_1 = \mathcal{FV}(E_0) \cup W_0$.

If $\Phi_{\mathcal{C}}(T_1, V_1, e)$ is undefined for all $e \in E$ then the derivation $E_0\lfloor_{W_0} \Rightarrow^0_{\epsilon} E_0\lfloor_{W_0}$ is a $\mathcal{C}$-refutation, thus $\epsilon \in \mathit{Ans}^{\mathcal{C}}_{\mathcal{R}}(E_0\lfloor_{W_0})$. Since, $\epsilon \leq \gamma$ $[\mathcal{FV}(E_0) \cup W_0]$ for any $\gamma \in \mathcal{U}_{\mathcal{R}}(E_0\lfloor_{W_0})$, we conclude that $\mathcal{C}$ is complete.

Otherwise, let $e_1 \in E_0$ be an equation for which $\Phi_{\mathcal{C}}(T_1, V_1, e_0)$ is defined, and let $\langle\pi_1, T_2\rangle = \Phi_{\mathcal{C}}(T_1, V_1, e)$. We assume that $\pi_1 : E_0\lfloor_{W_0} \Rightarrow_{e_1,\theta_1} E_1\lfloor_{W_1}$, $T_2 = \langle E_1\lfloor_{W_1}, \gamma_1, \rho_1\rangle$, and choose $V_2 = \mathcal{FV}(V_1\theta_1)$.

We can now repeat the above construction by starting from T_2, as we did for T_1. The construction is depicted below.

$$\begin{array}{l}
\gamma_0 \in \mathcal{U}_{\mathcal{R}}(E_0\lfloor_{W_0}) \Rightarrow \exists T_1 = \langle E_0\lfloor_{W_0}, \gamma_0, \rho_0\rangle \in \mathcal{A} \quad \text{(by (a))} \\
\qquad\qquad \text{Choose } V_1 = \mathcal{FV}(E_0) \cup W_0, \\
\qquad\qquad\qquad e_1 \in E_0 \text{ for which } \Phi_C(T_1, V_1, e_1) \text{ is defined.} \\
\qquad\qquad \Downarrow \\
\qquad\qquad \text{Let } \langle \pi_1, T_2\rangle = \Phi_C(T_1, V_1, e_1) \\
\qquad\qquad \text{where } \pi_1 : E_0\lfloor_{W_0} \Rightarrow_{e_1,\theta_1} E_1\lfloor_{W_1}, \\
\qquad\qquad\qquad T_2 = \langle E_1\lfloor_{W_1}, \gamma_1, \rho_1\rangle. \\
\qquad\qquad \text{Choose } V_2 = \mathcal{FV}(V_1\theta_1) \\
\qquad\qquad\qquad e_2 \in E_1 \text{ for which } \Phi_C(T_2, V_2, e_2) \text{ is defined} \\
\qquad\qquad \Downarrow \\
\qquad\qquad \vdots \\
\qquad\qquad \text{Choose } V_N = \mathcal{FV}(V_{N-1}\theta_{N-1}), \\
\qquad\qquad\qquad e_2 \in E_1 \text{ for which } \Phi_C(T_2, V_2, e_2) \text{ is defined.} \\
\qquad\qquad \Downarrow \\
\qquad\qquad \text{Let } \langle \pi_N, T_{N+1}\rangle = \Phi_C(T_N, V_N, e_N) \\
\qquad\qquad \text{where } \pi_N : E_{N-1}\lfloor_{W_{N-1}} \Rightarrow_{e_N,\theta_N} E_N\lfloor_{W_N} \\
\qquad\qquad\qquad T_{N+1} = \langle E_N\lfloor_{W_N}, \gamma_N, \rho_N\rangle
\end{array}$$

Let Π be the C-derivation obtained by concatenating the C-steps $\pi_1, \ldots, \pi_N$ in this order. By property (b), we have $T_1 \succ \ldots \succ T_{N+1}$. Since $\succ$ is well-founded, we will eventually reach a triple $T_{N+1} = \langle E_N\lfloor_{W_N}, \gamma_N, \rho_N\rangle$ with $E_N = F$ consisting of flex equations which can not be selected in a C-step starting with $F\lfloor_{W_N}$. Thus, Π is a C-refutation.

It remains to show that $\gamma_0 = \theta_1 \ldots \theta_N\gamma_N\ [\mathcal{FV}(E_0) \cup W_0]$. First, we prove by induction on k that

$$\forall k \in \{0, \ldots, N-1\}.\gamma_{N-k-1} = \theta_{N-k} \ldots \theta_N\gamma_N\ [V_{N-k}]. \tag{2}$$

The base case for $k = 0$ holds by condition (b) for π_N. If (2) holds for $k < N - 1$ then, since $V_{N-k} = \mathcal{FV}(V_{N-k-1}\theta_{N-k-1})$, we learn by Lemma 1 that

$$\theta_{N-k-1}\gamma_{N-k-1} = \theta_{N-k-1}\theta_{N-k} \ldots \theta_N\gamma_N\ [V_{N-k-1}].$$

By condition (b) for π_{N-k-1}, we know that $\gamma_{N-k-2} = \theta_{N-k-1}\gamma_{N-k-1}\ [V_{N-k-1}]$. Thus, $\gamma_{N-k-2} = \theta_{N-k-1}\theta_{N-k} \ldots \theta_N\gamma_N\ [V_{N-k-1}]$ and this concludes our inductive proof of (2). In particular, for $k = N - 1$, we have that $\gamma_0 = \theta_1 \ldots \theta_N\gamma_N\ [V_1]$. □

Theorem 2 (Completeness). *If $\mathcal{R}$ is confluent then* HOLN*is complete.*

To prove that HOLN is complete, we must identify a poset $(\mathcal{A}, \succeq)$ with $\mathcal{A} \subseteq Cfg$, and a partial function

$$\Phi_{\mathsf{HOLN}} : \mathcal{A} \times \mathcal{P}_{fin}(\mathcal{V}) \times Eq(\mathcal{F}, \mathcal{V}) \rightharpoonup step(C) \times \mathcal{A}$$

which satisfy conditions (a) and (b) of our generic completeness proof of a higher-order lazy narrowing calculus C. First, we introduce some useful notions and prove some auxiliary results.

Given an equation e, we define the *size* of e by $|e| := |s| + |t|$ if $e = s \approx t$ or $e = s \rhd t$. The *size* of $E = \overline{e_N}$ is defined by $|E| := \{|e_i| \mid 1 \leq i \leq N\}_m$, where $\{\}_m$ denotes the multiset constructor. The *length* of a rewrite proof ρ for $\gamma \in \mathcal{U}_\mathcal{R}(E\downarrow_W)$ is defined by $|\rho| := \Sigma_{i=1}^N |\rho(i)|$, where $\rho(i)$ is the number of rewrite steps of $\rho(i)$.

Let $\succeq$ be the lexicographic combination of the orderings $\succeq_A, \succeq_B, \succeq_C$, where:

- $\langle E\downarrow_W, \gamma, \rho\rangle \succeq_A \langle E'\downarrow_{W'}, \gamma', \rho'\rangle$ iff $|\rho| \geq |\rho'|$,
- $\langle E\downarrow_W, \gamma, \rho\rangle \succeq_B \langle E'\downarrow_{W'}, \gamma', \rho'\rangle$ iff $\{|X\gamma| \mid X \in Dom(\gamma)\} \geq_{mul} \{|X'\gamma'| \mid X' \in Dom(\gamma')\}$,
- $\langle E\downarrow_W, \gamma, \rho\rangle \succeq_C \langle E'\downarrow_{W'}, \gamma', \rho'\rangle$ iff $|E\gamma| \geq_{mul} |E'\gamma'|$.

Let $=_{A,B,C} := \succeq \cap \succeq^{-1}$. Then $\succ$ is obviously well-founded. Since $\mathcal{R}$ is confluent, condition (a) is obviously satisfied. It remains to show how Φ_{HOLN} can be defined in a way which satisfies condition (b).

The following six lemmata are crucial to justify the correctness of our definition of Φ_{HOLN}. If not stated otherwise, we assume that $\simeq \in \{\approx, \approx^{-1}, \rhd, \rhd^{-1}\}$.

Lemma 3. *Let $E = \overline{e_N}$, $T = \langle E\downarrow_W, \gamma, \rho\rangle \in Cfg$ with $e_k = \lambda\overline{x}.v(\overline{s_n}) \simeq \lambda\overline{x}.v(\overline{t_n}) \in E$, and $V \in \mathcal{P}_{fin}(\mathcal{V})$. Assume $\rho(k)$ has no rewrite steps at the head positions of the equational sides. We define*

$$E' = (\overline{e_{k-1}}, \overline{\lambda\overline{x}.s_n \simeq \lambda\overline{x}.t_n}, e_{k+1}, \ldots, e_N) \quad \textit{and} \quad \pi : E\downarrow_W \Rightarrow_{[\mathsf{d}], e_k, \epsilon} E'\downarrow_W.$$

Then π is a valid HOLN-step and there exists $\rho' \in Proof_\mathcal{R}(E', \gamma')$ such that $T' = \langle E'\downarrow_W, \gamma, \rho'\rangle \in Cfg$ and $T \succ T'$. We denote the pair $\langle \pi, T'\rangle$ by $\Phi_{[\mathsf{d}]}(T, V, e_k)$.

Proof. Since $\rho(k)$ has no rewrite steps at the head positions of the equational sides, we have:

$$\rho(k) : \lambda\overline{x}.v(\overline{s_n\gamma}) \simeq \lambda\overline{x}.v(\overline{t_n\gamma}) \rightarrow^*_\mathcal{R} \lambda\overline{x}.v(\overline{u_n}) \simeq \lambda\overline{x}.v(\overline{u_n})$$

and we can rearrange the rewrite steps of $\rho(k)$ into a sequence of rewrite derivations $R_1, R_2, \ldots, R_n$ where each R_j $(1 \leq j \leq n)$ is of the form

$$\begin{aligned} R_j : \lambda\overline{x}.v(\overline{u_{j-1}}, s_j\gamma, \ldots, s_n\gamma) &\simeq \lambda\overline{x}.v(\overline{u_{j-1}}, t_j\gamma, \ldots, t_n\gamma) \\ \rightarrow^*_\mathcal{R} \lambda\overline{x}.v(\overline{u_{j-1}}, u_j, s_{j+1}\gamma, \ldots, s_n\gamma) &\simeq \lambda\overline{x}.v(\overline{u_{j-1}}, u_j, t_{j+1}\gamma, \ldots, t_n\gamma) \end{aligned}$$

by rewriting only the j-th subterms of the sides of the equation. Then $|\rho(k)| = \Sigma_{j=1}^n |R_j|$ and we can extract from R_j a corresponding rewrite derivation

$$R'_j : \lambda\overline{x}.s_j\gamma \simeq \lambda\overline{x}.t_j\gamma \rightarrow^*_\mathcal{R} \lambda\overline{x}.u_j \simeq \lambda\overline{x}.u_j$$

with $|R'_j| = |R_j|$. We define $\rho'(i)$ for $i \in \{1, \ldots, N+n-1\}$ by

$$\rho'(i) = \begin{cases} \rho(i) & \text{if } i < k, \\ R'_{i-k+1} & \text{if } k \leq i < k+n, \\ \rho(i-n+1) & \text{if } k+n \leq i \leq N+n-1. \end{cases}$$

It is easy to see that $\rho' \in \mathit{Proof}_{\mathcal{R}}(E', \gamma)$, $T' = \langle E'\lfloor_W, \gamma, \rho'\rangle \in \mathit{Cfg}$ and $T =_A T'$, $T =_B T'$, $T \succ_C T'$. Thus $T \succ T'$. $\square$

Lemma 4. *Let $E = \overline{e_N}$, $T = \langle E\lfloor_W, \gamma, \rho\rangle \in \mathit{Cfg}$, $e_k = \lambda\overline{x}.X(\overline{s}) \simeq \lambda\overline{x}.X(\overline{s}) \in E$, and $V \in \mathcal{P}_{fin}(\mathcal{V})$. We define:*

$E' = (\overline{e_{k-1}}, e_{k+1}, \dots, e_N)$,

$\rho' : \{1, \dots, N-1\} \to \bigcup_{i \in \{1,\dots,N\}-\{k\}} \mathit{Proof}_{\mathcal{R}}(e_i, \gamma)$, $\rho'(i) = \begin{cases} \rho(i) & \text{if } 1 \le i < k, \\ \rho(i+1) & \text{if } k \le i < N \end{cases}$

$T' = \langle E'\lfloor_W, \gamma, \rho'\rangle$, $\pi : E\lfloor_W \Rightarrow_{[t],e_k,\epsilon} E'\lfloor_W$.

Then π is a valid HOLN-step, $\rho' \in \mathit{Proof}_{\mathcal{R}}(E', \gamma)$, $T' = \langle E'\lfloor_W, \gamma, \rho'\rangle \in \mathit{Cfg}$, and $T \succ T'$. We denote the pair $\langle \pi, T'\rangle$ by $\Phi_{[t]}(T, V, e_k)$.

Proof. Straightforward. $\square$

Lemma 5. *Let $E = \overline{e_N}$, $T = \langle E\lfloor_W, \gamma, \rho\rangle \in \mathit{Cfg}$, $e_k = \lambda\overline{x}.X(\overline{s_m}) \simeq \lambda\overline{x}.t \in E$ with $\lambda\overline{x}.t$ rigid, and $V \in \mathcal{P}_{fin}(\mathcal{V})$ such that $\mathcal{FV}(E\lfloor_W) \subseteq V$.*

(i) Assume $\mathrm{head}(X\gamma) \in \mathcal{F}$ *and $\rho(k)$ has no rewrite steps at the head positions of the equational sides. Let π be the HOLN-step $\pi : E\lfloor_W \Rightarrow_{[i],e_k,\theta} E'\lfloor_{W'}$. There exists $\rho' \in \mathit{Proof}_{\mathcal{R}}(E'\lfloor_{W'})$ such that $T' = \langle E'\lfloor_{W'}, \gamma', \rho'\rangle \in \mathit{Cfg}$, $T \succ T'$ and $\gamma = \theta\gamma'\ [V]$. We denote the pair $\langle \pi, T'\rangle$ by $\Phi_{[i]}(T, V, e_k)$.*

(ii) Assume $X\gamma = \lambda\overline{y_m}.y_i(\overline{u_p})$. Then there exist
- *a HOLN-step $\pi : E\lfloor_W \Rightarrow_{[p],e_k,\theta} E'\lfloor_{W'}$ and*
- *$T' = \langle E'\lfloor_{W'}, \gamma', \rho'\rangle \in \mathit{Cfg}$ such that $T \succ T'$ and $\gamma = \theta\gamma'\ [V]$.*

We denote the pair $\langle \pi, T'\rangle$ by $\Phi_{[p]}(T, V, e_k)$.

Proof. First, we prove Lemma 5.(i). Assume $\mathrm{head}(X\gamma) = f \in \mathcal{F}$ and $\rho(k)$ has no rewrite steps at the head positions of the equational sides. In this case $\lambda\overline{x}.t$ must be of the form $\lambda\overline{x}.f(\overline{t_n})$ and we can write $\gamma = \theta\gamma'\ [\mathit{Dom}(\gamma) \setminus \{\overline{H_n}\}]$, where $\theta = \{X \mapsto \lambda\overline{y_m}.f(\overline{H_n(\overline{y_m})})\}$, $\mathit{Dom}(\gamma') = (\mathit{Dom}(\gamma) \setminus \{X\}) \cup \{\overline{H_n}\}$, and $\overline{H_n}$ is a sequence of distinct fresh variables. Therefore, $\{\overline{H_n}\} \cap V = \emptyset$, and thus $\gamma = \theta\gamma'\ [V]$. In this case, $\rho(k)$ is of the form

$$\lambda\overline{x}.f(\overline{H_n(\overline{s_m})\gamma'}) \simeq \lambda\overline{x}.f(\overline{t_n\theta\gamma'}) \to_{\mathcal{R}}^{*} \lambda\overline{x}.f(\overline{u_n}) \simeq \lambda\overline{x}.f(\overline{u_n})$$

with no rewrite steps at the head positions of the equational sides. Let $E'' = E\theta$. Then $\gamma' \in \mathcal{U}_{\mathcal{R}}(E'')$ and $\rho \in \mathit{Proof}_{\mathcal{R}}(E'', \gamma')$. By Lemma 2 we learn that $\gamma' \in \mathcal{U}_{\mathcal{R}}(E''\lfloor_{W'})$. Note that $T'' = \langle E''\lfloor_{W'}, \gamma', \rho\rangle \in \mathit{Cfg}$ and $T \succ T''$. We can apply Lemma 3 to construct $\rho' \in \mathit{Proof}_{\mathcal{R}}(E', \gamma')$ such that $T' = \langle E'\lfloor_{W'}, \gamma', \rho'\rangle \in \mathit{Cfg}$ and $T'' \succ T'$. We conclude $T \succ T'$ from the transitivity of $\succ$.

Next, we prove Lemma 5.(ii). Assume $X\gamma = \lambda\overline{y_m}.y_i(\overline{u_p})$. Then we can write $\gamma = \theta\gamma'\ [\mathit{Dom}(\gamma) \setminus \{\overline{H_p}\}]$, where $\theta = \{X \mapsto \lambda\overline{y_m}.y_i(\overline{H_p(\overline{y_m})})\}$, $\mathit{Dom}(\gamma') = (\mathit{Dom}(\gamma) \setminus \{X\}) \cup \{\overline{H_p}\}$, and $\overline{H_p}$ is a sequence of distinct fresh variables. Therefore, $\{\overline{H_p}\} \cap V = \emptyset$, and thus $\gamma = \theta\gamma'\ [V]$. Let

$$\pi : E\lfloor_W \Rightarrow_{[p],e_k,\theta} E'\lfloor_{W'}$$

Obviously, $\gamma' \in \mathcal{U}_{\mathcal{R}}(E')$ and $\rho \in \mathit{Proof}_{\mathcal{R}}(E', \gamma')$. By Lemma 2, we learn that $\gamma' \in \mathcal{U}_{\mathcal{R}}(E'\lfloor_{W'})$. We define $T' := \langle E'\lfloor_{W'}, \gamma', \rho\rangle$. Then $T' \in \mathit{Cfg}$ and $T =_A T'$, $T \succ_B T'$. Thus, $T \succ T'$. This concludes our proof of Lemma 5.(ii). $\square$

Lemma 6. *Let $E = \overline{e_N}$, $T = \langle E|_W, \gamma, \rho\rangle \in Cfg$, $e_k \in E$, and $V \in \mathcal{P}_{fin}(\mathcal{V})$ such that $\mathcal{FV}(E|_W) \subseteq V$.*

(i) If $e_k = \lambda\overline{x}.s \simeq \lambda\overline{x}.t$ with $\mathrm{head}((\lambda\overline{x}.s)\gamma) = f \in \mathcal{F}$, $\simeq \in \{\approx, \rhd\}$, and $\rho(k)$ has a rewrite step at the head position of the left-hand side, then there exist

- *an $\overline{x}$-lifter $f(\overline{l_n}) \to r$ of a rewrite rule of $\mathcal{R}$ and*
- *$T' = \langle \underbrace{(\overline{e_{k-1}}, \lambda\overline{x}.s \rhd \lambda\overline{x}.f(\overline{l_n}), \lambda\overline{x}.r \simeq \lambda\overline{x}.t, e_{k+1}, \ldots, e_N)}_{E'}|_W, \gamma', \rho'\rangle \in Cfg$*

such that $T \succ T'$, $\gamma = \gamma'$ $[V]$, and $\rho'(k)$ has no rewrite steps at the head position of the left-hand side.
In this case we define $\Phi_{[o]}(T, V, e_k) := \langle \pi, T'\rangle$ where $\pi : E|_W \Rightarrow E'|_W$.

(ii) If $e_k = \lambda\overline{x}.s \approx \lambda\overline{x}.t$, $\mathrm{head}((\lambda\overline{x}.t)\gamma) \in \mathcal{F}$, and $\rho(k)$ has a rewrite step at the head position of the right-hand side, then there exist

- *an $\overline{x}$-lifter $f(\overline{l_n}) \to r$ of a rewrite rule of $\mathcal{R}$ and*
- *$T' = \langle \underbrace{(\overline{e_{k-1}}, \lambda\overline{x}.s \rhd \lambda\overline{x}.f(\overline{l_n}), \lambda\overline{x}.s \simeq \lambda\overline{x}.r, e_{k+1}, \ldots, e_N)}_{E'}|_W, \gamma', \rho'\rangle \in Cfg$*

such that $T \succ T'$, $\gamma = \gamma'$ $[V]$, and $\rho'(k)$ has no rewrite steps at the head position of the left-hand side.
In this case we define $\Phi_{[o]}(T, V, e_k) := \langle \pi, T'\rangle$ where $\pi : E|_W \Rightarrow E'|_W$.

Proof. We prove only Lemma 6.(i) because Lemma 6.(ii) has a similar proof. Under the given assumptions, we can assume $\rho(k)$ is of the form

$$e_k\gamma \to_{\mathcal{R}}^{*} \lambda\overline{x}.f(\overline{s'_n}) \simeq \lambda\overline{x}.t' \xrightarrow[p,\delta]{f(\overline{l_n})\to r} \lambda\overline{x}.r\delta \simeq \lambda\overline{x}.t' \to_{\mathcal{R}}^{*} \lambda\overline{x}.u \simeq \lambda\overline{x}.u$$

where p is the head position of the left-hand side, $f(\overline{l_n}) \to r$ is a fresh $\overline{x}$-lifter of a rewrite rule in $\mathcal{R}$ such that $\lambda\overline{x}.f(\overline{l_n})\delta = \lambda\overline{x}.f(\overline{s'_n})$, and the depicted rewrite step is the first one at position p. Then $(\lambda\overline{x}.s)\gamma = \lambda\overline{x}.f(\overline{s_n})$, and we can decompose $\rho(k)$ into two rewrite derivations:

$$R_1 : e\gamma = \lambda\overline{x}.f(\overline{s_n}) \simeq \lambda\overline{x}.t\gamma \to_{\mathcal{R}}^{*} \lambda\overline{x}.f(\overline{l_n})\delta \simeq \lambda\overline{x}.t'$$
$$R_2 : \lambda\overline{x}.f(\overline{l_n})\delta \simeq \lambda\overline{x}.t' \xrightarrow[p,\delta]{f(\overline{l_n})\to r} \lambda\overline{x}.r\delta \simeq \lambda\overline{x}.t' \to_{\mathcal{R}}^{*} \lambda\overline{x}.u \simeq \lambda\overline{x}.u$$

Let R'_1 be the rewrite derivation obtained from R_1 by rearranging the rewrite steps such that we first rewrite the left-hand sides and next the right-hand sides. This means that R'_1 can be decomposed into 2 rewrite derivations:

$$R''_1 : e\gamma = \lambda\overline{x}.f(\overline{s_n}) \simeq \lambda\overline{x}.t\gamma \to_{\mathcal{R}}^{*} \lambda\overline{x}.f(\overline{l_n})\delta \simeq \lambda\overline{x}.t\gamma$$
$$R''_2 : \lambda\overline{x}.f(\overline{l_n})\delta \simeq \lambda\overline{x}.t\gamma \xrightarrow[p,\delta]{f(\overline{l_n})\to r} \lambda\overline{x}.r\delta \simeq \lambda\overline{x}.t\gamma \to_{\mathcal{R}}^{*} \lambda\overline{x}.r\delta \simeq \lambda\overline{x}.t'$$

such that R''_1 has no rewrite steps at the head position of the left-hand side, and $|R''_1| + |R''_2| = |R'_1| = |R_1|$.

$f(l_n) \to r$ is a fresh $\overline{x}$-lifter, and thus $\mathcal{FV}(\lambda\overline{x}.f(\overline{l_n})) \cap Dom(\gamma) = \emptyset$. Since $Dom(\delta) \subseteq \mathcal{FV}(\lambda\overline{x}.f(\overline{l_n}))$, we conclude that $\underline{\gamma'} := \gamma \cup \delta$ is a well-defined substitution and $\gamma' \in \mathcal{U}_{\mathcal{R}}(E'|_W)$. Note that $E' = \overline{e'_{N+1}}$ with

- $e'_i = e_i$ and $\rho(i) \in Proof_{\mathcal{R}}(e'_i, \gamma')$, if $i < k$,

- $e'_k = \lambda\overline{x}.s \rhd \lambda\overline{x}.f(\overline{l_n})$ and $R''_1 \in \mathit{Proof}_{\mathcal{R}}(e'_k, \gamma')$,
- $e'_{k+1} = \lambda\overline{x}.r \simeq \lambda\overline{x}.t$ and $(R''_2, R_2) \in \mathit{Proof}_{\mathcal{R}}(e'_{k+1}, \gamma')$,
- $e'_i = e_{i-1}$ and $\rho(i-1) \in \mathit{Proof}_{\mathcal{R}}(e'_i, \gamma')$ if $i > k+1$.

Therefore, we can define $\rho' \in \mathit{Proof}_{\mathcal{R}}(E', \gamma')$ by $\rho'(i) = \begin{cases} \rho(i) & \text{if } i < k, \\ R''_1 & \text{if } i = k, \\ (R''_2, R_2) & \text{if } i = k+1, \\ \rho(i-1) & \text{if } i > k+1. \end{cases}$

Then obviously $T' \in \mathit{Cfg}$ and $T \succ_A T'$. Thus $T \succ T'$. □

Lemma 7. *Let $E = \overline{e_N}$, $T = \langle E\lfloor_W, \gamma, \rho\rangle \in \mathit{Cfg}$, $e_k = \lambda\overline{x}.X(\overline{y_n}) \simeq \lambda\overline{x}.X(\overline{y'_n}) \in E$ with $X \in W$, and $V \in \mathcal{P}_{fin}(\mathcal{V})$ such that $\mathcal{FV}(E\lfloor_W) \subseteq V$.*

We define the HOLN-step $\pi : E\lfloor_W \Rightarrow_{[\mathsf{fs}], e_k, \theta} E'\lfloor_{W'}$.

Then there exists $T' = \langle E'\lfloor_{W'}, \gamma', \rho'\rangle \in \mathit{Cfg}$ such that $T \succ T'$ and $\gamma = \theta\gamma'$ $[V]$. We denote the pair $\langle \pi, T'\rangle$ by $\Phi_{[\mathsf{fs}]}(T, V, e_k)$.

Proof. From $\gamma \in \mathcal{U}_{\mathcal{R}}(E\lfloor_W)$ and $X \in W$ results that $X\gamma$ is an $\mathcal{R}$-normal form. This implies that $\lambda\overline{x}.X(\overline{y_n})\gamma$ and $\lambda\overline{x}.X(\overline{y'_n})\gamma$ are $\mathcal{R}$-normal forms too, and thus $\rho(k)$ has no rewrite steps. Hence, γ is a unifier $\lambda\overline{x}.X(\overline{y})$ and $\lambda\overline{x}.X(\overline{y'_n})$. On the other hand, it is well-known that the substitution θ computed by π is a mgu [14] of $\lambda\overline{x}.X(\overline{y_n})$ and $\lambda\overline{x}.X(\overline{y'_n})$ over $\{X\}$. From the freshness condition on the variables introduced by θ, we conclude that $V \cap \mathit{Ran}(\theta) = \emptyset$, and thus $\theta \leq \gamma$ $[V]$. Therefore there exists γ' such that $\gamma = \theta\gamma'$ $[V]$. Together with Lemma 2, this implies that $\gamma' \in \mathcal{U}_{\mathcal{R}}(E'\lfloor_{W'})$ and that the map ρ' defined by

$$\rho'(i) = \begin{cases} \rho(i) & \text{if } 1 \leq i < k, \\ \rho(i+1) & \text{if } k \leq i < N \end{cases}$$

is a rewrite proof of $\gamma' \in \mathcal{U}_{\mathcal{R}}(E'\lfloor_{W'})$. This implies that $T' = \langle E'\lfloor_{W'}, \gamma', \rho'\rangle \in \mathit{Cfg}$. Since $T =_A T'$, $T \succeq_B T'$ and $T \succ_C T'$, we conclude that $T \succ T'$. □

Lemma 8. *Let $E = \overline{e_N}$, $T = \langle E\lfloor_W, \gamma, \rho\rangle \in \mathit{Cfg}$, $e_k \in E$, and $V \in \mathcal{P}_{fin}(\mathcal{V})$ such that $\mathcal{FV}(E\lfloor_W) \subseteq V$. Assume $\begin{cases} e_k = \lambda\overline{x}.X(\overline{y_m}) \approx \lambda\overline{x}.Y(\overline{y'_n}) \text{ and } X, Y \in W, \\ \text{or} \\ e_k = \lambda\overline{x}.X(\overline{y_m}) \rhd \lambda\overline{x}.Y(\overline{y'_n}) \text{ and } X \in W. \end{cases}$ and let π be the HOLN-step $\pi : E\lfloor_W \Rightarrow_{[\mathsf{fd}], e_k, \theta} E'\lfloor_{W'}$.*

Then there exists $T' = \langle E'\lfloor_{W'}, \gamma', \rho'\rangle \in \mathit{Cfg}$ such that $T \succ T'$ and $\gamma = \theta\gamma'$ $[V]$. We denote the pair $\langle \pi, T'\rangle$ by $\Phi_{[\mathsf{fd}]}(T, V, e_k)$.

Proof. From $\gamma \in \mathcal{U}_{\mathcal{R}}(E\lfloor_W)$ and $X \in W$ results that $X\gamma$ is an $\mathcal{R}$-normal form. This implies that $\lambda\overline{x}.X(\overline{y_m})\gamma$ is an $\mathcal{R}$-normal form. Also, if e_k is an unoriented equation, we learn from $Y \in W$ and $\gamma \in \mathcal{U}_{\mathcal{R}}(E\lfloor_W)$ that $Y\gamma$ is an $\mathcal{R}$-normal form, and thus $\lambda\overline{x}.Y(\overline{y'_n})\gamma$ is an $\mathcal{R}$-normal form too.

These observations imply that $\rho(k)$ has no rewrite steps, redardless whether e_k is an oriented equation or not. This implies that γ is a unifier of $\lambda\overline{x}.X(\overline{y_m})$ and $\lambda\overline{x}.X(\overline{y'_n})$. On the other hand, it is well-known that the substitution θ computed by π is a mgu [14] of $\lambda\overline{x}.X(\overline{y_m})$ and $\lambda\overline{x}.Y(\overline{y'_n})$ over $\{X, Y\}$. From the freshness condition on the variables introduced by θ, we conclude that $V \cap \mathit{Ran}(\theta) = \emptyset$, and thus $\theta \leq \gamma\ [V]$. Therefore there exists γ' such that $\gamma = \theta\gamma'\ [V]$. Together with Lemma 2, this implies that $\gamma' \in \mathcal{U}_{\mathcal{R}}(E'\lfloor_{W'})$ and the map ρ' defined by

$$\rho'(i) = \begin{cases} \rho(i) & \text{if } 1 \leq i < k, \\ \rho(i+1) & \text{if } k \leq i < N \end{cases}$$

is a rewrite proof of $\gamma' \in \mathcal{U}_{\mathcal{R}}(E'\lfloor_{W'})$. Thus $T' = \langle E'\lfloor_{W'}, \gamma', \rho'\rangle \in \mathit{Cfg}$. Since $T =_A T'$ and $T \succ_B T'$, we conclude that $T \succ T'$. □

We are ready now to define Φ_{HOLN}. Let $E = \overline{e_N}$, $E\lfloor_W \in \mathit{Goal}(\mathcal{F}, \mathcal{V})$, $V \in \mathcal{P}_{fin}(\mathcal{V})$ with $\mathcal{FV}(E\lfloor_W) \subseteq V$, and $e_k \in E$ an equation which can be selected in an HOLN-step starting with $E\lfloor_W$.

We choose $(\mathcal{A}, \succeq) = (\mathit{Cfg}, \succeq)$ where $\succeq$ is the lexicographic combination of $\succeq_A, \succeq_B, \succeq_C$, and distinguish the following cases:

- $e_k = \lambda\overline{x}.X(\overline{s_n}) \simeq \lambda\overline{x}.X(\overline{s_n})$. Then we define $\Phi_{\text{HOLN}}(T, V, e_k) := \Phi_{[t]}(T, V, e_k)$. In this case, Φ_{HOLN} satisfies condition (b) of our generic completeness proof, because of Lemma 4.
- Otherwise, assume $e_k = \lambda\overline{x}.v(\overline{s_n}) \simeq \lambda\overline{x}.v(\overline{t_n})$ and $\rho(k)$ has no rewrite steps at the head positions of the equational sides. Then we define $\Phi_{\text{HOLN}}(T, V, e_k) := \Phi_{[d]}(T, V, e_k)$. In this case, Φ_{HOLN} satisfies condition (b) of our generic completeness proof, because of Lemma 3.
- Otherwise, assume $e_k = \lambda\overline{x}.X(\overline{s_m}) \simeq \lambda\overline{x}.t$ with $\text{head}(X\gamma) \in \mathcal{F}$, $\lambda\overline{x}.t$ rigid and $\rho(k)$ has no rewrite steps at the head positions of the equational sides. Then we define $\Phi_{\text{HOLN}}(T, V, e_k) := \Phi_{[i]}(T, V, e_k)$. In this case, Φ_{HOLN} satisfies condition (b) of our generic completeness proof, because of Lemma 5.(i).
- Otherwise, assume $e_k = \lambda\overline{x}.X(\overline{s_m}) \simeq \lambda\overline{x}.t$ with $X\gamma = \lambda\overline{y_m}.y_i(\overline{u_p})$ and $\lambda\overline{x}.t$ rigid. Then we define $\Phi_{\text{HOLN}}(T, V, e_k) := \Phi_{[p]}(T, V, e_k)$. In this case, Φ_{HOLN} satisfies condition (b) of our generic proof because of Lemma 5.(ii).
- Otherwise, assume $e_k = \lambda\overline{x}.s \simeq \lambda\overline{x}.t$ with $\text{head}((\lambda\overline{x}.s)\gamma) \in \mathcal{F}$, $\simeq \in \{\approx, \rhd\}$, and $\rho(k)$ has a rewrite step at the head position of the left-hand side. Then there exists $\langle\pi_1, T''\rangle = \Phi_{[o]}(T, V, e_k)$ where T'' is a configuration of the form

  $$T'' = \langle(\underbrace{\overline{e_{k-1}}, \lambda\overline{x}.s \rhd \lambda\overline{x}.f(\overline{l_n}), \lambda\overline{x}.r \simeq \lambda\overline{x}.t, e_{k+1}, \ldots, e_N}_{E''})\lfloor_W, \gamma'', \rho''\rangle$$

  and $\rho''(k)$ has no rewrite steps at the head position of the left-hand side. Let $E'' = \overline{e''_{N+1}}$. Then $e''_k = \lambda\overline{x}.s \rhd \lambda\overline{x}.f(\overline{l_n})$ and we can determine $\langle\pi_2, T'\rangle = \Phi_{\text{HOLN}}(T'', V, e''_k)$ by appeal to the previous cases. Note that $\gamma = \gamma''\ [V]$ and $T \succ T'$ because $T \succ T''$ by Lemma 6, and $T'' \succ T'$ by property (b) of Φ_{HOLN} defined so far. We can write

  $$\pi_2 : E''\lfloor_W \Rightarrow_{\alpha, e''_k, \theta} E'\lfloor_{W'}$$

  and $T' = \langle E'\lfloor_{W'}, \gamma', \rho'\rangle$. It is easy to check that $\alpha \in \{[i], [d]\}$. We have $\gamma'' = \theta\gamma'\ [V]$ by property (b) of Φ_{HOLN} defined so far. Thus, $\gamma = \theta\gamma'\ [V]$. Also, note that the relation

  $$\pi : E\lfloor_W \Rightarrow_{\alpha, e_k, \theta} E'\lfloor_{W'}$$

is an [ov]-step if α = [i] and an [on]-step if α = [d]. Hence, in this case we can define $\Phi_{\mathrm{HOLN}}(T, V, e_k) := \langle \pi, T' \rangle$.

- Otherwise, assume $e_k = \lambda\overline{x}.X(\overline{y_n}) \simeq \lambda\overline{x}.X(\overline{y'_n})$ with $X \in W$. Then we define $\Phi_{\mathrm{HOLN}}(T, V, e_k) := \Phi_{[\mathrm{fs}]}\ (T, V, e_k)$. In this case, Φ_{HOLN} satisfies condition (b) of our generic proof, because of Lemma 7.
- Otherwise, assume $\begin{cases} e_k = \lambda\overline{x}.X(\overline{y_m}) \approx \lambda\overline{x}.Y(\overline{y'_n}) \text{ and } X, Y \in W \\ \text{or} \\ e_k = \lambda\overline{x}.X(\overline{y_m}) \rhd \lambda\overline{x}.Y(\overline{y'_n}) \text{ and } X \in W. \end{cases}$

  Then we define $\Phi_{\mathrm{HOLN}}(T, V, e_k) := \Phi_{[\mathrm{fd}]}\ (T, V, e_k)$. In this case, Φ_{HOLN} satisfies condition (b) of our generic completeness proof, because of Lemma 8.

Since these are all the possibilities which can be satisfied by a selected equation e_k in an HOLN-step, we conclude that our definition of Φ_{HOLN} satisfies condition (b) of our generic proof. □

Remark 2. We have actually proved a stronger result: if $\mathcal{R}$ is a confluent PRS then HOLN is *strongly complete*, i.e., completeness is independent of the choice of the equation in the current goal.

4 Refinements

There are two sources of don't know nondeterminism in computations with HOLN-derivations: the choice of the inference rule of HOLN, and the choice of the rewrite rule of $\mathcal{R}$ when narrowing steps are performed. In the sequel we will investigate the possibility to make the computation with HOLN-derivations more deterministic by reducing the choices of inference rules.

4.1 HOLN$_1$: a Refinement of HOLN for Left-Linear EPRSs

Programs restricted to left-linear (term or pattern) rewrite systems are widely accepted in functional logic programming. As we will see later, the notion of left-linear EPRS (LEPRS for short) extends the notion of left-linear term rewriting system to the higher-order case in a way which preserves many properties of their first-order counterpart which are relevant to our investigation.

In this subsection we study the behavior of HOLN when $\mathcal{R}$ is a LEPRS.

First, we confine our attention to a particular class of oriented equations produced upon outermost narrowing steps, the class of parameter-passing descendants.

Definition 12. *A* parameter-passing equation *of a goal* $E'|_{W'}$ *in an* HOLN-*derivation* $\Pi : E|_W \Rightarrow^*_{\theta} E'|_{W'}$ *is either*

(a) an equation $\lambda\overline{x}.s_i \rhd \lambda\overline{x}.l_i$ $(1 \le i \le n)$ *if the last step of* Π *is of the form:*

$$(E_1, \lambda\overline{x}.f(\overline{s_n}) \simeq \lambda\overline{x}.t, E_2)|_W \Rightarrow_{[\mathrm{on}],\epsilon} (E_1, \overline{\lambda\overline{x}.s_n \rhd \lambda\overline{x}.l_n}, \lambda\overline{x}.r \simeq \lambda\overline{x}.t, E_2)|_{W'}$$

(b) an equation $\lambda\overline{x}.H_i(\overline{s_m\theta}) \rhd \lambda\overline{x}.l_i$ *($1 \le i \le n$) if the last step of* Π *is of the form:*

$$(E_1, \lambda\overline{x}.X(\overline{s_m}) \simeq \lambda\overline{x}.t, E_2)\lfloor_W \Rightarrow_{[\mathrm{ov}],\theta} (E_1\theta, \overline{\lambda\overline{x}.H_n(\overline{s_m\theta}) \rhd \lambda\overline{x}.l_n}, \\ \lambda\overline{x}.r \simeq \lambda\overline{x}.t\theta, E_2\theta)\lfloor_{W'}.$$

A parameter-passing descendant *of a goal* $E'\lfloor_{W'}$ *in an* HOLN-*derivation* $\Pi : E\lfloor_W \Rightarrow^*_\theta E'\lfloor_{W'}$ *is either a parameter-passing equation in* Π *or a descendant of a parameter-passing equation in* Π.

Note that parameter-passing descendants are always oriented equations. To distinguish them, we will write $s \blacktriangleright t$ instead of $s \rhd t$.

The following lemma characterizes the HOLN-derivations when $\mathcal{R}$ is a (fully-extended) left-linear PRS, and can be proved by induction on the length of the HOLN-derivation.

Lemma 9. *Let* $\mathcal{R}$ *be a left-linear PRS and*

$$\Pi : E\lfloor_W \Rightarrow^*_\theta (E_1, s \blacktriangleright t, E_2)\lfloor_{W'}$$

an HOLN-derivation such that $s \blacktriangleright t$ *is a parameter-passing descendant of* $(E_1, s \blacktriangleright t, E_2)\lfloor_{W'}$ *in* Π. *Then*

(i) t *is a linear pattern,*
(ii) $(\mathcal{FV}(E_1, s) \cup \mathcal{FV}(E\theta) \cup W') \cap \mathcal{FV}(t) = \emptyset$,
(iii) if $\mathcal{R}$ *is an LEPRS then* t *is a fully-extended pattern.*

An important theoretical result which is relevant to our investigation is the validity of the standardization theorem for confluent LEPRS [17]. In the sequel we give a brief account to this theoretical result.

Definition 13. *A position* p *is a* pattern position *of a term* t *if* $p \in Pos(t)$ *and* $\mathrm{head}(t|_q) \notin \mathcal{FV}(t)$ *for all* $q \le p$. *We denote by* $Pat(t)$ *the set of pattern positions of a term* t.

For example, if $t = f(a, X(\lambda x.x))$ then ϵ and 1 and 2 are pattern positions of t, but 2 and 2·1 are not.

Definition 14. *Let* $E = \overline{e_N}$ *and* $\gamma \in \mathcal{U}_\mathcal{R}(E\lfloor_W)$. *A rewrite proof* $\rho \in Proof_\mathcal{R}(E, \gamma)$ *is* outside-in *if for any* $k \in \{1, \dots, N\}$ *we have:*

(i) $\rho(k)$ *is an* outside-in *rewrite derivation, that is, if* $\rho(k)$ *is of the form*

$$e_k\gamma \to^{l_1 \to r_1}_{p_1} \dots \to^{l_n \to r_n}_{p_n} u \simeq u$$

then the following condition is satisfied for all $1 \le i \le n-1$: *if there exists* j *with* $i < j$ *such that* $p_i = p_j + q$ *then* $q \in Pat(l_j)$ *for the least such* j,
(ii) If $e_k = s \blacktriangleright t \in E$ *and* $\rho(k)$ *has a rewrite step at position* $1{\cdot}p$ *such that no later rewrite steps take place above position* $1{\cdot}p$ *then* $p \in Pat(t)$.

A configuration $\langle E|_W, \gamma, \rho\rangle$ *is* outside-in *if* ρ *is an outside-in rewrite proof. We denote by* Cfg^{oi} *the set of outside-in configurations.*

Lemma 10. *Let* $\mathcal{R}$ *be a confluent LEPRS,* $\overline{e_N}|_W$ *a goal without parameter-passing descendants, and* $\gamma \in \mathcal{U}_{\mathcal{R}}(\overline{e_N}|_W)$. *Then there exists an outside-in rewrite proof* ρ *of* $\gamma \in \mathcal{U}_{\mathcal{R}}(\overline{e_N}|_W)$.

Proof. By the standardization theorem for LEPRSs [17], there exists a rewrite proof ρ of $\gamma \in \mathcal{U}_{\mathcal{R}}(\overline{e_N}|_W)$ such that for any $k \in \{1, \dots, N\}$, $\rho(k)$ is an outside-in rewrite proof of $\gamma \in \mathcal{U}_{\mathcal{R}}(e_k|_W)$. Since $E|_W$ has no parameter-passing descendants, we conclude that $T := \langle \overline{e_N}|_W, \gamma, \rho\rangle$ is an outside-in configuration. □

We are ready now to state our main theorem.

Theorem 3. *Let* $E = \overline{e_N}$, $T = \langle E|_W, \gamma, \rho\rangle \in Cfg^{oi}$, $V \in \mathcal{P}_{fin}(\mathcal{V})$, *and* $e_k \in E$ *an equation which can be selected in an HOLN-step* π *starting from* $E|_W$. *If* $\Phi_{\mathrm{HOLN}}(T, V, e_k) = \langle \pi, T'\rangle$ *then* $T' \in Cfg^{oi}$.

Proof. Let $\pi : E|_W \Rightarrow_{\alpha, e_k, \theta} E'|_{W'}$ and $T' = \langle E'|_{W'}, \gamma', \rho'\rangle$ with $E' = \overline{e'_{N'}}$. To prove that $T' \in Cfg^{oi}$, we must show that for all $i \in \{1, \dots, N'\}$:

(i) $\rho'(i)$ is an outside-in rewrite derivation, and
(ii) If $e'_i = s \blacktriangleright t$ and $\rho'(i)$ has a rewrite step at position $1{\cdot}p$ such that no later rewrite steps take place above position $1{\cdot}p$ then $p \in Pat(t)$.

We distinguish two cases, whether e'_i is a descendant of e_k in π or not.
<u>*Case 1.*</u> Assume e'_i is a descendant of e_k in π. Then $\alpha \in \{[\mathrm{i}], [\mathrm{p}], [\mathrm{d}], [\mathrm{on}], [\mathrm{ov}]\}$.

If $\alpha = [\mathrm{p}]$ then $i = k$, $e'_i = e_k\theta$ and $\rho'(i) = \rho(k)$. Since $\rho(k)$ is outside-in, $\rho'(i)$ is outside-in too. If e'_i is a parameter-passing descendant then $e_k = s \blacktriangleright t$ and $e'_i = s\theta \blacktriangleright t\theta$. Assume $\rho'(i)$ has a rewrite step at position $1{\cdot}p$ and no later rewrite steps take place above $1{\cdot}p$. Since $\rho'(i) = \rho(k)$ and $\rho(k)$ satisfies condition (ii), we learn that $1{\cdot}p \in Pat(t)$. Since $Pat(t) \subseteq Pat(t\theta)$, we learn that $1{\cdot}p \in Pat(t\theta)$. Thus, $\rho'(i)$ satisfies condition (ii).

If $\alpha \in \{[\mathrm{i}],[\mathrm{d}]\}$ then, by the definition of Φ_{HOLN}, $\rho(k)$ is an outside-in rewrite derivation without rewrite steps at the head positions of the equational sides. By construction, $\rho'(i)$ is outside-in too. Moreover, if e'_i is a parameter-passing descendant, then e_k is also a parameter-passing descendant. Since $T \in Cfg^{oi}$, we learn that $\rho(k)$ satisfies condition (ii). This implies that $\rho'(i)$ satisfies condition (ii) too.

Assume $\alpha \in \{[\mathrm{on}],[\mathrm{ov}]\}$. Then π is of the form

$$\pi : E|_W \Rightarrow_{\alpha, e_k, \theta} (\overline{e_{k-1}\theta}, \overline{\lambda\overline{x}.s_n \blacktriangleright \lambda\overline{x}.t_n}, \lambda\overline{x}.r \simeq \lambda\overline{x}.t\theta, e_{k+1}\theta, \dots, e_N\theta)|_{W'}$$

where $e_k = \lambda\overline{x}.s \simeq \lambda\overline{x}.t$ with $\simeq \in \{\approx, \approx^{-1}, \rhd\}$ and $\lambda\overline{x}.s\theta = \lambda\overline{x}.f(\overline{s_n})$. If $e'_i = e'_{k+n} = \lambda\overline{x}.r \simeq \lambda\overline{x}.t\theta$ then from the definition of Φ_{HOLN} (see proof of Lemma 3.2) results that we can assume w.l.o.g. that $\rho'(k+n)$ is obtained from $\rho(k)$ by removing an initial sequence of rewrite steps. Since $\rho(k)$ satisfies conditions (i) and (ii), we conclude that $\rho'(k+n)$ satisfies them too. Otherwise, $e'_i = e'_{k+j-1} = \lambda\overline{x}.s_j \blacktriangleright \lambda\overline{x}.l_j$. Let's assume $\rho'(k+j-1)$ is a rewrite derivation of the form

$$e'_{k+j-1}\gamma' = \lambda\overline{x}.s_j\gamma' \blacktriangleright \lambda\overline{x}.l_j\gamma' \rightarrow^{l'_1 \to r_1}_{1\cdot p_1,\delta_1} \ldots \rightarrow^{l'_m \to r_m}_{1\cdot p_m,\delta_m} \lambda\overline{x}.l_j\gamma' \blacktriangleright \lambda\overline{x}.l_j\gamma'$$

which successively rewrites at positions $1{\cdot}p_1, \ldots, 1{\cdot}p_m$. By the construction of $\rho'(k+j-1)$ from $\rho(k)$ results that $\rho(k)$ has a rewrite sub-derivation of the form

$$\begin{aligned}&\lambda\overline{x}.f(\overline{s_{j-1}}, s_j, s_{j+1}, \ldots, s_n)\gamma' \simeq \lambda\overline{x}.t\gamma \rightarrow^{l'_1 \to r_1}_{1\cdot q\cdot j\cdot q_1,\delta_1} \cdots \\ &\rightarrow^{l'_m \to r_m}_{1\cdot q\cdot j\cdot q_m,\delta_1} \lambda\overline{x}.f(\overline{s_{j-1}}, l_j, s_{j+1}, \ldots, s_n)\gamma' \simeq \lambda\overline{x}.t\gamma \rightarrow^*_{\mathcal{R}} \lambda\overline{x}.u \simeq \lambda\overline{x}.u\end{aligned}$$

which starts by rewriting successively at positions $1{\cdot}q{\cdot}j{\cdot}q_1, \ldots, 1{\cdot}q{\cdot}j{\cdot}q_m$. In addition, $q{\cdot}q_i = p_i$ for all $i \in \{1, \ldots, m\}$. This implies that if $\rho'(k+j-1)$ is not outside-in then the displayed sub-derivation of $\rho(k)$ is not outside-in, which contradicts with the fact that $\rho(k)$ is outside-in. Thus, $\rho'(k+j-1)$ is outside-in. It remains to prove that $\rho'(k+j-1)$ satisfies property (ii). Assume $\rho'(k+j-1)$ has a rewrite step at position $1{\cdot}p$ such that no later rewrite steps take place above $1{\cdot}p$. In the outside-in rewrite sub-derivation of $\rho(k)$ depicted above, the first rewrite step above position $1{\cdot}q{\cdot}j{\cdot}q_k$ is at position $1{\cdot}q$ with $f(\overline{l_n}) \to r$. This implies that $p_k = q{\cdot}q_k \in Pat(l_k)$, i.e., (ii) holds.

Case 2. If e' is not a descendant of e in π then $e'_i = e_i\theta$ for some $i \in \{1, \ldots, N\} - \{k\}$, and $\rho'(i) = \rho(i)$. Since $T \in Cfg^{oi}$, we have that $\rho(i)$ is outside-in, and thus $\rho'(i)$ is outside-in too.

Next, we prove that $\rho'(i)$ satisfies condition (ii). Let's assume e'_i be a parameter-passing descendant. Then e'_i is of the form $s\theta \blacktriangleright t\theta$ where $e_i = s \blacktriangleright t$. Suppose $\rho'(i)$ has a rewrite step at position $1{\cdot}p$ such that no later rewrite steps take place above position $1{\cdot}p$. Since $\rho'(i) = \rho(i)$, we conclude that $p \in Pat(t)$. Since $Pat(t) \subseteq Pat(t\theta)$, we conclude that $\rho'(i)$ satisfies condition (ii). $\square$

We are ready now to define our first refinement of the calculus HOLN.

Definition 15 (HOLN$_1$). HOLN$_1$ *is the calculus defined by* HOLN$_1 \cup \{[\text{v}]\}$ *be the calculus obtained from HOLN by dropping the application of inference rule [on] to selected equations of the form* $\lambda\overline{x}.f(\overline{s_n}) \blacktriangleright \lambda\overline{x}.X(\overline{y_k})$ *where* $f \in \mathcal{F}_d$.

Main properties of HOLN$_1$

First, we prove that if $\mathcal{R}$ is a confluent LEPRS then HOLN$_1$ is strongly complete.

Theorem 4 (Strong completeness). *Let $\mathcal{R}$ be a confluent LEPRS. Then* HOLN$_1$ *is strongly complete.*

Proof. Let $V \in \mathcal{P}_{fin}(\mathcal{V})$ with $\mathcal{FV}(E_0\lfloor_{W_0}) \subseteq V$, and $\gamma \in \mathcal{U}_{\mathcal{R}}(E_0\lfloor_{W_0})$. We choose $\mathcal{A} := Cfg^{oi}$, $\succeq$ is the lexicographic combination of $\succeq_A$, $\succeq_B$, $\succeq_C$, and $\Phi_{\text{HOLN}_1} := \Phi_{\text{HOLN}}$. To make our generic proof outlined in Section 3.2 work for these choices, we only have to check that:

(a) there exists a configuration $T_0 = \langle E_0\lfloor_{W_0}, \gamma_0, \rho\rangle \in Cfg^{oi}$. To prove this, we first observe that $E_0\lfloor_{W_0}$ has no parameter-passing descendants because it is the initial goal. Therefore, by Lemma 10, we conclude that there exists $T_0 = \langle E_0\lfloor_{W_0}, \gamma_0, \rho\rangle \in Cfg^{oi}$.

(b) If $E = \overline{e_N}$, $T = \langle E\lfloor_W, \gamma, \rho\rangle \in A$, $V \in \mathcal{P}_{fin}(\mathcal{V})$ with $\mathcal{FV}(E\lfloor_W) \subseteq V$, $e_k \in E$ is selectable in an HOLN-step, and $\Phi_{\text{HOLN}}(T, V, e_k) = \langle \pi, T'\rangle$ then $T' \in A$ and $\pi \in step(\text{HOLN}_1)$. The fact that $T' \in A$ follows from Lemma 3.

Thus, it remains to check that $\pi \in step(\text{HOLN}_1)$, i.e., that if $e_k = \lambda\overline{x}.f(\overline{s_n}) \blacktriangleright \lambda\overline{x}.X(\overline{y_n})$ with $f \in \mathcal{F}_d$ then π is not an [on]-step.

Let's assume, by contrary, that $e_k = \lambda\overline{x}.f(\overline{s_n}) \blacktriangleright \lambda\overline{x}.X(\overline{y_n})$ with $f \in \mathcal{F}_d$ and π is an [on]-step. From the definition of Φ_{HOLN} we learn that $\rho(k)$ has a rewrite step at the head position of the left-hand side. Since $\rho(k)$ can not have rewrite steps above the head position of the left-hand side and $\langle E_k\lfloor_{W_k}, \gamma_k, \rho_k\rangle \in Cfg^{oi}$, by property (ii) we learn that the head position of $\lambda\overline{x}.X(\overline{y_n})$ should be a pattern position, which is a contradiction. Hence, π is not an [on]-step. □

Another important property of HOLN_1 is soundness, and this is an immediate consequence of soundness of HOLN.

Theorem 5 (Soundness). HOLN_1 *is sound.*

4.2 Eager Variable Elimination

In this subsection we present a further refinement of HOLN_1 to reduce the nondeterminism of solving parameter-passing descendants of the form $s \blacktriangleright t$ where t is a flex term. The refinement is defined under the assumption that $\mathcal{R}$ is a left-linear EPRS.

First, we note that, by Lemma 9, if $e = s \blacktriangleright t$ is a parameter-passing descendant in a HOLN_1 derivation, then t is of the form $\lambda\overline{x}.X(\overline{y})$ where $\overline{y}$ is a permutation of $\overline{x}$ and $X \notin \mathcal{FV}(s)$. This implies that $\theta := \{X \mapsto \lambda\overline{y}.s\}$ is a well defined substitution and $\theta \in \mathcal{U}(e) \subseteq \mathcal{U}_{\mathcal{R}}(e)$.

Definition 16 (HOLN_2). *We define the calculus* $\text{HOLN}_2 := \text{HOLN}_1 \cup \{[\text{v}]\}$, *where* [v] *is a new inference rule defined by:*

[v] **Variable elimination.**

$$(E_1, s \blacktriangleright \lambda\overline{x}.X(\overline{y}), E_2)\lfloor_W \Rightarrow_{[v],\theta} (E_1, E_2)\theta\lfloor_{W'}$$

where $\theta = \{X \mapsto \lambda\overline{y}.s\}$, *and assume that* [v] *has the highest priority.*

The calculus HOLN_2 is called *higher-order lazy narrowing with eager variable elimination* because it addresses the possibility to eagerly eliminate the free variable occurring in the right-hand side of a parameter-passing descendant by binding it to left-hand side of the equation.

Proving soundness of HOLN_2 is trivial.

Theorem 6 (Soundness). *The calculus* HOLN_2 *is sound.*

In the rest of this subsection we will prove that HOLN_2 is strongly complete. First, we prove the following auxiliary lemmata.

Lemma 11. *Let $\mathcal{R}$ be a LEPRS and $\Pi : E\lfloor_W \Rightarrow^*_\theta (E_1, s \blacktriangleright t, E_2)\lfloor_{W'}$ be an HOLN_2-derivation. Then*

(i) t is a linear fully-extended pattern,
(ii) $(\mathcal{FV}(E_1, s) \cup W') \cap \mathcal{FV}(t) = \emptyset$.

Proof. By Lemma 9, we know that properties (i) and (ii) hold if Π is a HOLN_1-derivation. Thus, we only have to check that properties (i) and (ii) are preserved by [v]-steps.

Let $E = (E_1, s \blacktriangleright \lambda\overline{x}.X(\overline{y}), E_2)$ and $\pi : E\lfloor_W \Rightarrow_{[v], s\blacktriangleright\lambda\overline{x}.X(\overline{y}), \theta} E'\lfloor_{W'}$ be a [v]-step. Assume that all parameter-passing descendants $s' \blacktriangleright t' \in E$ satisfy the conditions

(a) t' is a linear fully-extended pattern,
(b) If $E = (E', s' \blacktriangleright t', E'')$ then $(\mathcal{FV}(E', s') \cup W) \cap \mathcal{FV}(t') = \emptyset$.

Let $E' = (E''_1, s'' \blacktriangleright t'', E''_2)$. We must show that

(i) t'' is a linear fully-extended pattern,
(ii) $(\mathcal{FV}(E''_1, s'') \cup W') \cap \mathcal{FV}(t'') = \emptyset$.

By the definition of [v], $s'' \blacktriangleright t''$ is a descendant of an equation $s' \blacktriangleright t' \in (E_1, E_2)$. We can write $E = (E'_1, s' \blacktriangleright t', E'_2)$. By (b), we have $X \notin \mathcal{FV}(t'')$, thus $t'' = t'$. Since (a) implies that t' is a linear fully-extended pattern, we learn that (i) holds. To prove (ii), we distinguish two cases:

1. $s \blacktriangleright \lambda\overline{x}.X(\overline{y}) \in E'_1$. Then $Ran(\theta) \subset \mathcal{FV}(E'_1)$, and therefore $\mathcal{FV}(E''_1, s'') \subset \mathcal{FV}(E'_1, s')$. Thus, $\mathcal{FV}(t'') \cap \mathcal{FV}(E''_1, s'') = \mathcal{FV}(t') \cap \mathcal{FV}(E''_1, s'') \subseteq \mathcal{FV}(t') \cap \mathcal{FV}(E'_1, s') = \emptyset$. Also, by (i) we have $X \notin W$, therefore $W' = W$, therefore $W' \cap \mathcal{FV}(t'') = W \cap \mathcal{FV}(t') = \emptyset$. Hence, (ii) holds.
2. $s \blacktriangleright \lambda\overline{x}.X(\overline{y}) \in E'_2$. Then $X \notin \mathcal{FV}(E'_1, s' \blacktriangleright t') \cup W$, and thus $E''_1 = E'_1$, $s'' = s'$, $t'' = t'$, $W = W'$. In this case (ii) follows from (b). □

Lemma 12. *Let $E = (\overline{e_{k-1}}, s \blacktriangleright \lambda\overline{x}.X(\overline{y}), e_{k+1}, \ldots, e_N)$, $T = \langle E\lfloor_W, \gamma, \rho\rangle \in Cfg^{oi}$, $V \in \mathcal{P}_{fin}(\mathcal{V})$ with $\mathcal{FV}(E\lfloor_W) \subseteq V$, and $\pi : E\lfloor_W \Rightarrow_{[v], s\blacktriangleright\lambda\overline{x}.X(\overline{y}), \theta} \overline{e'_{N-1}}\lfloor_{W'}$. We define*

$$\gamma' = \gamma\lceil_{Dom(\gamma) - \{X\}},$$
$$\rho' : \{1, \ldots, N-1\} \to \bigcup_{i=1}^{N-1} Proof_{\mathcal{R}}(e'_i, \gamma'),\ \rho'(i) = \begin{cases} \rho(i) & \text{if } i < k, \\ \rho(i+1) & \text{if } k \le i < N \end{cases}$$
$$T' = \langle \overline{e'_{N-1}}\lfloor_{W'}, \gamma', \rho'\rangle$$

Then $T' \in Cfg^{oi}$ and $T \succ T'$. We denote the pair $\langle \pi, T'\rangle$ by $\Phi_{[v]}(T, V, e_k)$.

Proof. From $T \in Cfg^{oi}$ we learn that $\rho(k)$ is an empty rewrite proof of $\gamma \in \mathcal{U}_{\mathcal{R}}(s \blacktriangleright \lambda\overline{x}.X(\overline{y}))$. This implies that $X\gamma = \lambda\overline{y}.s\gamma$, therefore $\rho' \in Proof_{\mathcal{R}}(\overline{e'_{N-1}}, \gamma')$ and $T' \in Cfg^{oi}$. Obviously, $T =_A T'$, $T \succeq_B T'$ and $T \succ_c T'$, thus $T \succ T'$. □

We are ready now to claim that the calculus HOLN_2 is strongly complete.

Theorem 7 (Strong completeness). *Let $\mathcal{R}$ be a confluent LEPRs. Then* HOLN_2 *is strongly complete.*

Proof. We choose $(\mathcal{A}, \succeq)$ as in the proof of strong completeness of HOLN_1 and define the partial function

$$\Phi_{\mathrm{HOLN}_2} : Cfg^{oi} \to \mathcal{P}_{fin}(V) \times Eq(\mathcal{F}, \mathcal{V}) \to step(\mathrm{HOLN}_2) \times Cfg^{oi}$$
$$\Phi_{\mathrm{HOLN}_2}(T, V, e) = \begin{cases} \Phi_{[v]}(T, V, e) & \text{if } \Phi_{[v]}(T, V, e) \text{ is defined,} \\ \Phi_{\mathrm{HOLN}_1}(T, V, e) & \text{otherwise.} \end{cases}$$

□

4.3 Lazy Narrowing with Confluent Constructor LEPRSs

In this subsection we present a refinement of HOLN for confluent constructor LEPRSs. This refinement has been inspired by a similar refinement of the calculus LNC with leftmost equation selection strategy for left-linear constructor TRSs [10], and addresses the possibility to avoid the generation of parameter-passing descendants of the form $s \blacktriangleright t$ with $t \notin \mathcal{T}(\mathcal{F}_c, \mathcal{V})$. The effect of this behavior is that the nondeterminism between inference rules [on] and [d] disappears for parameter-passing descendants.

Definition 17. *A C-derivation respects strategy S_{left} if every C-step is applied to the leftmost selectable equation.*

It has been shown [10] that the calculus LNC with leftmost equation selection strategy $\mathcal{S}_{\mathrm{left}}$ does not generate parameter-passing descendants with defined symbols in the right-hand side. It can be shown that the calculus HOLN_2 has this property too.

Lemma 13. *Let $\mathcal{R}$ be a confluent constructor LEPRS. If Π is an* HOLN_2*-derivation which respects strategy S_{left} then all equations $s \blacktriangleright t$ in Π satisfy the condition $t \in \mathcal{T}(\mathcal{F}_c, \mathcal{V})$.*

Proof. Because $\mathcal{R}$ is a constructor LEPRS, the only way to generate equations $s \blacktriangleright t$ with $t \notin \mathcal{T}(\mathcal{F}_c, \mathcal{V})$ is via α-steps of the form

$$(E_1, \lambda\overline{x_n}.s' \blacktriangleright \lambda\overline{x_n}.X(\overline{y_n}), E, \underbrace{\lambda\overline{z_k}.X(\overline{t_n}) \simeq \lambda\overline{z_k}.u}_{e}, E_2)|_W \Rightarrow_{\alpha, e, \{X \mapsto \lambda\overline{x_n}.f(\ldots)\}} \cdots$$

with $X \in \mathcal{FV}(t')$, $f \in \mathcal{F}_d$ and $\alpha \in \{[\mathrm{i}],[\mathrm{ov}]\}$. Such an α-step can not occur in the HOLN_2-derivation because it would not respect strategy $\mathcal{S}_{\mathrm{left}}$: the equation $\lambda\overline{x_n}.s' \blacktriangleright \lambda\overline{x_n}.X(\overline{y_n})$ is more to the left than e, and it can be selected in a [v]-step. □

Corollary 2. *Let Π be an* HOLN_2*-refutation which respects strategy S_{left}. Then Π has no* [on]*-steps applied to parameter-passing descendants.*

Proof. Immediate consequence of Lemma 13. □

We are ready now to define our refinement for confluent constructor LEPRS.

Definition 18 (HOLN$_3$). *HOLN$_3$ is the calculus obtained from HOLN$_2$ by dropping the application of rule* [on] *to parameter-passing descendants.*

Theorem 8 (Soundness). *HOLN$_3$ is sound.*

Proof. Immediate consequence from the soundness property of HOLN$_2$. □

Theorem 9 (Completeness). *The calculus HOLN$_3$ with leftmost equation selection strategy is complete.*

Proof. Immediate consequence of Corollary 2. □

Remark 3. In our previous work of studying higher-order lazy narrowing with confluent constructor LEPRSs, we have proposed another calculus to avoid the generation of parameter-passing descendants $s \blacktriangleright t$ with defined symbol occurrences in the left-hand side. Our previous calculus differs from HOLN$_3$ by replacing the inference rule [v] with another rule called [c], which is shown below.

[c] **Constructor propagation.**
If $\exists s \blacktriangleright \lambda\overline{x_k}.X(\overline{y_k}) \in E_1$ and $s' = \lambda\overline{y_k}.s(\overline{x_k})$ then

$$(E_1, \lambda\overline{z_n}.X(\overline{t_k}) \simeq \lambda\overline{z_n}.u, E_2)\lfloor_W \Rightarrow_{[c],\epsilon} (E_1, \lambda\overline{z_n}.s'(\overline{t_k}) \simeq \lambda\overline{z_n}.u, E_2)\lfloor_{W'}$$

By giving to rule [c] the highest priority, it can be shown that this calculus with leftmost equation selection strategy is strongly complete. Also, this calculus is sound for goals consisting of unoriented equations. □

4.4 HOLN$_4$: a Refinement to Detect Redundant Equations

In this subsection we investigate the possibility to detect and eliminate equations which do not contribute to the computation of an answer substitution. We call such equations *redundant.* By detecting and eliminating such useless equations we save computing time.

Definition 19. *Let $(E_1, e, E_2)\lfloor_W$ be a goal. The equation $e \in E$ is called* redundant in $E\lfloor_W$ *if $e = \lambda\overline{x_k}.s \blacktriangleright \lambda\overline{x_k}.X(\overline{y_k})$ such that the following conditions are satisfied:*

(a) $\overline{y_k}$ is a permutation of the sequence $\overline{x_k}$,
(b) $X \notin \mathcal{FV}(E_1, \lambda\overline{x_k}.s, E_2)$.

We define the calculus HOLN$_4$:= HOLN $\cup$ {[rm]} where [rm] is a new inference rule defined as follows:

[rm] Removal of redundant equations.
If $e = \lambda\overline{x_k}.s \blacktriangleright \lambda\overline{x_k}.X(\overline{y_k})$ is redundant in $(E_1, e, E_2)\lfloor_W$ then

$$(E_1, e, E_2)\lfloor_W \Rightarrow_{[rm],e,\epsilon} (E_1, E_2)\lfloor_W$$

To give the calculus HOLN$_4$ a more deterministic computational behavior, we may assume that the inference rule [rm] has the highest priority, i.e., that when [rm] is applicable to a selected equation, then we discard the application of the other inference rules.

Main properties.
We argue that HOLN$_4$ is sound and strongly complete.

Theorem 10. HOLN$_4$ *is sound.*

Proof. It is enough to check that if there is an HOLN$_4$-derivation

$$\Pi_1 : E\lfloor_W \Rightarrow_\theta^* (E_1, e, E_2)\lfloor_{W'}$$

with e redundant in $(E_1, e, E_2)\lfloor_{W_1}$ then the HOLN$_4$-derivation

$$\Pi_2 : E\lfloor_W \Rightarrow_\theta^* (E_1, e, E_2)\lfloor_{W'} \Rightarrow_{[\mathrm{rm}],e} (E_1, E_2)\lfloor_{W'}$$

obtained by extending Π_1 with an [rm]-step, satisfies the condition:

$$\{\theta\gamma\lceil_{\mathcal{FV}(E\lfloor_W)} \mid \gamma \in \mathcal{U}_\mathcal{R}(E_1, e, E_2)\} = \{\theta\gamma'\lceil_{\mathcal{FV}(E\lfloor_W)} \mid \gamma' \in \mathcal{U}_\mathcal{R}(E_1, E_2)\}.$$

By Lemma 1, it is sufficient to prove that

$$\{\gamma\lceil_V \mid \gamma \in \mathcal{U}_\mathcal{R}((E_1, e, E_2)\lfloor_{W'})\} = \{\gamma'\lceil_V \mid \gamma' \in \mathcal{U}_\mathcal{R}((E_1, E_2)\lfloor_{W'})\}$$

where $V = \mathcal{FV}(E\theta) \cup W'$. To prove this fact, it is sufficient to note that $X \notin V$. This follows by an easy induction proof on the length of Π_1. □

Theorem 11. *Let $\mathcal{R}$ be a confluent PRS. Then* HOLN$_4$ *is strongly complete.*

Proof. We choose $\mathcal{A} = Cfg$ and the poset $\langle \mathcal{A}, \succeq \rangle$ where $\succeq$ is the lexicographic combination of orderings $\succeq_A, \succeq_B, \succeq_C$ used in the completeness proof of HOLN.

Let $E = \overline{e_N}$, $E\lfloor_W \in Goal(\mathcal{F}, \mathcal{V})$, $V \in \mathcal{P}_{fin}(\mathcal{V})$ with $\mathcal{FV}(E\lfloor_W) \subseteq V$, and $T = \langle E\lfloor_W, \gamma, \rho \rangle \in Cfg$. For any equation $e_k \in E$ which is selectable in an HOLN$_4$-step we define:

$$\Phi_{\mathrm{HOLN}_4}(T, V, e_k) = \begin{cases} \langle \pi, \langle E'\lfloor_W, \gamma, \rho' \rangle \rangle \text{ if } \pi : E\lfloor_W \Rightarrow_{[\mathrm{rm}],e_k} E'\lfloor_W \in \mathit{step}(\mathrm{HOLN}_3), \\ \qquad\qquad\qquad \text{where } \rho'(i) = \begin{cases} \rho(i) & \text{if } 1 \le i < k, \\ \rho(i+1) & \text{if } k \le i < N \end{cases} \\ \Phi_{\mathrm{HOLN}}(T, V, e_k) \text{ otherwise.} \end{cases}$$

Obviously, $T' \in Cfg$ and $T \succeq_A T', T \succeq_B T', T \succ_C T'$. Thus, $T \succ T'$. □

If $(E_1, \lambda\overline{x_k}.s \blacktriangleright \lambda\overline{x_k}.X(\overline{y_k}), E_2)\lfloor_W$ a goal in a HOLN-derivation then, by Lemma 9, we know that $X \notin \mathcal{FV}(E_1, \lambda\overline{x_k}.s)$ and $\overline{y_k}$ is a permuted sequence of $\overline{x_k}$. Thus, we have the following result:

Lemma 14. *Let $\mathcal{R}$ be a confluent LEPRS and $E_1, e, E_2\lfloor_W \in Goal(\mathcal{F}, \mathcal{V})$. The equation e is redundant in $(E_1, e, E_2)\lfloor_W$ if $e = \lambda\overline{x_k}.s \blacktriangleright \lambda\overline{x_k}.X(\overline{y_k})$ and $X \notin \mathcal{FV}(E_2)$.*

5 Conclusions and Future Work

We have presented a new lazy narrowing calculus HOLN for EPRS designed to compute solutions which are normalized with respect to a given set of variables, and then have presented four refinements to reduce its nondeterminism in a way which preserves completeness.

The results presented in this paper owe largely to a new formalism in which we treat a goal as a pair consisting of a sequence of equations and a set of variables for which we want to compute normalized answers. This formulation of narrowing has the following advantages:

- it clarifies problems and locates points for optimization during the refutation process of goals,
- it simplifies the soundness and completeness proofs of the calculi,
- it simplifies and systematizes the implementation of the lazy narrowing calculus as a computational model of a higher-order FLP system.

A promising direction of research is to extend HOLN to conditional PRSs. A program specification using conditional PRSs is much more expressive because it allows the user to impose equational conditions under which rewrite steps are allowed. Such an extension is quite straightforward, but it adds nondeterminism in guessing the values of the additional variables in the conditional part of rewrite rules. We expect that this source of nondeterminism can be avoided if we restrict to certain classes of conditional PRSs.

Acknowledgements. The work reported in this paper has been supported in part by the Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Scientific Research (B) 12480066, 2000-2002. Mircea Marin has been supported by JSPS postdoc fellowship 00096, 2000-2001.

References

1. H.P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*, volume 90. North Holland, second edition, 1984.
2. H.P. Barendregt. *Lambda calculi with types*, volume 2 of *Handbook of Logic in Computer Science*, pages 118–309. Oxford University Press, 1992.
3. N.G. de Bruijn.
4. J.C. Gónzalez-Moreno, M.T. Hortalá Gónzalez, and M. Ródriguez-Artalejo. A Higher-Order Rewriting Logic for Functional Logic Programming. In *Proceedings of International Conference on Logic Programming*, pages 153–167, Leuven, 1997. MIT Press.
5. J.R. Hindley and J.P. Seldin. *Introduction to Combinatorics and λ-Calculus*. Cambridge University Press, 1986.
6. G. Huèt. A Unification Algorithm for Typed λ-Calculus. *Theoretical Computer Science*, 1975.
7. M. Marin, T. Ida, and T. Suzuki. On Reducing the Search Space of Higher-Order Lazy Narrowing. In A. Middeldorp and T. Sato, editors, *FLOPS'99*, volume 1722 of *LNCS*, pages 225–240. Springer-Verlag, 1999.

8. M. Marin, A. Middeldorp, T. Ida, and T. Yanagi. LNCA: A Lazy Narrowing Calculus for Applicative Term Rewriting Systems. Technical Report ISE-TR-99-158, Institute of Information Sciences and Electronics, University of Tsukuba, Tsukuba, Japan, 1999.
9. R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. Technical report, Institut für Informatik, TU München, 1994.
10. A. Middeldorp and S. Okui. A deterministic lazy narrowing calculus. *Journal of Symbolic Computation*, 25(6):733–757, 1998.
11. A. Middeldorp, S. Okui, and T. Ida. Lazy narrowing: Strong completeness and eager variable elimination. *Theoretical Computer Science*, 167(1,2):95–130, 1996.
12. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1:497–536, 1991.
13. K. Nakahara, A. Middeldorp, and T. Ida. A complete narrowing calculus for higher-order functional logic programming. In *Seventh International Conference on Programming Languages: Implementations, Logics and Programs 95 (PLILP'95)*, volume 982 of *LNCS*, 1995.
14. T. Nipkow. Functional unification of higher-order patterns. In *Proceedings of 8th IEEE Symposium on Logic in Computer Science*, pages 64–74, 1993.
15. C. Prehofer. *Solving Higher-Order Equations. From Logic to Programming.* Foundations of Computing. Birkhäuser Boston, 1998.
16. Z. Qian. Linear unification of higher-order patterns. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proceedings of the Colloquium on Trees in Algebra and Programming*, volume 668 of *LNCS*, pages 391–405, Orsay, France, April 1993.
17. V. van Oostrom. Personal communication.
18. D.A. Wolfram. The clausal theory of types. *Theoretical Computer Science*, 21, 1993.

On the Modularity of Deciding Call-by-Need

Irène Durand[1] and Aart Middeldorp[2*]

[1] Université de Bordeaux I
33405 Talence, France
idurand@labri.u-bordeaux.fr

[2] Institute of Information Sciences and Electronics
University of Tsukuba, Tsukuba 305-8573, Japan
ami@is.tsukuba.ac.jp

Abstract. In a recent paper we introduced a new framework for the study of call by need computations. Using elementary tree automata techniques and ground tree transducers we obtained simple decidability proofs for a hierarchy of classes of rewrite systems that are much larger than earlier classes defined using the complicated sequentiality concept. In this paper we study the modularity of membership in the new hierarchy. Surprisingly, it turns out that none of the classes in the hierarchy is preserved under signature extension. By imposing various conditions we recover the preservation under signature extension. By imposing some more conditions we are able to strengthen the signature extension results to modularity for disjoint and constructor-sharing combinations.

1 Introduction

The seminal work of Huet and Lévy [9] on optimal normalizing reduction strategies for orthogonal rewrite systems marks the beginning of the quest for decidable subclasses of (orthogonal) rewrite systems that admit a computable call by need strategy for deriving normal forms. Call by need means that the strategy may only contract *needed* redexes, i.e., redexes that are contracted in every normalizing rewrite sequence. Huet and Lévy showed that for the class of orthogonal rewrite systems every term not in normal form contains a needed redex and repeated contraction of needed redexes results in a normal form if the term under consideration has a normal form. However, neededness is in general undecidable. In order to obtain a decidable approximation to neededness Huet and Lévy introduced in the second part of [9] the subclass of *strongly sequential* systems. In a strongly sequential system at least one of the needed redexes in every reducible term can be effectively computed. Moreover, Huet and Lévy showed that strong sequentiality is a decidable property of orthogonal rewrite systems. Several authors ([2, 6, 10, 13, 15, 17, 20, 19]) studied extensions of the class of strong sequential rewrite systems that preserve its good properties.

* Partially supported by the Grant-in-Aid for Scientific Research C(2) 11680338 of the Ministry of Education, Science, Sports and Culture of Japan.

In a previous paper (Durand and Middeldorp [6]) we presented a uniform framework for decidable call by need based on approximations. We introduced classes $\mathcal{CBN}_\alpha$ of rewrite systems parameterized by approximation mappings α. In [6] we identified the properties an approximation mapping α has to satisfy in order that the resulting class $\mathcal{CBN}_\alpha$ is decidable and every rewrite system in that class admits a computable normalizing call by need strategy. We showed moreover that our classes are much larger than the corresponding classes based on the difficult sequentiality concept.

Not much is known about the complexity of the problem of deciding membership in one of the classes that guarantees a computable call by need strategy to normal form. Comon [2] showed that strong sequentiality of a left-linear rewrite system can be decided in exponential time. Moreover, for left-linear rewrite systems satisfying the additional syntactic condition that whenever two proper subterms of left-hand sides are unifiable one of them matches the other, strong sequentiality can be decided in polynomial time. The class of forward-branching systems (Strandh [18]), a proper subclass of the class of orthogonal strongly sequential systems, coincides with the class of transitive systems (Toyama *et al.* [21]) and can be decided in quadratic time (Durand [5]). For classes higher in the hierarchy only double exponential upper bounds are known ([7]).

Consequently, it is of obvious importance to have results available that enable to split a rewrite system into smaller components such that membership in $\mathcal{CBN}_\alpha$ of the components implies membership of the original system in $\mathcal{CBN}_\alpha$. Such *modularity* results have been extensively studied for basic properties like confluence and termination, see [8, 14, 16] for overviews.

The simplest kind of modularity results are concerned with enriching the signature. Most properties of rewrite systems are preserved under *signature extension*. Two notable exceptions are the normal form property and the unique normal form property (with respect to reduction), see Kennaway *et al.* [11]. Also some properties dealing with ground terms are not preserved under signature extension. For instance, consider the property that every ground term has a normal form, the rewrite system consisting of the single rewrite rule $\mathsf{f}(x) \to \mathsf{f}(x)$, and add a new constant a. (A slightly more interesting example is obtained by adding the rewrite rule $\mathsf{f}(\mathsf{b}) \to \mathsf{b}$.) It turns out that for no α membership in $\mathcal{CBN}_\alpha$ is preserved under signature extension. In this paper we present several sufficient conditions which guarantee the preservation of signature extension. These results are presented in Section 3.

Since preservation under signature extension does not give rise to a very useful technique for splitting a system into smaller components, in Section 4 we consider combinations of systems without common function symbols as well as constructor-sharing combinations.

In the next section we briefly recall the framework of our earlier paper [6] for analyzing call by need computations in term rewriting and we introduce some useful definitions.

2 Preliminaries

We assume the reader is familiar with the basics of term rewriting ([1,4,12]). We recall the following definitions from [6]. We refer to the latter paper for motivation and examples. A term rewrite system (TRS for short) consists of rewrite rules $l \to r$ that satisfy $\mathsf{root}(l) \notin \mathcal{V}$ and $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(l)$. If the second condition is not imposed we find it useful to speak of extended TRSs (eTRSs). Such systems arise naturally when we approximate TRSs, as explained below.

Let $\mathcal{R}$ be an eTRS over a signature $\mathcal{F}$. The set of ground normal forms of $\mathcal{R}$ is denoted by $\mathsf{NF}(\mathcal{R},\mathcal{F})$. Let $\mathcal{R}_\bullet$ be the eTRS $\mathcal{R} \cup \{\bullet \to \bullet\}$ over the extended signature $\mathcal{F}_\bullet = \mathcal{F} \cup \{\bullet\}$. We say that redex Δ in $C[\Delta] \in \mathcal{T}(\mathcal{F})$ is $\mathcal{R}$-needed if there is no term $t \in \mathsf{NF}(\mathcal{R},\mathcal{F})$ such that $C[\bullet] \to_{\mathcal{R}}^* t$. So to determine $\mathcal{R}$-neededness of a redex Δ in $C[\Delta]$ we replace it by $\bullet$ and check whether we can derive a normal form without $\bullet$. Redex Δ is $\mathcal{R}$-needed only if this is impossible. Note that $\mathsf{NF}(\mathcal{R},\mathcal{F}) = \mathsf{NF}(\mathcal{R}_\bullet,\mathcal{F}_\bullet)$. For orthogonal TRSs $\mathcal{R}$-neededness coincides with neededness. We denote by $\mathsf{WN}(\mathcal{R},\mathcal{F})$ the set of all ground terms in $\mathcal{T}(\mathcal{F})$ that rewrite in $\mathcal{R}$ to a normal form in $\mathsf{NF}(\mathcal{R},\mathcal{F})$. If no confusion can arise, we just write $\mathsf{WN}(\mathcal{R})$.

Let $\mathcal{R}$ and $\mathcal{S}$ be eTRSs over the same signature $\mathcal{F}$. We say that $\mathcal{S}$ approximates $\mathcal{R}$ if $\to_{\mathcal{R}}^* \subseteq \to_{\mathcal{S}}^*$ and $\mathsf{NF}(\mathcal{R},\mathcal{F}) = \mathsf{NF}(\mathcal{S},\mathcal{F})$. An approximation mapping is a mapping α from TRSs to eTRSs with the property that $\alpha(\mathcal{R})$ approximates $\mathcal{R}$, for every TRS $\mathcal{R}$. In the following we write $\mathcal{R}_\alpha$ instead of $\alpha(\mathcal{R})$. The class of *left-linear* TRSs $\mathcal{R}$ such that every reducible term in $\mathcal{T}(\mathcal{F})$ has an $\mathcal{R}_\alpha$-needed redex is denoted by $\mathcal{CBN}_\alpha$. Here $\mathcal{F}$ denotes the signature of $\mathcal{R}$. We assume throughout the paper that ground terms exist. Although not explicitly stated in our earlier paper [6] we assume that $\mathcal{R}$ is a proper TRS (i.e., not an eTRS). For arbitrary left-linear eTRSs $\mathcal{R}$ we introduce a corresponding class $\mathcal{CBN}$ which contains the eTRSs with the property that every reducible ground term has an $\mathcal{R}$-needed redex. So a TRS $\mathcal{R}$ belongs to $\mathcal{CBN}_\alpha$ if and only if $\mathcal{R}_\alpha \in \mathcal{CBN}$.

Next we define the approximation mappings s, nv, and g. Let $\mathcal{R}$ be a TRS. The *strong* approximation $\mathcal{R}_{\mathrm{s}}$ is obtained from $\mathcal{R}$ by replacing the right-hand side of every rewrite rule by a variable that does not occur in the corresponding left-hand side. The *nv* approximation $\mathcal{R}_{\mathrm{nv}}$ is obtained from $\mathcal{R}$ by replacing the variables in the right-hand sides of the rewrite rules by pairwise distinct variables that do not occur in the corresponding left-hand sides. An eTRS is called growing if for every rewrite rule $l \to r$ the variables in $\mathcal{V}\mathsf{ar}(l) \cap \mathcal{V}\mathsf{ar}(r)$ occur at depth 1 in l. The *growing* approximation $\mathcal{R}_{\mathrm{g}}$ is defined as any growing eTRS that is obtained from $\mathcal{R}$ by renaming the variables in the right-hand sides that occur at a depth greater than 1 in the corresponding left-hand sides. In [6] we showed that for a left-linear TRS $\mathcal{R}$ and $\alpha \in \{\mathrm{s},\mathrm{nv},\mathrm{g}\}$ membership of $\mathcal{R}$ in $\mathcal{CBN}_\alpha$ is decidable.

We conclude this preliminary section with some easy definitions. A rewrite rule $l \to r$ of an eTRS is collapsing if r is a variable. A redex with respect to a collapsing rewrite rule is also called collapsing and so is an eTRS that contains a collapsing rewrite rule. A redex is called *flat* if it does not contain smaller redexes. Let $\mathcal{R}$ be a TRS over the signature $\mathcal{F}$. A function symbol in $\mathcal{F}$ is called defined if it is the root symbol of a left-hand side of a rewrite rule in $\mathcal{R}$. All other function symbols in $\mathcal{F}$ are called constructors. A term without defined symbols is called a constructor term. We say that $\mathcal{R}$ is a constructor system (CS for short) if the arguments of the left-hand side of a rewrite rule are constructor terms.

Let $\mathcal{R}$ be an eTRS over the signature $\mathcal{F}$ and let $\mathcal{G} \subseteq \mathcal{F}$. We denote by $\mathsf{WN}(\mathcal{R},\mathcal{F},\mathcal{G})$ the set of terms in $\mathcal{T}(\mathcal{G})$ that have a normal form with respect to $(\mathcal{R},\mathcal{F})$. The subset of $\mathsf{WN}(\mathcal{R},\mathcal{F},\mathcal{G})$ consisting of those terms that admit a normalizing rewrite sequence in $(\mathcal{R},\mathcal{F})$ containing a root rewrite step is denoted by $\mathsf{WNR}(\mathcal{R},\mathcal{F},\mathcal{G})$. If $\mathcal{F} = \mathcal{G}$ then we just write $\mathsf{WNR}(\mathcal{R},\mathcal{F})$ or even $\mathsf{WNR}(\mathcal{R})$ if the signature is clear from the context. We also find it convenient to write $\mathsf{WN}_\bullet(\mathcal{R},\mathcal{F},\mathcal{G})$ for $\mathsf{WN}(\mathcal{R}_\bullet,\mathcal{F}_\bullet,\mathcal{G}_\bullet)$ and $\mathsf{WNR}_\bullet(\mathcal{R},\mathcal{F},\mathcal{G})$ for $\mathsf{WNR}(\mathcal{R}_\bullet,\mathcal{F}_\bullet,\mathcal{G}_\bullet)$. A reducible term without needed redexes is called *free*. A *minimal* free term has the property that its proper subterms are not free.

3 Signature Extension

In this section we study the question whether membership in $\mathcal{CBN}_\alpha$ is preserved after adding new function symbols.

Definition 1. *We say that a class C of eTRSs is* preserved under signature extension *if $(\mathcal{R},\mathcal{G}) \in C$ for all $(\mathcal{R},\mathcal{F}) \in C$ and $\mathcal{F} \subseteq \mathcal{G}$.*

The results we obtain in this section are summarized below. In the result marked with $(*)$ signature extension only holds if we further require that the set of $\mathcal{R}_{\mathsf{nv}}$-normalizable terms over the original signature is not increased.

approximation	sufficient conditions	Theorem
strong	$\exists$ ground normal form	1
nv	$\exists$ external normal form	2
nv	$\mathcal{R}$ is collapsing or $\mathcal{R}$ is a CS $(*)$	3
growing	$\exists$ external normal form	2

All our proofs follow the same strategy. We consider a TRS $\mathcal{R}$ over a signature $\mathcal{F}$ such that $(\mathcal{R},\mathcal{F}) \in \mathcal{CBN}_\alpha$. Let $\mathcal{G}$ be an extension of $\mathcal{F}$. Assuming that $(\mathcal{R},\mathcal{G}) \notin \mathcal{CBN}_\alpha$, we consider a minimal $(\mathcal{R}_\alpha,\mathcal{G})$-free term t in $\mathcal{T}(\mathcal{G})$. By replacing the maximal subterms of t that start with a function symbol in $\mathcal{G}\backslash\mathcal{F}$—such subterms will be called *aliens* or more precisely $\mathcal{G}\backslash\mathcal{F}$-aliens in the sequel—by a suitable term in $\mathcal{T}(\mathcal{F})$, we obtain an $(\mathcal{R}_\alpha,\mathcal{F})$-free term t' in $\mathcal{T}(\mathcal{F})$. Hence $(\mathcal{R},\mathcal{F}) \notin \mathcal{CBN}_\alpha$, contradicting the assumption.

Our first example shows that $\mathcal{CBN}_\mathsf{s}$ is not preserved under signature extension.

Example 1. Consider the TRS

$$\mathcal{R} = \left\{ \begin{array}{rcl} \mathsf{f}(x,\mathsf{g}(y),\mathsf{h}(z)) & \to & x \\ \mathsf{f}(\mathsf{h}(z),x,\mathsf{g}(y)) & \to & x \\ \mathsf{f}(\mathsf{g}(y),\mathsf{h}(z),x) & \to & x \\ \mathsf{a} & \to & \mathsf{a} \end{array} \right\}$$

over the signature $\mathcal{F}$ consisting of all symbols appearing in the rewrite rules. As $\mathsf{NF}(\mathcal{R},\mathcal{F}) = \varnothing$, trivially $(\mathcal{R},\mathcal{F}) \in \mathcal{CBN}_\mathsf{s}$. Let $\mathcal{G} = \mathcal{F} \cup \{\mathsf{b}\}$ with b a constant. We have $(\mathcal{R},\mathcal{G}) \notin \mathcal{CBN}_\mathsf{s}$ as the term $\mathsf{f}(\mathsf{a},\mathsf{a},\mathsf{a})$ has no $(\mathcal{R}_\mathsf{s},\mathcal{G})$-needed redex:

$$\begin{array}{lllllll}
\mathsf{f}(\bullet,\mathsf{a},\mathsf{a}) & \to_\mathsf{s} & \mathsf{f}(\bullet,\mathsf{g}(\mathsf{a}),\mathsf{a}) & \to_\mathsf{s} & \mathsf{f}(\bullet,\mathsf{g}(\mathsf{a}),\mathsf{h}(\mathsf{a})) & \to_\mathsf{s} & \mathsf{b}\\
\mathsf{f}(\mathsf{a},\bullet,\mathsf{a}) & \to_\mathsf{s} & \mathsf{f}(\mathsf{h}(\mathsf{a}),\bullet,\mathsf{a}) & \to_\mathsf{s} & \mathsf{f}(\mathsf{h}(\mathsf{a}),\bullet,\mathsf{g}(\mathsf{a})) & \to_\mathsf{s} & \mathsf{b}\\
\mathsf{f}(\mathsf{a},\mathsf{a},\bullet) & \to_\mathsf{s} & \mathsf{f}(\mathsf{g}(\mathsf{a}),\mathsf{a},\bullet) & \to_\mathsf{s} & \mathsf{f}(\mathsf{g}(\mathsf{a}),\mathsf{h}(\mathsf{a}),\bullet) & \to_\mathsf{s} & \mathsf{b}
\end{array}$$

The above example is interesting since it furthermore shows that $\mathcal{CBN}_\mathsf{s}$ *properly* includes the class of strongly sequential TRSs defined by Huet and Lévy [9], contrary to the claim in [6] that these two classes coincide.

One may wonder whether there are any nontrivial counterexamples, where nontrivial means that the set of ground normal forms is nonempty. Surprisingly, the answer is yes, provided we consider an approximation map α that is at least as good as nv.

Example 2. Consider the TRS

$$\mathcal{R} = \left\{\begin{array}{lllllllll}
\mathsf{f}(x,\mathsf{a},\mathsf{b}) & \to & \mathsf{g}(x) & \mathsf{f}(\mathsf{a},\mathsf{a},\mathsf{a}) & \to & \mathsf{g}(\mathsf{a}) & \mathsf{g}(\mathsf{a}) & \to & \mathsf{g}(\mathsf{a})\\
\mathsf{f}(\mathsf{b},x,\mathsf{a}) & \to & \mathsf{g}(x) & \mathsf{f}(\mathsf{b},\mathsf{b},\mathsf{b}) & \to & \mathsf{g}(\mathsf{a}) & \mathsf{g}(\mathsf{b}) & \to & \mathsf{g}(\mathsf{b})\\
\mathsf{f}(\mathsf{a},\mathsf{b},x) & \to & \mathsf{g}(x) & \mathsf{e}(x) & \to & x & & &
\end{array}\right\}$$

over the signature $\mathcal{F}$ consisting of all symbols appearing in the rewrite rules. First we show that $(\mathcal{R},\mathcal{F}) \in \mathcal{CBN}_\mathsf{nv}$. It is not difficult to show that the only $(\mathcal{R}_\mathsf{nv},\mathcal{F})$-normalizable terms are a, b, and $\mathsf{e}(t)$ for every $t \in \mathcal{T}(\mathcal{F})$. Since a and b are normal forms, we only have to show that every $\mathsf{e}(t)$ contains an $(\mathcal{R}_\mathsf{nv},\mathcal{F})$-needed redex, which is easy since $\mathsf{e}(t)$ itself is an $(\mathcal{R}_\mathsf{nv},\mathcal{F})$-needed redex. Let $\mathcal{G} = \mathcal{F} \cup \{\mathsf{c}\}$ with c a constant. We have $(\mathcal{R},\mathcal{G}) \notin \mathcal{CBN}_\mathsf{nv}$ as the term $\mathsf{f}(\mathsf{e}(\mathsf{a}),\mathsf{e}(\mathsf{a}),\mathsf{e}(\mathsf{a}))$ has no $(\mathcal{R}_\mathsf{nv},\mathcal{G})$-needed redex:

$$\begin{array}{lllllll}
\mathsf{f}(\bullet,\mathsf{e}(\mathsf{a}),\mathsf{e}(\mathsf{a})) & \to_\mathsf{nv} & \mathsf{f}(\bullet,\mathsf{a},\mathsf{e}(\mathsf{a})) & \to_\mathsf{nv} & \mathsf{f}(\bullet,\mathsf{a},\mathsf{b}) & \to_\mathsf{nv} & \mathsf{g}(\mathsf{c})\\
\mathsf{f}(\mathsf{e}(\mathsf{a}),\bullet,\mathsf{e}(\mathsf{a})) & \to_\mathsf{nv} & \mathsf{f}(\mathsf{b},\bullet,\mathsf{e}(\mathsf{a})) & \to_\mathsf{nv} & \mathsf{f}(\mathsf{b},\bullet,\mathsf{a}) & \to_\mathsf{nv} & \mathsf{g}(\mathsf{c})\\
\mathsf{f}(\mathsf{e}(\mathsf{a}),\mathsf{e}(\mathsf{a}),\bullet) & \to_\mathsf{nv} & \mathsf{f}(\mathsf{a},\mathsf{e}(\mathsf{a}),\bullet) & \to_\mathsf{nv} & \mathsf{f}(\mathsf{a},\mathsf{b},\bullet) & \to_\mathsf{nv} & \mathsf{g}(\mathsf{c})
\end{array}$$

For $\alpha = \mathsf{s}$ there is no nontrivial counterexample.

Theorem 1. *The subclass of $\mathcal{CBN}_\mathsf{s}$ consisting of all orthogonal TRSs $(\mathcal{R},\mathcal{F})$ such that $\mathsf{NF}(\mathcal{R},\mathcal{F}) \neq \varnothing$ is preserved under signature extension.* □

We refrain from giving the proof at this point since the statement easily follows from Theorem 3 below, whose proof is presented in detail. Our second result states that the subclass of $\mathcal{CBN}$ consisting of all eTRSs $\mathcal{R}$ with the property defined below is preserved under signature extension.

Definition 2. *We say that an eTRS $\mathcal{R}$ has* external *normal forms if there exists a ground normal form which is not an instance of a proper non-variable subterm of a left-hand sides of a rewrite rule in $\mathcal{R}$.*

Note that the TRS of Example 2 lacks external normal forms as both ground normal forms a and b appear in the left-hand sides of the rewrite rules. Further note that it is decidable whether a left-linear TRS has external normal forms by straightforward tree automata techniques. Finally note that the external normal form property is satisfied whenever there exists a constant not occurring in the left-hand sides of the rewrite rules.

Before proving our second result we present a useful lemma which is used repeatedly in the sequel.

Lemma 1. *Let $\mathcal{R}$ be a left-linear eTRS. Every minimal free term belongs to* $\mathsf{WNR}(\mathcal{R})$.

Proof. Let $\mathcal{F}$ be the signature of $\mathcal{R}$ and let $t \in \mathcal{T}(\mathcal{F})$ be a minimal free term. For every redex position p in t we have $t[\bullet]_p \in \mathsf{WN}_\bullet(\mathcal{R})$. Let p' be the minimum position above p at which a contraction takes place in any rewrite sequence from $t[\bullet]_p$ to a normal form in $\mathcal{T}(\mathcal{F})$ and define $P = \{p' \mid p \text{ is a redex position in } t\}$. Let p^* be a minimal position in P. We show that $p^* = \epsilon$. If $p^* > \epsilon$ then we consider the term $t|_{p^*}$. Let q be a redex position in $t|_{p^*}$. There exists a redex position p in t such that $p = p^*q$. We have $t|_{p^*}[\bullet]_q = (t[\bullet]_p)|_{p^*} \in \mathsf{WN}_\bullet(\mathcal{R})$ by the definition of p^*. Since $t|_{p^*}$ has at least one redex, it follows that $t|_{p^*}$ is free. As $t|_{p^*}$ is a proper subterm of t we obtain a contradiction to the minimality of t. Hence $p^* = \epsilon$. So there exists a redex position p in t and a rewrite sequence A: $t[\bullet]_p \rightarrow^+_{\mathcal{R},\mathcal{F}_\bullet} u \in \mathsf{NF}(\mathcal{R},\mathcal{F})$ that contains a root rewrite step. Because $\mathcal{R}$ is left-linear and $\bullet$ does not occur in the rewrite rules of $\mathcal{R}$, $\bullet$ cannot contribute to this sequence. It follows that if we replace in A every occurrence of $\bullet$ by $t|_p$ we obtain an $(\mathcal{R},\mathcal{F})$-rewrite sequence from t to u with a root rewrite step. □

In particular, minimal free terms are not root-stable.

Theorem 2. *The subclass of $\mathcal{CBN}$ consisting of all left-linear eTRSs with external normal forms is preserved under signature extension.*

Proof. Let $(\mathcal{R},\mathcal{F}) \in \mathcal{CBN}$ and let $c \in \mathsf{NF}(\mathcal{R},\mathcal{F})$ be an external normal form. Let $\mathcal{F} \subseteq \mathcal{G}$. We have to show that $(\mathcal{R},\mathcal{G}) \in \mathcal{CBN}$. Suppose to the contrary that $(\mathcal{R},\mathcal{G}) \notin \mathcal{CBN}$. According to Lemma 1 there exists a term $t \in \mathsf{WNR}(\mathcal{R},\mathcal{G})$ without $(\mathcal{R},\mathcal{G})$-needed redex. Let t' be the term in $\mathcal{T}(\mathcal{F})$ obtained from t by replacing every $\mathcal{G}\backslash\mathcal{F}$-alien by c. Because t is not root-stable and $\mathcal{R}$ left-linear, t' must be reducible. Hence t' contains an $(\mathcal{R},\mathcal{F})$-needed redex Δ, say at position p. Because c is an external normal form, Δ is also a redex in t and hence there exists a rewrite sequence $t[\bullet]_p \rightarrow^+_{\mathcal{R},\mathcal{G}_\bullet} u$ with $u \in \mathsf{NF}(\mathcal{R},\mathcal{G})$. If we replace in this rewrite sequence every $\mathcal{G}\backslash\mathcal{F}$-alien by c, we obtain a rewrite sequence $t'[\bullet]_p \rightarrow^+_{\mathcal{R},\mathcal{F}_\bullet} u'$. Because c does not unify with a proper non-variable subterm of a left-hand side of a rewrite rule, it follows that $u' \in \mathsf{NF}(\mathcal{R},\mathcal{F})$. Hence Δ is not an $(\mathcal{R},\mathcal{F})$-needed redex in t', yielding the desired contradiction. □

Note that for $\mathcal{CBN}_s$ the above theorem is a special case of Theorem 1 since the existence of an external normal form implies the existence of a ground normal form.

In the remainder of this section we present a signature extension result for TRSs without external normal form. Such TRSs are quite common (see also Lemma 4 below).

Example 3. Consider the TRS

$$\mathcal{R} = \left\{ \begin{array}{lllllll} 0 + y & \to & y & \quad & \mathsf{s}(x) + y & \to & \mathsf{s}(x + y) \\ 0 \times y & \to & 0 & \quad & \mathsf{s}(x) \times y & \to & x \times y + y \end{array} \right\}$$

over the signature $\mathcal{F}$ consisting of all symbols appearing in the rewrite rules. Since every normal form is of the form $\mathsf{s}^n(0)$ for some $n \geqslant 0$, it follows that $\mathcal{R}$ lacks external normal forms.

We start with some preliminary results.

Definition 3. *Let $\mathcal{R}$ be a TRS. Two redexes Δ_1, Δ_2 are called* pattern equal, *denoted by $\Delta_1 \approx \Delta_2$, if they have the same redex pattern, i.e., they are redexes with respect to the same rewrite rule.*

Lemma 2. *Let $\mathcal{R}$ be an orthogonal TRS, $\alpha \in \{\mathrm{s}, \mathrm{nv}\}$, and suppose that $\Delta \approx \Delta'$. If $C[\Delta] \in \mathsf{WN}(\mathcal{R}_\alpha)$ then $C[\Delta'] \in \mathsf{WN}(\mathcal{R}_\alpha)$.*

Proof. Let $C[\Delta] \to^* t$ be a normalizing rewrite sequence in $\mathcal{R}_\alpha$. If we replace every descendant of Δ by Δ' then we obtain a (possibly shorter) normalizing rewrite sequence $C[\Delta'] \to^* t$. The reason is that every descendant Δ'' of Δ satisfies $\Delta'' \approx \Delta$ due to orthogonality and hence if Δ'' is contracted to some term u then Δ rewrites to the same term because the variables in the right-hand sides of the rewrite rules in $\mathcal{R}_\alpha$ are fresh. Moreover, as t is a normal form, there are no descendants of Δ left. Note that the resulting sequence can be shorter since rewrite steps below a descendant of Δ are not mimicked. □

The above lemma does not hold for the growing approximation, as shown by the following example.

Example 4. Consider the TRS $\mathcal{R} = \{\mathsf{f}(x) \to x, \mathsf{a} \to \mathsf{b}, \mathsf{c} \to \mathsf{c}\}$. We have $\mathcal{R}_\mathrm{g} = \mathcal{R}$. Consider the redexes $\Delta = \mathsf{f}(\mathsf{a})$ and $\Delta' = \mathsf{f}(\mathsf{c})$. Clearly $\Delta \approx \Delta'$. Redex Δ admits the normal form b, but Δ' has no normal form.

Lemma 3. *Let $\mathcal{R}$ be an orthogonal TRS over a signature $\mathcal{F}$, $\alpha \in \{\mathrm{s}, \mathrm{nv}\}$, and $\mathcal{F} \subseteq \mathcal{G}$. If $\mathsf{WN}(\mathcal{R}_\alpha, \mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{G}, \mathcal{F})$ then $\mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{F}) = \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G}, \mathcal{F})$.*

Proof. The inclusion $\mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{F}) \subseteq \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G}, \mathcal{F})$ is obvious. For the reverse inclusion we reason as follows. Let $t \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G}, \mathcal{F})$ and consider a rewrite sequence A in $(\mathcal{R}_\alpha, \mathcal{G}_\bullet)$ that normalizes t. We may write $t = C[t_1, \ldots, t_n]$ such that $t_1, \ldots, t_n$ are the maximal subterms of t that are rewritten in A at their root positions. Hence A can be rearranged into A':

$$t \to^*_{\mathcal{R}_\alpha, \mathcal{G}_\bullet} C[\Delta_1, \ldots, \Delta_n] \to^*_{\mathcal{R}_\alpha, \mathcal{G}_\bullet} C[u_1, \ldots, u_n]$$

for some redexes $\Delta_1, \dots, \Delta_n$ and normal form $C[u_1, \dots, u_n] \in \mathcal{T}(\mathcal{G})$. Since the context C cannot contain $\bullet$, all occurrences of $\bullet$ are in the substitution parts of the redexes $\Delta_1, \dots, \Delta_n$. If we replace in $C[\Delta_1, \dots, \Delta_n]$ every $\mathcal{G}_\bullet \backslash \mathcal{F}$-alien by some ground term $c \in \mathcal{T}(\mathcal{F})$, we obtain a term $t' = C[\Delta'_1, \dots, \Delta'_n]$ with $\Delta'_i \in \mathcal{T}(\mathcal{F})$ and $\Delta_i \approx \Delta'_i$ for every i. Repeated application of Lemma 2 yields $t' \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G})$. Because $\bullet$ cannot contribute to the creation of a normal form, we actually have $t' \in \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{G})$ and thus $t' \in \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{G}, \mathcal{F})$ as $t' \in \mathcal{T}(\mathcal{F})$. The assumption yields $t' \in \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{F})$. Since $\mathsf{WN}(\mathcal{R}_\alpha, \mathcal{F}) \subseteq \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{F})$ clearly holds, we obtain $t' \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{F})$. Now, if we replace in the first part of A' every $\mathcal{G} \backslash \mathcal{F}$-alien by c then we obtain a (possibly shorter) rewrite sequence $t \rightarrow^*_{\mathcal{R}_\alpha, \mathcal{F}_\bullet} C[\Delta''_1, \dots, \Delta''_n] \in \mathcal{T}(\mathcal{F}_\bullet)$ with $\Delta_i \approx \Delta''_i$ and thus also $\Delta'_i \approx \Delta''_i$ for every i. Repeated application of Lemma 2 yields $C[\Delta''_1, \dots, \Delta''_n] \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{F})$ and therefore $t \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{F})$ as desired. □

We note that for $\alpha = \mathrm{s}$ the preceding lemma is a simple consequence of Lemma 6 below. The restriction to $\alpha \in \{\mathrm{s}, \mathrm{nv}\}$ is essential. For $\mathcal{R}_\mathrm{g}$ we have the following counterexample.

Example 5. Consider the orthogonal TRS

$$\mathcal{R} = \left\{ \begin{array}{lll@{\qquad}lll} \mathsf{f}(x, \mathsf{a}, y) & \rightarrow & \mathsf{g}(y) & \mathsf{h}(\mathsf{a}) & \rightarrow & \mathsf{a} \\ \mathsf{f}(\mathsf{a}, \mathsf{b}(x), y) & \rightarrow & \mathsf{a} & \mathsf{h}(\mathsf{b}(x)) & \rightarrow & \mathsf{i}(x) \\ \mathsf{f}(\mathsf{b}(x), \mathsf{b}(y), z) & \rightarrow & \mathsf{a} & \mathsf{i}(\mathsf{a}) & \rightarrow & \mathsf{a} \\ \mathsf{g}(\mathsf{a}) & \rightarrow & \mathsf{b}(\mathsf{g}(\mathsf{a})) & \mathsf{i}(\mathsf{b}(x)) & \rightarrow & \mathsf{b}(\mathsf{a}) \\ \mathsf{g}(\mathsf{b}(x)) & \rightarrow & \mathsf{g}(\mathsf{a}) & \mathsf{j}(x, \mathsf{a}) & \rightarrow & \mathsf{f}(x, \mathsf{h}(\mathsf{b}(\mathsf{a})), \mathsf{a}) \\ & & & \mathsf{j}(x, \mathsf{b}(y)) & \rightarrow & \mathsf{f}(x, \mathsf{h}(\mathsf{b}(\mathsf{a})), y) \end{array} \right\}$$

over the signature $\mathcal{F}$ consisting of all symbols appearing in the rewrite rules and let $\mathcal{S} = \mathcal{R}_\mathrm{g}$. Let $\mathcal{G} = \mathcal{F} \cup \{\mathsf{c}\}$ with c a constant. With some effort one can check that $\mathsf{WN}(\mathcal{S}, \mathcal{F}) = \mathsf{WN}(\mathcal{S}, \mathcal{G}, \mathcal{F})$. However, $\mathsf{WN}_\bullet(\mathcal{S}, \mathcal{F})$ is different from $\mathsf{WN}_\bullet(\mathcal{S}, \mathcal{G}, \mathcal{F})$ as witnessed by the term $t = \mathsf{j}(\bullet, \mathsf{b}(\mathsf{a}))$. In $(\mathcal{S}_\bullet, \mathcal{G}_\bullet)$ we have $t \rightarrow \mathsf{f}(\bullet, \mathsf{h}(\mathsf{b}(\mathsf{a})), \mathsf{c}) \rightarrow^+ \mathsf{f}(\bullet, \mathsf{a}, \mathsf{c}) \rightarrow \mathsf{g}(\mathsf{c})$, hence $t \in \mathsf{WN}_\bullet(\mathcal{S}, \mathcal{G}, \mathcal{F})$, but one easily verifies that t does not have a normal form in $(\mathcal{S}_\bullet, \mathcal{F}_\bullet)$.

Lemma 4. *The set of ground normal forms of a CS without external normal forms coincides with the set of ground constructor terms.*

Proof. Clearly every ground constructor term is a normal form. Suppose there exists a ground normal form that contains a defined function symbol. Since every subterm of a normal form is a normal form, there must be a ground normal form t whose root symbol is a defined function symbol. By definition of external normal form, t must be an instance of a proper non-variable subterm of a left-hand side l. This implies that a proper subterm of l contains a defined function symbol, contradicting the assumption that the TRS under consideration is a CS. □

Lemma 5. *Let $(\mathcal{R}, \mathcal{F})$ and $(\mathcal{S}, \mathcal{G})$ be orthogonal TRSs and $\alpha \in \{\mathrm{s}, \mathrm{nv}\}$ such that $(\mathcal{R}, \mathcal{F}) \subseteq (\mathcal{S}, \mathcal{G})$ and $\mathsf{WN}(\mathcal{S}_\alpha, \mathcal{G}, \mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{F})$. If $t \in \mathsf{WNR}(\mathcal{S}_\alpha, \mathcal{G})$ and $\mathsf{root}(t) \in \mathcal{F}$ then there exists a flat redex Ξ in $\mathcal{T}(\mathcal{F})$. Moreover, if $\mathcal{R}_\alpha$ is collapsing then we may assume that Ξ is $\mathcal{R}_\alpha$-collapsing.*

Proof. From $t \in \mathsf{WNR}(\mathcal{S}_\alpha, \mathcal{G})$ we infer that $t \rightarrow^*_{\mathcal{S}_\alpha,\mathcal{G}} \Delta$ for some redex $\Delta \in \mathsf{WN}(\mathcal{S}_\alpha, \mathcal{G})$. By considering the first such redex it follows that Δ is a redex with respect to $(\mathcal{R}_\alpha, \mathcal{G})$. If we replace in Δ the subterms below the redex pattern by an arbitrary ground term in $\mathcal{T}(\mathcal{F})$ then we obtain a redex $\Delta' \in \mathcal{T}(\mathcal{F})$ with $\Delta \approx \Delta'$. Lemma 2 yields $\Delta' \in \mathsf{WN}(\mathcal{S}_\alpha, \mathcal{G})$ and thus $\Delta' \in \mathsf{WN}(\mathcal{S}_\alpha, \mathcal{G}, \mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{F})$. Hence $\mathsf{NF}(\mathcal{R}, \mathcal{F}) = \mathsf{NF}(\mathcal{R}_\alpha, \mathcal{F}) \neq \varnothing$. Therefore, using orthogonality, we obtain a flat redex $\Xi \in \mathcal{T}(\mathcal{F})$ by replacing the variables in the left-hand side of any rewrite rule in $\mathcal{R}$ by terms in $\mathsf{NF}(\mathcal{R}, \mathcal{F})$. If $\mathcal{R}_\alpha$ is a collapsing then we take any $\mathcal{R}_\alpha$-collapsing rewrite rule. ☐

We are now ready for the final signature extension result of this section. The condition $\mathsf{WN}(\mathcal{R}_\alpha, \mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{G}, \mathcal{F})$ expresses that the set of $\mathcal{R}_\alpha$-normalizable terms in $\mathcal{T}(\mathcal{F})$ is not enlarged by allowing terms in $\mathcal{T}(\mathcal{G})$ to be substituted for the variables in the rewrite rules. We stress that this condition is decidable for left-linear $\mathcal{R}$ and $\alpha \in \{\mathrm{s}, \mathrm{nv}, \mathrm{g}\}$ by standard tree automata techniques.

Theorem 3. *Let $\mathcal{R}$ be an orthogonal TRS over a signature $\mathcal{F}$, $\alpha \in \{\mathrm{s}, \mathrm{nv}\}$, and $\mathcal{F} \subseteq \mathcal{G}$ such that $\mathsf{WN}(\mathcal{R}_\alpha, \mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{G}, \mathcal{F})$. If $(\mathcal{R}, \mathcal{F}) \in \mathcal{CBN}_\alpha$ and $\mathcal{R}$ is a CS or $\mathcal{R}_\alpha$ is collapsing then $(\mathcal{R}, \mathcal{G}) \in \mathcal{CBN}_\alpha$.*

Proof. If $(\mathcal{R}, \mathcal{F})$ has external normal forms then the result follows from Theorem 2. So we assume that $(\mathcal{R}, \mathcal{F})$ lacks external normal forms. We also assume that $\mathcal{R} \neq \varnothing$ for otherwise the result is trivial. Suppose to the contrary that $(\mathcal{R}, \mathcal{G}) \notin \mathcal{CBN}_\alpha$. According to Lemma 1 there exists a term $t \in \mathsf{WNR}(\mathcal{R}_\alpha, \mathcal{G})$ without $(\mathcal{R}_\alpha, \mathcal{G})$-needed redex. Lemma 5 (with $\mathcal{R} = \mathcal{S}$) yields a flat redex $\Xi \in \mathcal{T}(\mathcal{F})$. If $\mathcal{R}_\alpha$ is collapsing then we may assume that Ξ is $\mathcal{R}_\alpha$-collapsing. Let t' be the term in $\mathcal{T}(\mathcal{F})$ obtained from t by replacing every $\mathcal{G}\backslash\mathcal{F}$-alien by Ξ. Let P be the set of positions of those aliens. Since t' is reducible, it contains an $(\mathcal{R}_\alpha, \mathcal{F})$-needed redex, say at position q. We show that $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G})$. We consider two cases.

1. Suppose that $q \in P$. Since $t \in \mathsf{WNR}(\mathcal{R}_\alpha, \mathcal{G})$, $t \rightarrow^*_{\mathcal{R}_\alpha,\mathcal{G}} \Delta$ for some redex $\Delta \in \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{G}) \subseteq \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G})$. Since the root symbol of every alien belongs to $\mathcal{G}\backslash\mathcal{F}$, aliens cannot contribute to the creation of Δ and hence we may replace them by arbitrary terms in $\mathcal{T}(\mathcal{G}_\bullet)$ and still obtain a redex that is pattern equal to Δ. We replace the alien at position q by $\bullet$ and every alien at position $p \in P \setminus \{q\}$ by $t'|_p$. This gives $t'[\bullet]_q \rightarrow^*_{\mathcal{R}_\alpha,\mathcal{G}_\bullet} \Delta'$ with $\Delta' \approx \Delta$. Lemma 2 yields $\Delta' \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G})$ and hence $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G})$.
2. Suppose that $q \notin P$. Since Ξ is flat, it follows by orthogonality that q is also a redex position in t. Since t is an $(\mathcal{R}_\alpha, \mathcal{G})$-free term, $t[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G})$. We distinguish two further cases.
   (a) First assume that $\mathcal{R}$ is a CS. Since t is not root-stable, its root symbol must be defined. From Lemma 4 we learn that a ground normal form of $\mathcal{R}$ (and thus of $\mathcal{R}_\alpha$) cannot contain defined symbols. Hence any $(\mathcal{R}_\alpha, \mathcal{G}_\bullet)$-rewrite sequence that normalizes $t[\bullet]_q$ contains a root step and thus $t[\bullet]_q \in \mathsf{WNR}_\bullet(\mathcal{R}_\alpha, \mathcal{G})$. Hence $t[\bullet]_q \rightarrow^*_{\mathcal{R}_\alpha,\mathcal{G}_\bullet} \Delta' \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G})$ for some redex $\Delta' \in \mathcal{T}(\mathcal{G}_\bullet)$. Similar to case (1) above, we replace the $\mathcal{G}\backslash\mathcal{F}$-aliens

in $t[\bullet]_q$ by Ξ. This yields $t'[\bullet]_q \to^*_{\mathcal{R}_\alpha,\mathcal{G}_\bullet} \Delta''$ with $\Delta'' \approx \Delta'$. Lemma 2 yields $\Delta'' \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha,\mathcal{G})$ and hence $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha,\mathcal{G})$.

(b) Next assume that $\mathcal{R}_\alpha$ is collapsing. Because Ξ is a collapsing redex, we have $\Xi \to_{\mathcal{R}_\alpha,\mathcal{G}} t|_p$ for all $p \in P$. Hence $t'[\bullet]_q \to^*_{\mathcal{R}_\alpha,\mathcal{G}_\bullet} t[\bullet]_q$ and thus $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha,\mathcal{G})$.

As $t' \in \mathcal{T}(\mathcal{F})$, we have $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{R}_\alpha,\mathcal{G},\mathcal{F})$ and thus $t'[\bullet]_q \notin \mathsf{WN}_\bullet(\mathcal{R}_\alpha,\mathcal{F})$ by Lemma 3, contradicting the assumption that q is the position of an $(\mathcal{R}_\alpha,\mathcal{F})$-needed redex in t'. □

It can be shown that all conditions in the above theorem are necessary. Below we show the necessity of the $\mathsf{WN}(\mathcal{R}_\alpha,\mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha,\mathcal{G},\mathcal{F})$ condition for noncollapsing CSs $\mathcal{R}_\alpha$. Due to lack of space we omit the other (complicated) counterexamples.

Example 6. Consider the orthogonal noncollapsing CS

$$\mathcal{R} = \left\{ \begin{array}{lcllcl} \mathsf{f}(x,\mathsf{a},\mathsf{b}) & \to & \mathsf{g}(x) & \mathsf{g}(\mathsf{a}) & \to & \mathsf{g}(\mathsf{a}) \\ \mathsf{f}(\mathsf{b},x,\mathsf{a}) & \to & \mathsf{g}(x) & \mathsf{g}(\mathsf{b}) & \to & \mathsf{g}(\mathsf{a}) \\ \mathsf{f}(\mathsf{a},\mathsf{b},x) & \to & \mathsf{g}(x) & \mathsf{h}(x) & \to & \mathsf{i}(x) \\ \mathsf{f}(\mathsf{a},\mathsf{a},\mathsf{a}) & \to & \mathsf{a} & \mathsf{i}(\mathsf{a}) & \to & \mathsf{a} \\ \mathsf{f}(\mathsf{b},\mathsf{b},\mathsf{b}) & \to & \mathsf{a} & \mathsf{i}(\mathsf{b}) & \to & \mathsf{b} \end{array} \right\}$$

over the signature $\mathcal{F}$ consisting of all symbols appearing in the rewrite rules and let $\mathcal{S} = \mathcal{R}_{\mathrm{nv}}$. Let $\mathcal{G} = \mathcal{F} \cup \{\mathsf{c}\}$ with c a constant. Note that $\mathsf{WN}(\mathcal{S},\mathcal{F}) \neq \mathsf{WN}(\mathcal{S},\mathcal{G},\mathcal{F})$ as witnessed by the term $\mathsf{f}(\mathsf{a},\mathsf{a},\mathsf{b})$. With some effort one can check that $(\mathcal{R},\mathcal{F}) \in \mathcal{CBN}$. However, $(\mathcal{R},\mathcal{G}) \notin \mathcal{CBN}$ as $\mathsf{f}(\mathsf{h}(\mathsf{a}),\mathsf{h}(\mathsf{a}),\mathsf{h}(\mathsf{a}))$ lacks $(\mathcal{S},\mathcal{G})$-needed redexes.

We show that Theorem 1 is a special case of Theorem 3 by proving that for $\alpha = \mathrm{s}$ the condition $\mathsf{WN}(\mathcal{R}_\alpha,\mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha,\mathcal{G},\mathcal{F})$ is a consequence of $\mathsf{NF}(\mathcal{R},\mathcal{F}) \neq \varnothing$.

Lemma 6. *Let $\mathcal{R}$ be an eTRS over a signature $\mathcal{F}$. If $\mathsf{NF}(\mathcal{R},\mathcal{F}) \neq \varnothing$ then $\mathsf{WN}(\mathcal{R}_\mathrm{s},\mathcal{F}) = \mathcal{T}(\mathcal{F})$.*

Proof. If $\mathsf{NF}(\mathcal{R},\mathcal{F}) \neq \varnothing$ then there must be a constant $c \in \mathsf{NF}(\mathcal{R},\mathcal{F})$. Define the TRS $\mathcal{R}' = \{l \to c \mid l \to r \in \mathcal{R}\}$ over the signature $\mathcal{F}$. Clearly $\to_{\mathcal{R}'} \subseteq \to_\mathrm{s}$. Consider a precedence (i.e., a well-founded proper order on $\mathcal{F}$) $>$ with $f > c$ for every function symbol $f \in \mathcal{F}$ different from c. The TRS $\mathcal{R}'$ is compatible with the induced recursive path order $>_{\mathrm{rpo}}$ ([3]) and thus terminating. Since $\mathcal{R}_\mathrm{s}$ and $\mathcal{R}'$ have the same normal forms, it follows that $\mathcal{R}_\mathrm{s}$ is weakly normalizing. □

Proof of Theorem 1. Let $\mathcal{R}$ be an orthogonal TRS over a signature $\mathcal{F}$ such that $(\mathcal{R},\mathcal{F}) \in \mathcal{CBN}_\mathrm{s}$. Let $\mathcal{F} \subseteq \mathcal{G}$. We have to show that $(\mathcal{R},\mathcal{G}) \in \mathcal{CBN}_\mathrm{s}$. Since $\mathcal{R}_\mathrm{s}$ is collapsing (if $\mathcal{R} \neq \varnothing$; otherwise $\mathcal{R}$ is a CS), the result follows from Theorem 3 provided that $\mathsf{WN}(\mathcal{R}_\mathrm{s},\mathcal{F}) = \mathsf{WN}(\mathcal{R}_\mathrm{s},\mathcal{G},\mathcal{F})$. From Lemma 6 we obtain $\mathsf{WN}(\mathcal{R}_\mathrm{s},\mathcal{F}) = \mathcal{T}(\mathcal{F})$ and $\mathsf{WN}(\mathcal{R}_\mathrm{s},\mathcal{G},\mathcal{F}) = \mathsf{WN}(\mathcal{R}_\mathrm{s},\mathcal{G}) \cap \mathcal{T}(\mathcal{F}) = \mathcal{T}(\mathcal{G}) \cap \mathcal{T}(\mathcal{F}) = \mathcal{T}(\mathcal{F})$. □

We conclude this section by remarking that we have to use Theorem 3 only once. After adding a single new function symbol we obtain an external normal form and hence we can apply Theorem 2 for the remaining new function symbols.

4 Modularity

The results obtained in the previous section form the basis for the modularity results presented in this section. We first consider disjoint combinations.

Definition 4. *We say that a class $\mathcal{C}$ of TRSs is* modular (for disjoint combinations) *if $(\mathcal{R} \cup \mathcal{R}', \mathcal{F} \cup \mathcal{F}') \in \mathcal{C}$ for all $(\mathcal{R}, \mathcal{F}), (\mathcal{R}', \mathcal{F}') \in \mathcal{C}$ such that $\mathcal{F} \cap \mathcal{F}' = \varnothing$.*

To simplify notation, in the remainder of this section we write $\mathcal{S}$ for $\mathcal{R} \cup \mathcal{R}'$ and $\mathcal{G}$ for $\mathcal{F} \cup \mathcal{F}'$. The condition in Theorem 2 is insufficient for modularity as shown by the following example.

Example 7. Consider the TRS

$$\mathcal{R} = \left\{ \begin{array}{lcl} \mathsf{f}(x, \mathsf{a}, \mathsf{b}) & \to & \mathsf{a} \\ \mathsf{f}(\mathsf{b}, x, \mathsf{a}) & \to & \mathsf{a} \\ \mathsf{f}(\mathsf{a}, \mathsf{b}, x) & \to & \mathsf{a} \end{array} \right\}$$

over the signature $\mathcal{F}$ consisting of all symbols appearing in the rewrite rules and the TRS $\mathcal{R}' = \{\mathsf{g}(x) \to x\}$ over the signature $\mathcal{F}'$ consisting of a constant c in addition to g. Both TRSs have external normal forms and belong to $\mathcal{CBN}_{\mathsf{nv}}$, as one easily shows. Their union does not belong to $\mathcal{CBN}_{\mathsf{nv}}$ as the term $\mathsf{f}(\mathsf{g}(\mathsf{a}), \mathsf{g}(\mathsf{a}), \mathsf{g}(\mathsf{a}))$ has no $(\mathcal{S}_{\mathsf{nv}}, \mathcal{G})$-needed redex:

$$\begin{array}{lllllll} \mathsf{f}(\bullet, \mathsf{g}(\mathsf{a}), \mathsf{g}(\mathsf{a})) & \to_{\mathsf{nv}} & \mathsf{f}(\bullet, \mathsf{a}, \mathsf{g}(\mathsf{a})) & \to_{\mathsf{nv}} & \mathsf{f}(\bullet, \mathsf{a}, \mathsf{b}) & \to_{\mathsf{nv}} & \mathsf{a} \\ \mathsf{f}(\mathsf{g}(\mathsf{a}), \bullet, \mathsf{g}(\mathsf{a})) & \to_{\mathsf{nv}} & \mathsf{f}(\mathsf{b}, \bullet, \mathsf{g}(\mathsf{a})) & \to_{\mathsf{nv}} & \mathsf{f}(\mathsf{b}, \bullet, \mathsf{a}) & \to_{\mathsf{nv}} & \mathsf{a} \\ \mathsf{f}(\mathsf{g}(\mathsf{a}), \mathsf{g}(\mathsf{a}), \bullet) & \to_{\mathsf{nv}} & \mathsf{f}(\mathsf{a}, \mathsf{g}(\mathsf{a}), \bullet) & \to_{\mathsf{nv}} & \mathsf{f}(\mathsf{a}, \mathsf{b}, \bullet) & \to_{\mathsf{nv}} & \mathsf{a} \end{array}$$

If we forbid collapsing rules like $\mathsf{g}(x) \to x$, modularity holds. The following theorem can be proved along the lines of the proof of Theorem 2; because there are no collapsing rules and the eTRSs are left-linear, aliens cannot influence the possibility to perform a rewrite step in the non-alien part of a term.

Theorem 4. *The subclass of $\mathcal{CBN}$ consisting of all noncollapsing left-linear eTRSs with external normal forms is modular.* □

We find it convenient to separate the counterpart of Theorem 3 into two parts. The next two lemmata are used in both proofs. The nontrivial proof of the first one is omitted due to lack of space.

Lemma 7. *Let $(\mathcal{R}, \mathcal{F})$ and $(\mathcal{R}', \mathcal{F}')$ be disjoint TRSs. If $\alpha \in \{\mathrm{s}, \mathrm{nv}\}$ then* $\mathsf{WN}(\mathcal{S}_\alpha, \mathcal{G}, \mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{G}, \mathcal{F})$. □

Lemma 8. *Let $(\mathcal{R}, \mathcal{F})$ and $(\mathcal{R}', \mathcal{F}')$ be disjoint orthogonal TRSs and $\alpha \in \{\mathrm{s}, \mathrm{nv}\}$. If $\mathsf{WN}(\mathcal{S}_\alpha, \mathcal{G}, \mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{F})$ then $\mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G}, \mathcal{F}) = \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{F})$.*

Proof. The previous lemma yields $\mathsf{WN}(\mathcal{R}_\alpha, \mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{G}, \mathcal{F})$. From Lemma 3 we obtain $\mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{F}) = \mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{G}, \mathcal{F})$. Another application of the previous lemma yields the desired $\mathsf{WN}_\bullet(\mathcal{R}_\alpha, \mathcal{F}) = \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G}, \mathcal{F})$. □

Theorem 5. *Let $(\mathcal{R}, \mathcal{F})$ and $(\mathcal{R}', \mathcal{F}')$ be disjoint orthogonal noncollapsing CSs such that $\mathsf{WN}(\mathcal{R}_{\mathrm{nv}}, \mathcal{G}, \mathcal{F}) = \mathsf{WN}(\mathcal{R}_{\mathrm{nv}}, \mathcal{F})$ and $\mathsf{WN}(\mathcal{R}'_{\mathrm{nv}}, \mathcal{G}, \mathcal{F}') = \mathsf{WN}(\mathcal{R}'_{\mathrm{nv}}, \mathcal{F}')$. If $(\mathcal{R}, \mathcal{F}), (\mathcal{R}', \mathcal{F}') \in \mathcal{CBN}_{\mathrm{nv}}$ then $(\mathcal{S}, \mathcal{G}) \in \mathcal{CBN}_{\mathrm{nv}}$.*

Proof. We assume that both $\mathcal{R}$ and $\mathcal{R}'$ are nonempty, for otherwise the result follows from Theorem 3. Suppose to the contrary that $(\mathcal{S}, \mathcal{G}) \notin \mathcal{CBN}_{\mathrm{nv}}$. According to Lemma 1 there exists a term $t \in \mathsf{WNR}(\mathcal{S}_{\mathrm{nv}}, \mathcal{G})$ without $(\mathcal{S}_{\mathrm{nv}}, \mathcal{G})$-needed redex. Assume without loss of generality that $\mathsf{root}(t) \in \mathcal{F}$. Lemma 5 yields a flat redex $\Xi \in \mathcal{T}(\mathcal{F})$. Let t' be the term in $\mathcal{T}(\mathcal{F})$ obtained from t by replacing every $\mathcal{G}\backslash\mathcal{F}$-alien by Ξ. Let P be the set of positions of those aliens. Since t' is reducible, it contains an $(\mathcal{R}_{\mathrm{nv}}, \mathcal{F})$-needed redex, say at position q. We show that $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_{\mathrm{nv}}, \mathcal{G})$. We consider two cases.

1. Suppose that $q \in P$. Since $t \in \mathsf{WNR}(\mathcal{S}_{\mathrm{nv}}, \mathcal{G})$, $t \to^*_{\mathcal{S}_{\mathrm{nv}}, \mathcal{G}} \Delta$ for some redex $\Delta \in \mathsf{WN}(\mathcal{S}_{\mathrm{nv}}, \mathcal{G}) \subseteq \mathsf{WN}_\bullet(\mathcal{S}_{\mathrm{nv}}, \mathcal{G})$. Since $\mathcal{S}_{\mathrm{nv}}$ is noncollapsing and the root symbol of every alien belongs to $\mathcal{G}\backslash\mathcal{F}$, aliens cannot contribute to the creation of Δ and hence we may replace them by arbitrary terms in $\mathcal{T}(\mathcal{G}_\bullet)$ and still obtain a redex that is pattern equal to Δ. We replace the alien at position q by $\bullet$ and every alien at position $p \in P \setminus \{q\}$ by $t'|_p$. This gives $t'[\bullet]_q \to^*_{\mathcal{S}_{\mathrm{nv}}, \mathcal{G}_\bullet} \Delta'$ with $\Delta' \approx \Delta$. Lemma 2 yields $\Delta' \in \mathsf{WN}_\bullet(\mathcal{S}_{\mathrm{nv}}, \mathcal{G})$ and hence $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_{\mathrm{nv}}, \mathcal{G})$.
2. Suppose that $q \notin P$. Since Ξ is flat, it follows by orthogonality that q is also a redex position in t. Since t is an $(\mathcal{S}_{\mathrm{nv}}, \mathcal{G})$-free term, $t[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_{\mathrm{nv}}, \mathcal{G})$. Since t is not root-stable, its root symbol must be defined. From Lemma 4 we learn that a ground normal form of $\mathcal{S}$ (and thus of $\mathcal{S}_{\mathrm{nv}}$) cannot contain defined symbols. Hence any $(\mathcal{S}_{\mathrm{nv}}, \mathcal{G}_\bullet)$-rewrite sequence that normalizes $t[\bullet]_q$ contains a root step and thus $t[\bullet]_q \in \mathsf{WNR}_\bullet(\mathcal{S}_{\mathrm{nv}}, \mathcal{G})$. Hence $t[\bullet]_q \to^*_{\mathcal{S}_{\mathrm{nv}}, \mathcal{G}_\bullet} \Delta \in \mathsf{WN}_\bullet(\mathcal{S}_{\mathrm{nv}}, \mathcal{G})$ for some redex $\Delta \in \mathcal{T}(\mathcal{G}_\bullet)$. We replace the $\mathcal{G}\backslash\mathcal{F}$-aliens in $t[\bullet]_q$ by Ξ. Since $\mathcal{S}$ is noncollapsing, this yields $t'[\bullet]_q \to^*_{\mathcal{S}_{\mathrm{nv}}, \mathcal{G}_\bullet} \Delta'$ with $\Delta' \approx \Delta$. Lemma 2 yields $\Delta' \in \mathsf{WN}_\bullet(\mathcal{S}_{\mathrm{nv}}, \mathcal{G})$ and hence $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_{\mathrm{nv}}, \mathcal{G})$.

As $t' \in \mathcal{T}(\mathcal{F})$, we have $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_{\mathrm{nv}}, \mathcal{G}, \mathcal{F})$ and thus $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{R}_{\mathrm{nv}}, \mathcal{F})$ by Lemmata 7 and 8, contradicting the assumption that q is the position of an $(\mathcal{R}_{\mathrm{nv}}, \mathcal{F})$-needed redex in t'. □

Theorem 6. *Let $(\mathcal{R}, \mathcal{F})$ and $(\mathcal{R}', \mathcal{F}')$ be disjoint orthogonal TRSs and $\alpha \in \{\mathrm{s}, \mathrm{nv}\}$ such that $\mathsf{WN}(\mathcal{R}_\alpha, \mathcal{G}, \mathcal{F}) = \mathsf{WN}(\mathcal{R}_\alpha, \mathcal{F})$ and $\mathsf{WN}(\mathcal{R}'_\alpha, \mathcal{G}, \mathcal{F}') = \mathsf{WN}(\mathcal{R}'_\alpha, \mathcal{F}')$. If $(\mathcal{R}, \mathcal{F}), (\mathcal{R}', \mathcal{F}') \in \mathcal{CBN}_\alpha$ and both $\mathcal{R}_\alpha$ and $\mathcal{R}'_\alpha$ are collapsing then $(\mathcal{S}, \mathcal{G}) \in \mathcal{CBN}_\alpha$.*

Proof. We assume that both $\mathcal{R}$ and $\mathcal{R}'$ are nonempty, for otherwise the result follows from Theorem 3. Suppose to the contrary that $(\mathcal{S}, \mathcal{G}) \notin \mathcal{CBN}_\alpha$. According to Lemma 1 there exists a term $t \in \mathsf{WNR}(\mathcal{S}_\alpha, \mathcal{G})$ without $(\mathcal{S}_\alpha, \mathcal{G})$-needed redex. Assume without loss of generality that $\mathsf{root}(t) \in \mathcal{F}'$. Lemma 5 yields a flat $\mathcal{R}'_\alpha$-collapsing redex $\Xi \in \mathcal{T}(\mathcal{F}')$. Let t' be the term in $\mathcal{T}(\mathcal{F}')$ obtained from

t by replacing every $\mathcal{G}\backslash\mathcal{F}'$-alien by Ξ. Let P be the set of positions of those aliens. Since t' is reducible, it contains an $(\mathcal{R}'_\alpha, \mathcal{F}')$-needed redex, say at position q. We show that $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$. Because Ξ is a collapsing redex, we have $\Xi \to_{\mathcal{R}_\alpha,\mathcal{G}} t|_p$ for all $p \in P$. Hence $t' \to^*_{\mathcal{R}_\alpha,\mathcal{G}_\bullet} t$ and thus, by orthogonality, $t'[\bullet]_q \to^*_{\mathcal{R}_\alpha,\mathcal{G}_\bullet} t[\bullet]_q$. Hence it suffices to show that $t[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$. We distinguish two cases.

1. Suppose that $q \in P$. Since $t \in \mathsf{WNR}(\mathcal{S}_\alpha, \mathcal{G})$, $t \to^*_{\mathcal{S}_\alpha,\mathcal{G}} \Delta$ for some redex $\Delta \in \mathsf{WN}(\mathcal{S}_\alpha, \mathcal{G}) \subseteq \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$. We distinguish two further cases.
   (a) If $t|_q$ is a normal form then it cannot contribute to the creation of Δ and hence by replacing it by $\bullet$ we obtain $t[\bullet]_q \to^*_{\mathcal{S}_\alpha,\mathcal{G}} \Delta'$ with $\Delta \approx \Delta'$. Lemma 2 yields $\Delta' \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$ and thus $t[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$.
   (b) Suppose $t|_q$ is reducible. Because t is a minimal free term, $t|_q$ contains an $(\mathcal{S}_\alpha, \mathcal{G})$-needed redex, say at position q'. So $t|_q[\bullet]_{q'} \notin \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$. In particular, $t|_q[\bullet]_{q'}$ does not $(\mathcal{S}_\alpha, \mathcal{G})$-rewrite to a collapsing redex, for otherwise it would rewrite to a normal form in one extra step. Hence the root symbol of every reduct of $t|_q[\bullet]_{q'}$ belongs to $\mathcal{F}$. Since qq' is not the position of an $(\mathcal{S}_\alpha, \mathcal{G})$-needed redex in t, $t[\bullet]_{qq'} \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$. Since any normalizing $(\mathcal{S}_\alpha, \mathcal{G})$-rewrite sequence must contain a rewrite step at a position above q, we may write $t[\bullet]_{qq'} \to^*_{\mathcal{S}_\alpha,\mathcal{G}} C[\Delta'] \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$ such that Δ' is the first redex above position q. Since $\mathsf{root}(\Delta') \in \mathcal{F}'$, the subterm $t|_q[\bullet]_{q'}$ of $t[\bullet]_{qq'}$ does not contribute to the creation of Δ' and hence $t[\bullet]_q \to^*_{\mathcal{S}_\alpha,\mathcal{G}} C[\Delta'']$ with $\Delta'' \approx \Delta'$. Lemma 2 yields $C[\Delta''] \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$ and thus $t[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$.
2. Suppose that $q \notin P$. Since Ξ is flat, it follows by orthogonality that q is also a redex position in t. Since t is an $(\mathcal{S}_\alpha, \mathcal{G})$-free term, $t[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G})$.

As $t' \in \mathcal{T}(\mathcal{F}')$, we have $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{S}_\alpha, \mathcal{G}, \mathcal{F}')$ and thus $t'[\bullet]_q \in \mathsf{WN}_\bullet(\mathcal{R}'_\alpha, \mathcal{F}')$ by Lemmata 7 and 8, contradicting the assumption that q is the position of an $(\mathcal{R}'_\alpha, \mathcal{F}')$-needed redex in t'. □

It is rather surprising that the presence of collapsing rules helps to achieve modularity; for most properties of TRSs collapsing rules are an obstacle for modularity (see e.g. Middeldorp [14]).

The next result is the modularity counterpart of Theorem 1. It is an easy corollary of the preceding theorem.

Theorem 7. *The subclass of $\mathcal{CBN}_s$ consisting of all orthogonal TRSs $(\mathcal{R}, \mathcal{F})$ such that $\mathsf{NF}(\mathcal{R}, \mathcal{F}) \neq \varnothing$ is modular.*

Klop and Middeldorp [13] showed the related result that strong sequentiality is a modular property of orthogonal TRSs. We already remarked in the preceding section that $\mathcal{CBN}_s$ properly includes all strongly sequential TRSs. Actually, in [13] it is remarked that it is sufficient that the left-hand sides of the two strongly sequential rewrite systems do not share function symbols. One easily verifies that for our modularity results it is sufficient that $\mathcal{R}_\alpha$ and $\mathcal{R}'_\alpha$ do not share function symbols. Actually, we can go a step further by considering so-called *constructor-sharing* combinations. In such combinations the participating systems may share constructors but not defined symbols. It can be shown that the results obtained in this section extend to constructor-sharing combinations, provided we strengthen the requirements in Theorems 4, 5, and 6 by forbidding the presence of *constructor-lifting* rules. A rewrite rule $l \to r$ is called constructor-lifting if $\mathsf{root}(r)$ is a shared constructor. The necessity of this condition is shown in the following examples.

Example 8. Consider the TRS

$$\mathcal{R} = \left\{ \begin{array}{lcl} \mathsf{f}(x,\mathsf{c}(\mathsf{a}),\mathsf{c}(\mathsf{b})) & \to & \mathsf{a} \\ \mathsf{f}(\mathsf{c}(\mathsf{b}),x,\mathsf{c}(\mathsf{a})) & \to & \mathsf{a} \\ \mathsf{f}(\mathsf{c}(\mathsf{a}),\mathsf{c}(\mathsf{b}),x) & \to & \mathsf{a} \end{array} \right\}$$

over the signature $\mathcal{F}$ consisting of all symbols appearing in the rewrite rules and the TRS $\mathcal{R}' = \{\, \mathsf{g}(x) \to \mathsf{c}(x) \,\}$ over the signature $\mathcal{F}'$ consisting of a constant d in addition to g and c. Both TRSs have external normal forms, lack collapsing rules, and belong to $\mathcal{CBN}_{\mathsf{nv}}$. Their union does not belong to $\mathcal{CBN}_{\mathsf{nv}}$ as the term $\mathsf{f}(\mathsf{g}(\mathsf{a}),\mathsf{g}(\mathsf{a}),\mathsf{g}(\mathsf{a}))$ has no $(\mathcal{S}_{\mathsf{nv}},\mathcal{G})$-needed redex. Note that $\mathcal{R}$ and $\mathcal{R}'$ share the constructor c and hence $\mathsf{g}(x) \to \mathsf{c}(x)$ is constructor-lifting.

A simple modification of the above example shows the necessity of forbidding constructor-lifting rules in Theorem 5 even if we require that the constituent CSs lack external normal forms.

Example 9. Consider the TRSs

$$\mathcal{R} = \left\{ \begin{array}{lclclcl} \mathsf{f}(x,\mathsf{a},\mathsf{b}) & \to & \mathsf{c}(\mathsf{g}(x)) & \quad & \mathsf{g}(x) & \to & \mathsf{g}(\mathsf{a}) \\ \mathsf{f}(\mathsf{b},x,\mathsf{a}) & \to & \mathsf{c}(\mathsf{g}(x)) & & \mathsf{h}(x) & \to & x \\ \mathsf{f}(\mathsf{a},\mathsf{b},x) & \to & \mathsf{c}(\mathsf{g}(x)) & & & & \end{array} \right\}$$

and

$$\mathcal{R}' = \left\{ \begin{array}{rcl} \mathsf{i}(\mathsf{a}) & \to & \mathsf{a} \\ \mathsf{i}(\mathsf{c}(x)) & \to & x \end{array} \right\}$$

over the signatures $\mathcal{F}$ and $\mathcal{F}'$ consisting of function symbols that appear in their respective rewrite rules. The two TRSs are obviously collapsing and share the constructors a and c. One easily verifies that both TRSs belong to $\mathcal{CBN}_{\mathsf{nv}}$ and that $\mathsf{WN}(\mathcal{R}_{\mathsf{nv}},\mathcal{G},\mathcal{F}) = \mathsf{WN}(\mathcal{R}_{\mathsf{nv}},\mathcal{F})$ and $\mathsf{WN}(\mathcal{R}_{\mathsf{nv}},\mathcal{G},\mathcal{F}') = \mathcal{T}(\mathcal{F}') = \mathsf{WN}(\mathcal{R}'_{\mathsf{nv}},\mathcal{F}')$. However, the union of the two TRSs does not belong to $\mathcal{CBN}_{\mathsf{nv}}$ as the term $\mathsf{i}(\mathsf{f}(\Delta,\Delta,\Delta))$ with any collapsing redex Δ has no $(\mathcal{S}_{\mathsf{nv}},\mathcal{G})$-needed redex.

References

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

2. H. Comon. Sequentiality, monadic second-order logic and tree automata. *Information and Computation*, 157:25–51, 2000.
3. N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.
4. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier, 1990.
5. I. Durand. Bounded, strongly sequential and forward-branching term rewriting systems. *Journal of Symbolic Computation*, 18:319–352, 1994.
6. I. Durand and A. Middeldorp. Decidable decidable call by need computations in term rewriting (extended abstract). In *Proceedings of the 14th International Conference on Automated Deduction*, volume 1249 of *LNAI*, pages 4–18, 1997.
7. I. Durand and A. Middeldorp. On the complexity of deciding call-by-need. Technical Report 1194-98, LaBRI, Université de Bordeaux I, 1998.
8. B. Gramlich. *Termination and Confluence Properties of Structured Rewrite Systems*. PhD thesis, Universität Kaiserslautern, 1996.
9. G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems, I and II. In *Computational Logic, Essays in Honor of Alan Robinson*, pages 396–443. The MIT Press, 1991. Original version: Report 359, Inria, 1979.
10. F. Jacquemard. Decidable approximations of term rewriting systems. In *Proceedings of the 7th International Conference on Rewriting Techniques and Applications*, volume 1103 of *LNCS*, pages 362–376, 1996.
11. R. Kennaway, J.W. Klop, R. Sleep, and F.-J. de Vries. Comparing curried and uncurried rewriting. *Journal of Symbolic Computation*, 21(1):15–39, 1996.
12. J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science, Vol. 2*, pages 1–116. Oxford University Press, 1992.
13. J.W. Klop and A. Middeldorp. Sequentiality in orthogonal term rewriting systems. *Journal of Symbolic Computation*, 12:161–195, 1991.
14. A. Middeldorp. *Modular Properties of Term Rewriting Systems*. PhD thesis, Vrije Universiteit, Amsterdam, 1990.
15. T. Nagaya and Y. Toyama. Decidability for left-linear growing term rewriting systems. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications*, volume 1631 of *LNCS*, pages 256–270, 1999.
16. E. Ohlebusch. *Modular Properties of Composable Term Rewriting Systems*. PhD thesis, Universität Bielefeld, 1994.
17. M. Oyamaguchi. NV-sequentiality: A decidable condition for call-by-need computations in term rewriting systems. *SIAM Journal on Computation*, 22:114–135, 1993.
18. R. Strandh. Classes of equational programs that compile into efficient machine code. In *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*, volume 355 of *LNCS*, pages 449–461, 1989.
19. T. Takai, Y. Kaji, and H. Seki. Right-linear finite path overlapping term rewriting systems effectively preserve recognizability. In *Proceedings of the 11th International Conference on Rewriting Techniques and Applications*, volume 1833 of *LNCS*, pages 246–260, 2000.
20. Y. Toyama. Strong sequentiality of left-linear overlapping term rewriting systems. In *Proceedings of the 7th IEEE Annual Symposium on Logic in Computer Science*, pages 274–284, 1992.
21. Y. Toyama, S. Smetsers, M. van Eekelen, and R. Plasmeijer. The functional strategy and transitive term rewriting systems. In *Term Graph Rewriting: Theory and Practice*, pages 61–75. Wiley, 1993.

New Completeness Results for Lazy Conditional Narrowing

Mircea Marin and Aart Middeldorp

Institute of Information Sciences and Electronics
University of Tsukuba, Tsukuba 305-8573, Japan
{mmarin,ami}@score.is.tsukuba.ac.jp

1 Lazy Conditional Narrowing

Narrowing was originally invented as a general method for solving unification problems in equational theories that are presented by confluent term rewriting systems (TRSs for short). More recently, narrowing was proposed as the computational mechanism of functional-logic programming languages and several new completeness results concerning the completeness of various narrowing strategies and calculi have been obtained in the past few years. Here completeness means that for every solution to a given goal a solution that is at least as general is computed by the narrowing strategy. Since narrowing is a complicated operation, numerous calculi consisting of a small number of more elementary inference rules that simulate narrowing have been proposed.

Completeness issues for the lazy narrowing calculus LNC have been extensively studied in [3] and [2]. The main result of [3] is the completeness of LNC with leftmost selection for arbitrary confluent TRSs and normalized solutions. In [2] restrictions on the participating TRSs and solutions are presented which guarantee that all non-determinism due to the choice of inference rule of LNC is removed.

In this paper we consider the lazy conditional narrowing calculus LCNC of [4]. LCNC is the extension of LNC to conditional term rewrite systems (CTRSs for short). The extension is motivated by the observation that CTRSs are much more expressive than unconditional TRSs for describing interesting problems in a natural and concise way. However, the additional expressive power raises two problems: (1) confluence is insufficient to guarantee the completeness with respect to normalized solutions, and (2) the search space increases dramatically because the conditions of the applied rewrite rule are added to the current goal.

In [4] several completeness results for LCNC are presented. The only result which does not assume some kind of termination assumption does not permit extra variables in the conditions and right-hand sides of the rewrite rules. In this paper we show the completeness of LCNC with leftmost selection for the class of (confluent) *deterministic* oriented CTRSs. Determinism was introduced by Ganzinger [1] and has proved to be very useful for the study of the (unique) termination behavior of well-moded Horn clause programs (cf. [5]).

We consider goals G consisting of *unoriented* equations $s \approx t$ and *oriented* equations $s \rhd t$. An *oriented* CTRS $\mathcal{R}$ is a set of rewrite rules of the form

$l \to r \Leftarrow c$ where c is a goal consisting of oriented equations. The induced rewrite relation is denoted by $\to_{\mathcal{R}}$. A substitution θ is a solution of a goal $G = e_1, \ldots, e_n$ if $s\theta \leftrightarrow^*_{\mathcal{R}} t\theta$ if $e_i = s \approx t$ and $s\theta \to^*_{\mathcal{R}} t\theta$ if $e_i = s \rhd t$, for all $i \in \{1, \ldots, n\}$. For confluent CTRSs $\mathcal{R}$ the relations $\leftrightarrow^*_{\mathcal{R}}$ and $\downarrow^*_{\mathcal{R}}$ coincide and hence unoriented equations can be viewed as *joinability* statements.

The lazy conditional narrowing calculus LCNC$_\ell$ consist of the following inference rules:

[o] *outermost narrowing*

$$\frac{f(s_1, \ldots, s_n) \simeq t, G}{s_1 \rhd l_1, \ldots, s_n \rhd l_n, c, r \simeq t, G} \qquad \simeq \in \{\approx, \rhd, \backsimeq\}$$

if $f(l_1, \ldots, l_n) \to r \Leftarrow c$ is a fresh variant of a rewrite rule in $\mathcal{R}$,

[i] *imitation*

$$\frac{f(s_1, \ldots, s_n) \simeq x, G}{(s_1 \simeq x_1, \ldots, s_n \simeq x_n, G)\theta} \qquad \simeq \in \{\approx, \rhd, \backsimeq, \lhd\}$$

if $\theta = \{x \mapsto f(x_1, \ldots, x_n)\}$ with $x_1, \ldots, x_n$ fresh variables,

[d] *decomposition*

$$\frac{f(s_1, \ldots, s_n) \simeq f(t_1, \ldots, t_n), G}{s_1 \simeq t_1, \ldots, s_n \simeq t_n, G} \qquad \simeq \in \{\approx, \rhd\},$$

[v] *variable elimination*

$$\frac{s \simeq x, G}{G\theta} \qquad \simeq \in \{\approx, \rhd, \backsimeq, \lhd\}$$

if $x \notin \mathcal{V}\text{ar}(s)$ and $\theta = \{x \mapsto s\}$,

[t] *removal of trivial equations*

$$\frac{s \simeq s, G}{G} \qquad \simeq \in \{\approx, \rhd\}.$$

In the above rules we use $s \backsimeq t$ to denote the equation $t \approx s$ and $s \lhd t$ to denote the equation $t \rhd s$.

Compared to the definition of LCNC in [4], in LCNC$_\ell$ leftmost selection is built-in. Another difference is that the *parameter-passing equations* $s_1 \rhd l_1, \ldots, s_n \rhd l_n$ created by the outermost narrowing rule [o] are regarded as oriented equations. This is in line with the earlier completeness proofs for LNC and LCNC [2–4]. Furthermore, the conditions c are placed before $r \simeq t$ whereas in [4] they are placed after $r \simeq t$. The relative locations of c and $r \simeq t$ are irrelevant for the completeness results reported in that paper, but our new completeness proofs do not go through if we stick to the order in [4]. Finally, the removal of trivial equations rule [t] is usually restricted to equations between identical variables. The variation adopted here allows us to shorten some trivial derivations.

If G_1 and G_2 are the upper and lower goal in the inference rule $[\alpha]$, we write $G_1 \Rightarrow_{[\alpha]} G_2$. The applied rewrite rule or substitution may be supplied as subscript, that is, we write things like $G_1 \Rightarrow_{[o], l \to r \Leftarrow c} G_2$ and $G_1 \Rightarrow_{[i], \theta} G_2$. A finite LCNC$_\ell$-derivation $G_1 \Rightarrow_{\theta_1} \cdots \Rightarrow_{\theta_{n-1}} G_n$ may be abbreviated to $G_1 \Rightarrow_\theta^* G_n$ where θ is the composition $\theta_1 \cdots \theta_{n-1}$ of the substitutions $\theta_1, \ldots, \theta_{n-1}$ computed along its steps. An LCNC$_\ell$-refutation is an LCNC$_\ell$-derivation ending in the empty goal $\square$.

2 Soundness and Completeness

Soundness of LCNC$_\ell$ is not difficult to prove.

Theorem 1. *Let $\mathcal{R}$ be an arbitrary CTRS and G a goal. If $G \Rightarrow_\theta^* \square$ then $\theta\restriction_{\mathcal{V}\mathrm{ar}(G)}$ is a solution of G.* $\square$

Let X be a finite set of variables. A goal $G = e_1, \ldots, e_n$ is called X-*deterministic* if $\mathcal{V}\mathrm{ar}(s_i) \subseteq X \cup \bigcup_{j=1}^{i-1} \mathcal{V}\mathrm{ar}(e_j)$ if $e_i = s_i \rhd t_i$ and $\mathcal{V}\mathrm{ar}(e_i) \subseteq X \cup \bigcup_{j=1}^{i-1} \mathcal{V}\mathrm{ar}(e_j)$ if $e_i = s_i \approx t_i$, for all $1 \leqslant i \leqslant n$. An oriented CTRS $\mathcal{R}$ is called deterministic if and only if $\mathcal{V}\mathrm{ar}(r) \subseteq \mathcal{V}\mathrm{ar}(l, c)$ and the conditions c of every rewrite rule $l \to r \Leftarrow c$ in $\mathcal{R}$ are $\mathcal{V}\mathrm{ar}(l)$-deterministic.

The following oriented CTRS $\mathcal{R}$ is a natural encoding of the efficient computation of Fibonacci numbers:

$$\begin{array}{rcl@{\qquad}rcl}
0 + y & \to & y & \mathsf{nth}(0, \langle y, z\rangle) & \to & y \\
\mathsf{s}(x) + y & \to & \mathsf{s}(x + y) & \mathsf{nth}(\mathsf{s}(x), \langle y, z\rangle) & \to & \mathsf{nth}(x, z) \\
\mathsf{fib}(0) & \to & \langle 0, \mathsf{s}(0)\rangle & & & \\
\mathsf{fib}(\mathsf{s}(x)) & \to & \langle z, y + z\rangle \Leftarrow \mathsf{fib}(x) \rhd \langle y, z\rangle & & &
\end{array}$$

Note that $\mathcal{R}$ is deterministic. It is also terminating. However, when we change the oriented condition $\mathsf{fib}(x) \rhd \langle y, z\rangle$ into an unoriented condition $\mathsf{fib}(x) \approx \langle y, z\rangle$ then we lose termination: Because $\mathsf{fib}(\mathsf{s}(0)) \to^+ \langle \mathsf{s}(0), \mathsf{s}(0)\rangle$ we have $\mathsf{fib}(0) \downarrow \langle 0, \mathsf{nth}(0, \mathsf{fib}(\mathsf{s}(0)))\rangle$ and thus

$$\mathsf{fib}(\mathsf{s}(0)) \to \langle \mathsf{nth}(0, \mathsf{fib}(\mathsf{s}(0))), 0 + \mathsf{nth}(0, \mathsf{fib}(\mathsf{s}(0)))\rangle$$

which gives rise to an infinite rewrite sequence. As a consequence, the completeness results for LCNC presented in [4], which deal only with join CTRSs, do not apply to this CTRS. The completeness result presented below does not have this problem.

Theorem 2. *Let $\mathcal{R}$ be a confluent oriented deterministic CTRS and let G be an X-deterministic goal. If θ is a solution of G such that $\theta\restriction_X$ is normalized then $G \Rightarrow_{\theta'}^* \square$ for some substitution θ' with $\theta' \leqslant \theta$ $[\mathcal{V}\mathrm{ar}(G)]$.* $\square$

The proof is based on a transformation of rewrite proofs of $G\theta$. The crucial insight is that determinism permits one to identify suitable sets of variables X_1, $X_2, \ldots$ such that each intermediate goal G_i in the refutation $G \Rightarrow G_1 \Rightarrow G_2 \Rightarrow \cdots \Rightarrow \square$ that is being constructed is X_i-deterministic.

3 Refinements

Completeness with respect to normalized solutions does not imply that all solutions computed by LCNC$_\ell$ are normalized. This is illustrated in the following example.

Example 1. Consider the confluent TRS $\mathcal{R} = \{\mathsf{f}(\mathsf{a}) \to \mathsf{b}, \mathsf{g}(\mathsf{b}) \to \mathsf{c}\}$ and the goal $G = x \approx \mathsf{f}(y), \mathsf{g}(x) \approx \mathsf{c}, \mathsf{f}(y) \approx \mathsf{b}$. The following LCNC$_\ell$-refutation computes the non-normalized solution $\theta = \{x \mapsto \mathsf{f}(\mathsf{a}), y \mapsto \mathsf{a}\}$:

$$\begin{aligned} G &\Rightarrow_{[\mathsf{v}], \{x \mapsto \mathsf{f}(y)\}} \mathsf{g}(\mathsf{f}(y)) \approx \mathsf{c}, \mathsf{f}(y) \approx \mathsf{b} \Rightarrow_{[\mathsf{o}], \mathsf{g}(\mathsf{b}) \to \mathsf{c}} \mathsf{f}(y) \rhd \mathsf{b}, \mathsf{c} \approx \mathsf{c}, \mathsf{f}(y) \approx \mathsf{b} \\ &\Rightarrow_{[\mathsf{o}], \mathsf{f}(\mathsf{a}) \to \mathsf{b}} y \rhd \mathsf{a}, \mathsf{b} \rhd \mathsf{b}, \mathsf{c} \approx \mathsf{c}, \mathsf{f}(y) \approx \mathsf{b} \Rightarrow_{[\mathsf{v}], \{y \mapsto \mathsf{a}\}} \mathsf{b} \rhd \mathsf{b}, \mathsf{c} \approx \mathsf{c}, \mathsf{f}(\mathsf{a}) \approx \mathsf{b} \\ &\Rightarrow^2_{[\mathsf{t}]} \mathsf{f}(\mathsf{a}) \approx \mathsf{b} \Rightarrow_{[\mathsf{o}], \mathsf{f}(\mathsf{a}) \to \mathsf{b}} \mathsf{a} \rhd \mathsf{a}, \mathsf{b} \approx \mathsf{b} \Rightarrow^2_{[\mathsf{t}]} \square \end{aligned}$$

Substitution θ can be normalized to $\theta' = \{x \mapsto \mathsf{b}, y \mapsto \mathsf{a}\}$. Since θ' is also a computed answer of G:

$$\begin{aligned} G &\Rightarrow_{[\mathsf{o}], \mathsf{f}(\mathsf{a}) \to \mathsf{b}} y \rhd \mathsf{a}, x \approx \mathsf{b}, \mathsf{g}(x) \approx \mathsf{c}, \mathsf{f}(y) \approx \mathsf{b} \\ &\Rightarrow_{[\mathsf{v}], \{y \mapsto \mathsf{a}\}} x \approx \mathsf{b}, \mathsf{g}(x) \approx \mathsf{c}, \mathsf{f}(\mathsf{a}) \approx \mathsf{b} \Rightarrow_{[\mathsf{v}], \{x \mapsto \mathsf{b}\}} \mathsf{g}(\mathsf{b}) \approx \mathsf{c}, \mathsf{f}(\mathsf{a}) \approx \mathsf{b} \\ &\Rightarrow_{[\mathsf{o}], \mathsf{g}(\mathsf{b}) \to \mathsf{c}} \mathsf{b} \rhd \mathsf{b}, \mathsf{c} \approx \mathsf{c}, \mathsf{f}(\mathsf{a}) \approx \mathsf{b} \Rightarrow^2_{[\mathsf{t}]} \mathsf{f}(\mathsf{a}) \approx \mathsf{b} \\ &\Rightarrow_{[\mathsf{o}], \mathsf{f}(\mathsf{a}) \to \mathsf{b}} \mathsf{a} \rhd \mathsf{a}, \mathsf{b} \approx \mathsf{b} \Rightarrow^2_{[\mathsf{t}]} \square, \end{aligned}$$

there is no need to compute the non-normalized solution θ.

We propose a simple marking strategy to avoid computing non-normalized solutions. Before presenting the formal details, we illustrate how this strategy avoids the first refutation in the previous example. We want to compute normalized bindings $x\theta$ and $y\theta$ for the variables x and y. Therefore, we mark the occurrences of x and y in G with the marker $\dagger$. Whenever a marked variable is instantiated we mark the root symbol of the instantiation. So the first step becomes

$$x^\dagger \approx \mathsf{f}(y^\dagger), \mathsf{g}(x^\dagger) \approx \mathsf{c}, \mathsf{f}(y^\dagger) \approx \mathsf{b} \Rightarrow_{[\mathsf{v}], \{x^\dagger \mapsto \mathsf{f}^\dagger(y^\dagger)\}} \mathsf{g}(\mathsf{f}^\dagger(y^\dagger)) \approx \mathsf{c}, \mathsf{f}(y^\dagger) \approx \mathsf{b}$$

Note that only the first occurrence of f in the new goal is marked. The next step is the same as before since the marker is not involved:

$$\mathsf{g}(\mathsf{f}^\dagger(y^\dagger)) \approx \mathsf{c}, \mathsf{f}(y^\dagger) \approx \mathsf{b} \Rightarrow_{[\mathsf{o}], \mathsf{g}(\mathsf{b}) \to \mathsf{c}} \mathsf{f}^\dagger(y^\dagger) \rhd \mathsf{b}, \mathsf{c} \approx \mathsf{c}, \mathsf{f}(y^\dagger) \approx \mathsf{b}$$

At this point we do *not* perform the [o] step since the marker on f signals that any instantiation $\mathsf{f}(y\theta)$ of $\mathsf{f}(y)$ should be normalized, but then $\mathsf{f}(y\theta)$ cannot be rewritten to b. Since there are no other applicable inference rules, the derivation ends in failure after just two steps.

The introduction and propagation of the markers is achieved by modifying the inference rules [i], [d], [v], and [t] of LCNC$_\ell$. In the following $t^\dagger$ denotes the term obtained from t by marking its root symbol, provided the root symbol of t unmarked. Otherwise $t^\dagger = t$.

[i] *imitation*

$$\frac{f(s_1,\dots,s_n) \simeq x, G}{(s_1 \simeq x_1,\dots,s_n \simeq x_n, G)\theta} \qquad \frac{f(s_1,\dots,s_n) \simeq x^\dagger, G}{(s_1\theta')^\dagger \simeq x_1^\dagger,\dots,(s_n\theta')^\dagger \simeq x_n^\dagger, G\theta'}$$

with $\simeq\, \in \{\approx, \rhd, \backsimeq, \lhd\}$, $\theta = \{x \mapsto f(x_1,\dots,x_n), x^\dagger \mapsto f^\dagger(x_1,\dots,x_n)\}$, $\theta' = \{x, x^\dagger \mapsto f^\dagger(x_1,\dots,x_n)\}$, and $x_1,\dots,x_n$ fresh variables,

[d] *decomposition*

$$\frac{f(s_1,\dots,s_n) \simeq f(t_1,\dots,t_n), G}{s_1 \simeq t_1,\dots,s_n \simeq t_n, G} \qquad \frac{f^\dagger(s_1,\dots,s_n) \simeq f(t_1,\dots,t_n), G}{s_1^\dagger \simeq t_1,\dots,s_n^\dagger \simeq t_n, G}$$

$$\frac{f(s_1,\dots,s_n) \simeq f^\dagger(t_1,\dots,t_n), G}{s_1 \simeq t_1^\dagger,\dots,s_n \simeq t_n^\dagger, G} \qquad \frac{f^\dagger(s_1,\dots,s_n) \simeq f^\dagger(t_1,\dots,t_n), G}{s_1^\dagger \simeq t_1^\dagger,\dots,s_n^\dagger \simeq t_n^\dagger, G}$$

for $\simeq\, \in \{\approx, \rhd\}$,

[v] *variable elimination*

$$\frac{s \rhd x, G}{G\theta} \quad \frac{s \rhd x^\dagger, G}{G\theta'} \quad \frac{s \simeq x}{G\theta'} \quad \frac{s \simeq x^\dagger}{G\theta'} \qquad \simeq\, \in \{\approx, \backsimeq, \lhd\} \qquad x \notin \mathcal{V}\mathrm{ar}(s)$$

with $\theta = \{x \mapsto s, x^\dagger \mapsto s^\dagger\}$ and $\theta' = \{x, x^\dagger \mapsto s^\dagger\}$,

[t] *removal of trivial equations*

$$\frac{s \simeq t, G}{G} \qquad \simeq\, \in \{\approx, \rhd\},$$

if s and t are identical after removing all marks.

Note that the imitation rule [i] is not applicable if the root symbol of the non-variable side of the equation is marked.

From the completeness proof of LCNC$_\ell$ we obtain several insights to make the calculus more deterministic. Furthermore, and similar to the refinements developed in [2] for the unconditional case, we succeeded in removing almost all non-determinism due to the choice of the inference rule by imposing further syntactic conditions on the participating CTRSs and restricting the set of solutions for which completeness needs to be established.

References

1. H. Ganzinger. Order-sorted completion: The many-sorted way. *Theoretical Computer Science*, 89:3–32, 1991.
2. A. Middeldorp and S. Okui. A deterministic lazy narrowing calculus. *Journal of Symbolic Computation*, 25(6):733–757, 1998.
3. A. Middeldorp, S. Okui, and T. Ida. Lazy narrowing: Strong completeness and eager variable elimination. *Theoretical Computer Science*, 167(1,2):95–130, 1996.
4. A. Middeldorp, T. Suzuki, and M. Hamada. Complete selection functions for a lazy conditional narrowing calculus. *Journal of Functional and Logic Programming*, 2002(3), March 2002.
5. E. Ohlebusch. Termination of logic programs: Transformational methods revisited. *Applicable Algebra in Engineering, Communication and Computing*, 12:73–116, 2001.

Open CFLP: An Open System for Collaborative Constraint Functional Logic Programming

Norio Kobayashi[†] Mircea Marin[‡] Tetsuo Ida[‡] Zhanbin Che[‡]

[†] Doctoral Program in Engineering, University of Tsukuba
[‡] Institute of Information Sciences and Electronics, University of Tsukuba
Tennoudai, Tsukuba 305-8573, Japan
{nori,mmarin,ida,czb}@score.is.tsukuba.ac.jp

Abstract

We will discuss a collaborative constraint functional logic programming system, called Open CFLP, which solves equations by collaboration between distributed constraint solvers in an open environment. This system enables to solve equations in a higher-order functional logic programming style and to exploit the constraint solving service providers without knowing their locations and implementation details. We mainly describe how the system works with an example and its architecture.

1 Introduction

An open environment, such as the Internet, promotes the development of applications which can take advantage of the huge variety of services available online. They are of particular relevance for collaborative constraint solving, a programming style which integrates the constraint solving capabilities of various constraint solvers in a unified framework: the system can configure itself dynamically by accessing the constraint solving services deployed in the open environment.

In this paper, we describe the design and implementation of a system which solves constraints described as a set of equations by a collaboration between constraint solving services in an open environment.

2 Example

We start with a small example to illustrate how Open CFLP solves a problem. We consider the following problem:

$$ty'(t) + y(t) = 40t/3, \quad y(1) = -10,$$
$$t^2y''(t) + ty'(t) - y(t) = 10, \quad y(T) = 4$$

where y and T are variables. First, we write a functional logic program as a conditional pattern rewrite system [4] in a Mathematica notebook [5]. We use map and assume that map is not a built-in function in Open CFLP.

```
R =FLPProgram[
     {map[λ[{t},f[t]],{}] → {},
      map[λ[{t},f[t]],[H | T]] →
        [f[H] | map[λ[{t},f[t]],T]]},
     Signature →
      {DefinedSymbols →
        {map : (ℂ → ℂ) × TyList[ℂ]
          → TyList[ℂ]}}]
```

Next we specify the goal to be solved. A goal is an existentially quantified conjunction of equational formulas. The goal for this example can be defined as follows with type-annotated variables y and T:

```
G =exists[{y : ℂ → ℂ, T : ℂ},
    {λ[{t},t y'[t] + y[t]] == λ[{t},40 t/3]
     λ[{t},t² y''[t] + t y'[t] - y[t]]
     == λ[{t},10],
    map[λ[{t},y[t]],{1,T}] == {-10,4}}]
```

We now have to find solvers in the open environment. Here let us assume that the necessary solvers are on the Internet. Our computing model is that we are not allowed to download the solvers, but are allowed only to use the service of the solvers. In Open CFLP, solving services are named by URIs, and the URIs can be found by keywords that characterize their services. Note that we do not treat names of each solver. First we find a service of solving equations over the domain defined by a higher-order functional logic program.

```
HOLNURIs =LookupService["higher-order
            functional logic programming"]
```

LookupService returns a list of URIs. Each URI denotes a specification of a constraint solving service. The user can browse the specifications associated with a given URI u by calling:

```
HOLNSpec = GetServiceSpec[u]
```

where HOLNSpec is a list of specifications such as configuration options and comments. In this way, the user can choose the URI, say HOLNURI, which provides the best description for his needed service. The call

```
aHOLN = FindSolver[HOLNURI]
```

returns an entity which we call *elementary collaborative*. An elementary collaborative is a collection of basic solvers that satisfy the descriptions associated with the argument of FindSolver. The reason why an elementary collaborative is implemented as a collection of solvers is to solve goals concurrently with its solvers. Elementary collaboratives can be configured by calling ConfigSolver. For example,

```
aHOLN = ConfigSolver[aHOLN,
            {Prog → R, Calculus → "HOLN"}]
```

configures the elementary collaborative aHOLN with the functional logic program stored in R and with the higher-order lazy narrowing calculus HOLN.

Similarly, we can find elementary collaboratives aDeriv, aPolyn and aLinear for solving systems of differential equations, polynomials and linear equations respectively.

Next, we program a collaborative. For this example, we define a collaborative in which the elementary collaboratives aHOLN, aDeriv, aPolyn and aLinear collaborate as follows:

```
aCollabo = NewCollaborative[repeat[
      seq[aHOLN, aDeriv, aPolyn, aLinear]]]
```

seq[aHOLN, aDeriv, aPolyn, aLinear] is a collaborative which applies the elementary collaboratives aHOLN, aDeriv, aPolyn and aLinear sequentially, from left to right. Furthermore, aCollabo applies repeatedly the sequential collaborative mentioned above to the goal until it reaches a fixed-point.

Finally, we apply the goal G to the collaborative aCollabo:

```
ApplyCollaborative[aCollabo, {G}]
```

and obtain the following solutions.

$$\{\{y \mapsto \lambda[\{t\}, -10 + \frac{20(-1+t^2)}{3t}], T \mapsto -\frac{2}{5}\},$$
$$\{y \mapsto \lambda[\{t\}, -10 + \frac{20(-1+t^2)}{3t}], T \mapsto \frac{5}{2}\}\}$$

3 System Architecture

Figure 1 shows the architecture of Open CFLP. Open CFLP consists of five kinds of components: *frontend*, *coordinator*, *object wrapper for solver*, *service repository* and *broker*.

The frontend is a user interface implemented with Mathematica notebook which allows the user to input Open CFLP problems. The coordinator interprets collaboratives and manages the collaboration among solvers.

Functionally, the object wrappers, service repository and broker are grouped together to form a framework called MAXCOR (MAth eXchange for CORBA). MAXCOR realizes transparent communication between solvers in an open environment. Solvers may be implemented in different languages and may use

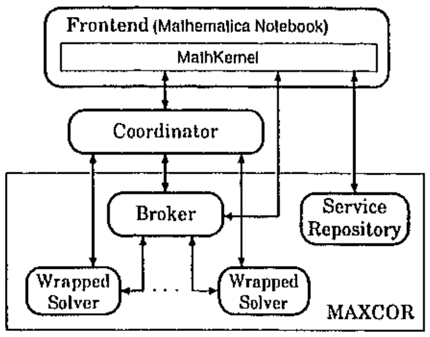


Figure 1: The architecture of Open CFLP

proprietary data formats. MAXCOR has to solve this heterogeneity problem by adopting a standard for communication, and by translating between proprietary data and a standard data format. In order to solve this problem, we employ CORBA [1] as the communication protocol, and MathML [3] and DOM/Value mapping [2] as the common data format for representing mathematical expressions. The MAXCOR components—object wrappers, service repository and broker—are implemented as CORBA objects. Each solver is wrapped by a corresponding object wrapper which translates between the proprietary data format of the solver and the common data format. Object wrappers implement a common interface.

In Open CFLP, the identification of solvers should be location independent. Solvers are found with a discovery service provided by a service repository and a broker. The service repository provides two operations: `LookupService` and `GetServiceSpec`, and the broker provides the operation `FindSolver`. These operations were described in Section 2.

Finally, we summarize the properties of Open CFLP obtained by MAXCOR:

- portability—language and platform independence among solvers,
- scalability and modularity—solvers implemented as CORBA objects,
- data and operation interoperability—the common IDL definition which includes DOM/Value mapping and MathML, and
- location independence of solvers—the discovery service.

4 Conclusions

We have described an open system for constraint functional logic programming. The system solves equations by a collaboration among a functional logic solver based on higher-order lazy narrowing calculus and specialized solvers which implement methods over several domains. Those solvers are located in an open environment. Our system achieves transparent access to solvers and relies on global standards for communication and data representation.

References

[1] The Common Object Request Broker; Architecture and Specification, Editional Revision: CORBA 2.6, Object Management Group, 2002.

[2] XML DOM: DOM/Value Mapping Specification, Object Management Group, 2001.

[3] `http://www.w3.org/Math`.

[4] M. Marin, T. Ida, and T. Suzuki. "Higher-Order Lazy Narrowing Calculus: A Solver for Higher-Order Equations." In Proc. of the 8th Int'l Conf. Computer Aided Systems (Euro-CAST 2001), LNCS, vol. 2178, pp.478–493, Springer-Verlag, 2001.

[5] S. Wolfram. The Mathematica Book, 4th Edition, Wolfram Media Inc. Champaign, Illinois, USA, and Cambridge University Press, 1999.

[6] T. Ida, N. Kobayashi, and M. Marin. "An Open Environment for Cooperative Scientific Problem Solving." In Proc. of the 4th Int'l Mathematica Symposium (IMS 2001), pp. 71–78, Chiba, Japan, 2001.

```
R =FLPProgram[
     {map[λ[{t}, f[t]], {}] → {},
      map[λ[{t}, f[t]], [H | T]] → [f[H] | map[λ[{t}, f[t]], T]]},
     Signature →
      {DefinedSymbols → {map : (ℂ → ℂ) × TyList[ℂ] → TyList[ℂ]}}]
```

Next we specify the goal to be solved. A goal is an existentially quantified conjunction of equational formulas. The goal for this example can be defined as follows with type-annotated variables y and T:

```
G =exists[{y : ℂ → ℂ, T : ℂ},
    {λ[{t}, t y'[t] + y[t]] == λ[{t}, 40 t/3],
     λ[{t}, t^2 y''[t] + t y'[t] - y[t]] == λ[{t}, 10],
    map[λ[{t}, y[t]], {1, T}] == {-10, 4}}]
```

We now have to find solvers in the open environment. Here let us assume that the necessary solvers are on the Internet. Our computing model is that we are not allowed to download the solvers, but are allowed only to use the service of the solvers. In Open CFLP, solving services are named by URIs, and the URIs can be found by keywords that characterize their services. Note that we do not treat names of each solver. First we find a service of solving equations over the domain defined by a higher-order functional logic program.

```
HOLNURIs = LookupService["higher-order functional logic programming"]
```

`LookupService` returns a list of URIs. Each URI denotes a specification of a constraint solving service. The user can browse the specifications associated with a given URI u by calling:

```
HOLNSpec = GetServiceSpec[u]
```

where `HOLNSpec` is a list of specifications such as configuration options and comments. In this way, the user can choose the URI, let's call it `HOLNURI`, which provides the best description for his needed service. The call

```
aHOLN = FindSolver[HOLNURI]
```

returns an entity which we call *elementary collaborative*. An elementary collaborative is a collection of basic solvers that satisfy the descriptions associated with the argument of `FindSolver`. The reason why an elementary collaborative is implemented as a collection of solvers is to solve goals concurrently with its solvers. Elementary collaboratives can be configured by calling `ConfigSolver`. For example,

```
aHOLN = ConfigSolver[aHOLN, {Prog → R, Calculus →"HOLN"}]
```

configures the elementary collaborative `aHOLN` with the functional logic program stored in `R` and with the higher-order lazy narrowing calculus HOLN.

Similarly, we can find elementary collaboratives `aDeriv`, `aPolyn` and `aLinear` for solving systems of differential equations, polynomials and linear equations respectively.

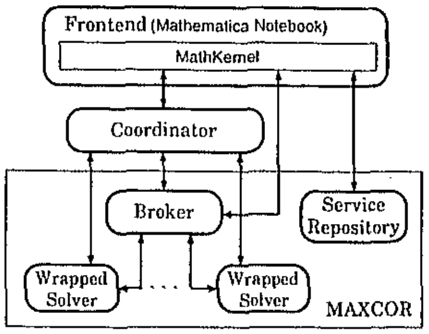


Fig. 1. The architecture of Open CFLP

Next, we program a collaborative. For this example, we define a collaborative in which the elementary collaboratives `aHOLN`, `aDeriv`, `aPolyn` and `aLinear` collaborate as follows:

```
aCollabo = NewCollaborative[repeat[seq[aHOLN, aDeriv, aPolyn, aLinear]]]
```

`seq[aHOLN, aDeriv, aPolyn, aLinear]` is a collaborative which applies the elementary collaboratives `aHOLN`, `aDeriv`, `aPolyn` and `aLinear` sequentially, from left to right. Furthermore, `aCollabo` applies repeatedly the sequential collaborative mentioned above to the goal until it reaches a fixed-point.

Finally, we apply the goal `G` to the collaborative `aCollabo`:

```
ApplyCollaborative[aCollabo, {G}]
```

and obtain the following solutions.

$$\{\{y \mapsto \lambda[\{t\}, -10 + \tfrac{20(-1+t^2)}{3t}], T \mapsto -\tfrac{2}{5}\}, \{y \mapsto \lambda[\{t\}, -10 + \tfrac{20(-1+t^2)}{3t}], T \mapsto \tfrac{5}{2}\}\}$$

3 System Architecture

Figure 1 shows the architecture of Open CFLP. Open CFLP consists of five kinds of components: *frontend, coordinator, object wrapper for solver, service repository* and *broker*.

The frontend is a user interface implemented using Mathematica notebook which allows the user to input Open CFLP problems. The coordinator interprets collaboratives and manages the collaboration among solvers.

Functionally, the object wrappers, service repository and broker are grouped together to form a framework called MAXCOR (MAth eXchange for CORBA). MAXCOR realizes transparent communication between solvers in an open environment. Solvers may be implemented in different languages and may use proprietary data formats. MAXCOR has to solve this heterogeneity problem by adopting a standard for communication, and by translating between proprietary data and a standard data format. In order to solve this problem, we employ CORBA [1] as the communication protocol, and MathML [3] and DOM/Value mapping [2] as the common data format for representing mathematical expressions. The

MAXCOR components—object wrappers, service repository and broker—are implemented as CORBA objects. Each solver is wrapped by a corresponding object wrapper which translates between the proprietary data format of the solver and the common data format. Object wrappers implement a common interface and specify a language-specific interface for the solver implementation. For instance, in order to develop a solver with our object wrapper for Mathematica, the developer must implement the Mathematica interface of the object wrapper.

In Open CFLP, the identification of solvers should be location independent. Solvers are found with a discovery service provided by a service repository and a broker. The service repository provides two operations: `LookupService` and `GetServiceSpec`, and the broker provides the operation `FindSolver`. These operations were described in Section 2.

Finally, we summarize the properties of Open CFLP obtained by MAXCOR:

- portability—language and platform independence among solvers,
- scalability and modularity—solvers implemented as CORBA objects,
- data and operation interoperability—the common IDL definition which includes DOM/Value mapping and MathML, and
- location independence of solvers—the discovery service.

4 Conclusions

We have described an open system for constraint functional logic programming. The system solves equations by a collaboration among a functional logic solver based on higher-order lazy narrowing calculus and specialized solvers which implement methods over several domains. Those solvers are located in an open environment. Our system achieves transparent access to solvers and relies on global standards for communication and data representation. As future work, we intend to address security issues. We will introduce SSL and CORBA firewall, which allows secure communication between MAXCOR components through firewalls.

References

1. The Common Object Request Broker; Architecture and Specification, Editional Revision: CORBA 2.6, Object Management Group, 2002.
2. XML DOM: DOM/Value Mapping Specification, Object Management Group, 2001.
3. `http://www.w3.org/Math`.
4. T. Ida, N. Kobayashi, and M. Marin. "An Open Environment for Cooperative Scientific Problem Solving." In Proc. of the 4th Int'l Mathematica Symposium (IMS 2001), pp. 71-78, Chiba, Japan, 2001.
5. M. Marin, T. Ida, and T. Suzuki. "Higher-Order Lazy Narrowing Calculus: A Solver for Higher-Order Equations." In Proc. of the 8th Int'l Conf. Computer Aided Systems (EuroCAST 2001), LNCS, vol. 2178, pp.478–493, Springer-Verlag, 2001.
6. S. Wolfram. The Mathematica Book, 4th Edition, Wolfram Media Inc. Champaign, Illinois, USA, and Cambridge University Press, 1999.

Reducing Search Space in Solving Higher-order Equations*

Tetsuo Ida[1], Mircea Marin[1], and Taro Suzuki[2]

[1] Institute of Information Sciences and Electronics, University of Tsukuba, Tsukuba 305-8573, Japan
{ida,mmarin}@score.is.tsukuba.ac.jp
[2] Department of Computer Software, University of Aizu, Aizu Wakamatsu 965-8580, Japan
taro@u-aizu.ac.jp

Abstract. We describe the results of our investigation of equational problem solving in higher-order setting. The main problem is identified to be that of reducing the search space of higher-order lazy narrowing calculi, namely how to reduce the search space without losing the completeness of the calculi. We present a higher-order calculus HOLN_0 as a system of inference rules and discuss various refinements that enable the reduction of the search space by eliminating some sources of non-determinism inherent in the calculus.

1 Introduction

With the advancement of high speed computers, it becomes possible to make significant discoveries in some area where in the past the search space involved is so large that exhaustive search was infeasible. Those successful discoveries are realized by the combination of development of methods for reducing the search space by careful analysis of properties of the subject domain and by appealing to brute force of powerful computers. Many techniques have been developed to reduce the search space in the discipline of computational intelligence, such as simulated annealing and genetic algorithms. Most of techniques to date rely in part on statistical methods.

Contrary to the above approach, in our study we only use a logical framework and investigate the reduction of search space. We further focus on equational problem solving. To treat the problem abstractly we use simply-typed lambda calculus. Our problem is then to find solutions of higher-order equations over the domain of simply-typed λ-terms. The problem is not only of theoretical interest but also of practical importance since the formulation is used to model computation of functional logic programming and constraint solving.

The paper is organized as follows. We devote Section 2 to the background of our study, focusing on narrowing. We try to explain it in an intuitive way,

* This work has been also supported in part by JSPS Grant-in-Aid for Scientific Research (B) 12480066, 2000-2002, and (C) 13680388, 2001-2002. Mircea Marin has been supported by JSPS postdoc fellowship 00096, 2000-2001.

without giving formal definitions of notions. The "undefined" notions in this section are defined either in Section 3 or readers are referred to a standard textbook [BN98]. In particular, we assume readers' knowledge of basic theories of rewriting. In Section 3 we give main results of our research. We first present a higher-order lazy narrowing calculus HOLN_0 as a system of inference rules that operate on goals. Then we give various refinements of HOLN_0 together with the completeness and soundness results. In Section 4 we summarize our results and point out directions of further research.

2 Background

Our problem is as follows. Given an equational theory represented as a set $\mathcal{R}$ of rewrite rules,[1] prove a formula

$$\exists X_1, \ldots, X_m.(s_1 \approx t_1, \ldots, s_n \approx t_n). \tag{1}$$

The sequence of equations in (1) is called a *goal*. We are interested not only in the validity of the formula, but the values, say $v_1, \ldots, v_m$, that instantiate the variables $X_1, \ldots, X_m$. Those values are represented as a substitution $\tau = \{X_1 \mapsto v_1, \ldots, X_m \mapsto v_m\}$. Obtaining such a substitution τ is called *solving* the problem (1), and τ is called the *solution*.

There arise naturally questions about the intended domain over which a solution is sought, and a method for finding it. The notion of solution could be formally defined. For the present purpose, it suffices to note that the intended domain can be fixed to be that of terms generated from a given signature, i.e. the domain of term algebra. It coincides with Herbrand universe in the framework of the first-order predicate logic.

A solution is then defined as follows.

Definition 1 An equation $s \approx t$ is called *immediately solvable* by a substitution τ if $s\tau = t\tau$.

Definition 2 Let G be a goal and $\mathcal{R}$ be a rewriting system. A substitution τ is called a solution of G for $\mathcal{R}$ if

$$G\tau \to_{\mathcal{R}}^{*} F\tau,$$

where $F\tau$ is a sequence of immediately solvable equations by an empty substitution.

The next question is how to find the solution; the question to which we address in this paper. An exhaustive search in this domain is clearly infeasible. A systematic method called *narrowing* for searching the solution in the first-order case was devised in the late 70s [Fay79,Hul80].

The narrowing method is the following. We define narrowing for an equation. Extension to a goal is straightforward.

[1] Reader can assume that our discussion in this section is in the framework of first-order term rewrite systems, although it can be carried over to simply-typed (higher-order) pattern rewrite systems.

Definition 3 (Narrowing).

(i) If the given equation e is immediately solvable, we employ the most general substitution σ. We denote this as $e \rightsquigarrow_\sigma \texttt{true}$.

(ii) We rewrite e to e' in the following way:

$$e \rightsquigarrow_\theta e' \tag{2}$$

if there exist

(a) a fresh variant $l \rightarrow r$ of a rewrite rule in $\mathcal{R}$, and

(b) a non-variable position p in e

such that $e|_p\theta = l\theta$ by the most general unifier θ and $e' = (e[r]_p)\theta$.

The relation defined above is called *one step narrowing*. We say that e is narrowed to e' in one step.

In case (i), the substitution σ is clearly a solution since it satisfies Definition 2. In case (ii), θ approximates a solution. We narrow the equation e repeatedly by (ii) until it is an immediately solvable one e_n by a substitution σ, and then rewrite e_n to true by (i). Namely,

$$e(= e_0) \rightsquigarrow_{\theta_1} e_1 \rightsquigarrow_{\theta_2} \cdots \rightsquigarrow_{\theta_n} e_n \rightsquigarrow_\sigma \texttt{true}. \tag{3}$$

Here we explicitly write the substitution that we employ in each step of narrowing. We will further denote the derivation (3) as:

$$e(= e_0) \rightsquigarrow_\theta^* \texttt{true} \tag{4}$$

where θ is the composition of substitutions employed in each one step narrowing, i.e. $\theta_1 \cdots \theta_n\sigma$. The sequence (4) of one step narrowing is called a *narrowing derivation.*

It is easy to prove that θ is a solution of e. More generally, we have the following soundness theorem.

Theorem 1. *Let G be a goal, and $\mathcal{R}$ be a rewrite system. A substitution θ in a narrowing derivation*

$$G \rightsquigarrow_\theta^* \top, \tag{5}$$

is a solution of G, where $\top$ is a goal consisting only of trues.

Soundness is not much of a problem in the quest for reduction of the search space. What is intriguing is the property of completeness which is defined as follows. Let $\mathit{Vars}(t)$ denote the set of free variables in term t.

Definition 4 Narrowing is *complete* if for any solution τ of a goal G there exists a substitution θ such that

$$G \rightsquigarrow_\theta^* \top \text{ and } \theta \preceq \tau[\mathit{Vars}(G)].$$

Intuitively, completeness of narrowing is the ability to find all solutions. The completeness depends on the kind of solutions that we would like to have and on the kind of rewrite systems. Understanding to what extent generalization of these factors is possible requires deep investigation. Our problem then becomes how to reduce the search space while ensuring the completeness.

There are two sources of non-determinism in narrowing, i. e. choices of rewrite rules at Definition 3(ii)(a) and of positions in an equation at Definition 3(ii)(b). Non-determinism at (a) is unavoidable since the choice determines the solutions, and we have to explore all the possibilities. Therefore we focused on eliminating unnecessary visits of positions in an equation subjected to narrowing. Note that the restriction to non-variable positions at (b) produces already big reduction of the search space. The method without the condition of non-variable-ness is *paramodulation*. It is easy to see that in the case of paramodulation the search space expands greatly since $e|_p\theta = l\theta$ means simply an instantiation of a variable $e|_p$ to the left hand side of the variant of a rewrite rule. Narrowing in the first-order case eliminates this possibility. However, the problem of variable instantiation pops up in the higher-order case.

There are certain assumptions and restrictions that we have to make in order to make our problem tractable. We assume that

- $\mathcal{R}$ is confluent pattern rewrite system[2], and
- solutions are $\mathcal{R}$-normalized.

Confluence is fundamental for our system to be regarded as a sound computing system. A pattern rewrite system is a suitable subset of higher-order systems to which several important notions and properties are carried over from the first-order systems; e.g., pattern unification is unitary.

A crucial observation in our study of the reduction of the search space is the lifting lemma introduced originally in order to prove the completeness [Hul80], and the standardization theorem [HL92].

Lemma 1 (Lifting lemma). *Let G be a goal and τ be a solution of G. For*

$$G\tau \rightarrow^*_{\mathcal{R}} F\tau,$$

there exists a narrowing derivation

$$G \leadsto^*_\theta F \text{ such that } \theta \preceq \tau[\mathcal{V}ars(G)].$$

$F\tau$ is reached from $G\tau$ by repeated rewriting of $G\tau$ without backtracking if $\mathcal{R}$ is confluent. The lifting lemma tells that for any such rewrite sequence, we can find a corresponding narrowing derivation. Here "corresponding" means that rewrite sequence and the narrowing derivation employ the same rewrite rules at the same positions. There are many paths to reach $F\tau$, but we do not have to explore all these paths if there is a "standard" path to $F\tau$. This means that we only have to consider positions that make the rewrite sequence standard. Then,

[2] The definition of pattern rewrite system is given in Section 3

by the lifting lemma we can focus on the corresponding narrowing derivation. Indeed, for a class of left-linear confluent systems such a standard derivation exists [Bou83,Suz96]. In the standard rewrite sequence, terms are rewritten in an outside-in manner, and hence equations are narrowed in an outside-in manner [IN97].

The next task is how to define a calculus that generates a narrowing derivation that corresponds to the standard derivation. Such a calculus can easily be shown to hold the completeness property. In our previous research we designed lazy narrowing calculus called LNC with completeness proofs [MO98,MOI96]. LNC is a rule-based specification of lazy narrowing in the first-order framework. In particular LNC with the eager variable elimination strategy serves as our starting point. LNC is designed to be more general than the one that generates the narrowing derivation corresponding to the standard rewrite sequence, however.

3 Higher-Order Lazy Narrowing Calculi

We now present higher-order lazy narrowing calculi to be called HOLN. HOLN are higher-order extensions of LNC as well as a refinement of Prehofer's higher-order narrowing calculus LN [Pre98] using the results obtained in the study of LNC.

We developed a family of HOLN from the initial calculus HOLN$_0$. The rest of the calculi are derived by refining HOLN$_0$. Each calculus is defined as a pair consisting of a domain of simply-typed λ-terms and a set of inference rules. We always work with $\beta\eta^{-1}$ normal forms, and assume that transformation of a simply-typed λ-term into its $\beta\eta^{-1}$ normal form is an implicit meta operation.

Since the domain is the same in all the calculi, we only specify the set of inference rules. The inference rules operate on pairs consisting of a sequence G of equations and a set W of variables to which normalized values of the solution is bound. This pair is denoted by $E|_W$, and is called *goal of HOLN*. The reason for this new definition of a goal is that we usually specify our problem as a pair of equations together with the set of variables for which we intend to compute normalized bindings. This pair can be carried along throughout the process of equation solving by the calculi.

We present four calculi of HOLN together with the soundness and completeness results. The results are given without proofs. Interested readers should refer to [MSI01].

Before describing the calculi, we briefly explain our notions and notations.

- X, Y and H, possibly subscripted, denote free variables. Furthermore, whenever H (possibly subscripted) is introduced, it is meant to be fresh.
- x, y, z (possibly subscripted) denote bound variables.
- v denotes a bound variable or a function symbol.
- $\overline{t_n}$ denotes a sequence of terms $t_1, \ldots, t_n$.
- A term of the form $\lambda\overline{x_k}.X(\overline{t_n})$ is called *flex*, and *rigid* otherwise.
- An equation is *flex* if both sides are flex terms.

- A *pattern* t is a term such that every occurrence of the form $X(\overline{t_n})$ in t satisfies that $\overline{t_n}$ is a sequence of distinct bound variables. Furthermore, the pattern t is called *fully-extended* if $\overline{t_n}$ is a sequence of all the bound variables in the scope of X.
- A pattern is *linear* if no free variable occurs more than once in it.

Definition 5 (Pattern Rewrite System)

- A *pattern rewrite system* (PRS for short) is a finite set of rewrite rules $f(\overline{l_n}) \to r$, where $f(\overline{l_n})$ and r are terms of the same base type, and $f(\overline{l_n})$ is a pattern.
- A *left-linear* (resp. *fully-extended*) PRS is a PRS consisting of rewrite rules which have a linear (resp. fully-extended) pattern on their left hand side. A fully-extended PRS is abbreviated to EPRS, and a left-linear EPRS is abbreviated to LEPRS.

3.1 HOLN$_0$

The inference rules are relations of the form

$$(E_1, e, E_2)\downharpoonleft_W \Rightarrow_{\alpha,\theta} (E_1\theta, E, E_2\theta)\downharpoonleft_{W'}$$

where e is the selected equation, α is the label of the inference rule, θ is the substitution computed in this inference step, $W' = \bigcup_{X\in W} \mathcal{V}ars(X\theta)$, and E is a sequence of equations whose elements are the *descendants* of e. If an equation e' is an equation in E_1 or E_2, then e' has only one descendant in the inference step, namely the corresponding equation $e'\theta$ in $E_1\theta$ or $E_2\theta$.

HOLN distinguish between *oriented* equations written as $s \rhd t$, and *unoriented* equations written as $s \approx t$. In oriented equations, only the left hand sides are allowed to be rewritten; for a solution θ, $s\theta \rhd t\theta \to_{\mathcal{R}}^{*} t\theta \rhd t\theta$. For brevity, we introduce the following notation: $s \trianglerighteq t$ denotes $s \approx t$, $t \approx s$, or $s \rhd t$; $s \approxeq t$ denotes $s \approx t$, $t \approx s$, $s \rhd t$, or $t \rhd s$.

3.2 The Calculus HOLN$_0$

HOLN$_0$ consists of three groups of inference rules: *pre-unification rules, lazy narrowing rules,* and *removal rules of flex equations.*

Pre-unification rules

[i] Imitation.

$$(E_1, \lambda\overline{x_k}.X(\overline{s_m}) \approxeq \lambda\overline{x_k}.g(\overline{t_n}), E_2)\downharpoonleft_W \Rightarrow_{[i],\theta} (E_1, \overline{\lambda\overline{x_k}.H_n(\overline{s_m}) \approxeq \lambda\overline{x_k}.t_n}, E_2)\theta\downharpoonleft_{W'}$$

where $\theta = \{X \mapsto \lambda\overline{y_m}.g(\overline{H_n(\overline{y_m})})\}$.

[p] Projection.
If $\lambda\overline{x_k}.t$ is rigid then

$$(E_1, \lambda\overline{x_k}.X(\overline{s_m}) \approxeq \lambda\overline{x_k}.t, E_2)\downharpoonright_W \Rightarrow_{[\mathsf{p}],\theta} (E_1, \lambda\overline{x_k}.X(\overline{s_m}) \approxeq \lambda\overline{x_k}.t, E_2)\theta\downharpoonright_{W'}$$

where $\theta = \{X \mapsto \lambda\overline{y_m}.y_i(\overline{H_n(\overline{y_m})})\}$

[d] Decomposition.

$$(E_1, \lambda\overline{x_k}.v(\overline{s_n}) \trianglerighteq \lambda\overline{x_k}.v(\overline{t_n}), E_2)\downharpoonright_W \Rightarrow_{[\mathsf{d}],\epsilon} (E_1, \overline{\lambda\overline{x_k}.s_n \trianglerighteq \lambda\overline{x_k}.t_n}, E_2) \downharpoonright_{W'}$$

Lazy narrowing rules

[on] Outermost narrowing at non-variable position.
If $f(\overline{l_n}) \rightarrow r$ is an $\overline{x_k}$-lifted rewrite rule of $\mathcal{R}$ then

$$\begin{aligned}(E_1, \lambda\overline{x_k}.f(\overline{s_n}) \trianglerighteq \lambda\overline{x_k}.t, E_2) \downharpoonright_W \Rightarrow_{[\mathsf{on}],\epsilon} &(E_1, \overline{\lambda\overline{x_k}.s_n \rhd \lambda\overline{x_k}.l_n},\\ &\lambda\overline{x_k}.r \trianglerighteq \lambda\overline{x_k}.t, E_2) \downharpoonright_{W'}\end{aligned}$$

[ov] Outermost narrowing at variable position.
If $f(\overline{l_n}) \rightarrow r$ is an $\overline{x_k}$-lifted rewrite rule of $\mathcal{R}$ and $\begin{cases}\lambda\overline{x_k}.X(\overline{s_m}) \text{ is not a pattern}\\ \text{or}\\ X \notin W\end{cases}$
then

$$\begin{aligned}(E_1, \lambda\overline{x_k}.X(\overline{s_m}) \trianglerighteq \lambda\overline{x_k}.t, E_2) \downharpoonright_W \Rightarrow_{[\mathsf{ov}],\theta} &(E_1\theta, \overline{\lambda\overline{x_k}.H_n(\overline{s_m\theta}) \rhd \lambda\overline{x_k}.l_n},\\ &\lambda\overline{x_k}.r \trianglerighteq \lambda\overline{x_k}.t\theta, E_2\theta) \downharpoonright_{W'}\end{aligned}$$

where $\theta = \{X \mapsto \lambda\overline{y_m}.f(\overline{H_n(\overline{y_m})})\}$.

Removal rules

[t] Trivial equation.

$$(E_1, \lambda\overline{x_k}.X(\overline{s_n}) \approxeq \lambda\overline{x_k}.X(\overline{s_n}), E_2)\downharpoonright_W \Rightarrow_{[\mathsf{t}],\epsilon} (E_1, E_2)\downharpoonright_W$$

[fs] Flex-same.
If $X \in W$ then

$$(E_1, \lambda\overline{x_k}.X(\overline{y_n}) \trianglerighteq \lambda\overline{x_k}.X(\overline{y'_n}), E_2)\downharpoonright_W \Rightarrow_{[\mathsf{fs}],\theta} (E_1, E_2)\theta\downharpoonright_{W'}$$

where $\theta = \{X \mapsto \lambda\overline{y_n}.H(\overline{z_p})\}$ with $\{\overline{z_p}\} = \{y_i \mid y_i = y'_i, 1 \leq i \leq n\}$.

[fd] Flex-different.
If $\begin{cases}X \in W \text{ and } Y \in W & \text{if } \trianglerighteq \text{ is } \approx\\ X \in W & \text{if } \trianglerighteq \text{ is } \rhd\end{cases}$ then

$$(E_1, \lambda\overline{x_k}.X(\overline{y_m}) \trianglerighteq \lambda\overline{x_k}.Y(\overline{y'_n}), E_2)\downharpoonright_W \Rightarrow_{[\mathsf{fd}],\theta} (E_1, E_2)\theta\downharpoonright_{W'}$$

where $\theta = \{X \mapsto \lambda\overline{y_m}.H(\overline{z_p}), Y \mapsto \lambda\overline{y'_n}.H(\overline{z_p})\}$ with $\{\overline{z_p}\} = \{\overline{y_m}\} \cap \{\overline{y'_n}\}$.

The pre-unification rules and the removal rules constitute a system equivalent to the transformation system for higher-order pre-unification of Snyder and Gallier [SG89]. The lazy narrowing rules play the role of one step narrowing in more elementary way. Namely narrowing is applied at the outermost position (excluding the λ-binder) of either side of the terms of the selected equation, leaving the computation of the most general unifier in later steps of the narrowing derivation. Note also that the system performs narrowing at variable positions as well by the rule [ov], in contract to the first-order narrowing, where narrowing at variable position is unnecessary.

In HOLN we do not have to search for positions at which subterms are narrowed, since terms are processed at the outermost position. Narrowing defined in Definition 3(ii) is decomposed into more elementary operations of the eight inference rules and the non-determinism associated with search for positions have disappeared. This facilitates the implementation and analysis of the behavior of narrowing. However, we now introduced a new source of non-determinism. Namely, the non-determinism associated with the choices of the inference rules. More than one rule can be applied to the same goal.

Let C denote a calculus in the family of HOLN. We call an inference step of calculus C *C-step*. A *C-derivation* is a sequence of C-steps

$$E|_W (= E_0|_{W_0}) \Rightarrow_{\alpha_1,\theta_1} E_1|_{W_1} \Rightarrow_{\alpha_2,\theta_2} \cdots \Rightarrow_{\alpha_n,\theta_n} E_n|_{W_n},$$

where E_n consists of flex equations and there is no C-step starting with $E_n|_{W_n}$. We write this derivation as $E_0|_{W_0} \Rightarrow_\theta^* E_n|_{W_n}$, where $\theta = \theta_1 \cdots \theta_n$, as before.

We redefine completeness for calculus C.

Definition 6 A substitution τ is a C-solution of a goal $E|_W$ for $\mathcal{R}$ if

- $\tau|_W$ is $\mathcal{R}$-normalized, and
- τ is a solution of E.

Definition 7 Calculus C is *C-complete* if for any C-solution τ of a goal $E|_W$ there exists a substitution θ such that

$$E|_W \Rightarrow_\theta^* F|_{W'} \text{ and } \theta\sigma \preceq \tau[\mathcal{V}ars(E)], \text{ where } \sigma \text{ is a } C\text{-solution of } F|_{W'}.$$

The prefix "C-" is omitted whenever it is clear from the context.

We have for HOLN$_0$ the following result.

Theorem 2. *Let $\mathcal{R}$ be a confluent PRS. The calculus* HOLN$_0$ *is sound and complete.*

3.3 HOLN$_1$

When $\mathcal{R}$ is a confluent LEPRS, we can design a calculus that has more deterministic behavior with respect to the application of rule [on]. HOLN$_1$ is a refinement of HOLN$_0$ in that HOLN$_1$ does not apply rule [on] to parameter-passing descendants of the form $\lambda\overline{x_k}.f(\overline{s_n}) \rhd \lambda\overline{x_k}.X(\overline{y_k})$ where f is a defined function sysmbol. Here a parameter-passing descendant is defined as follows.

Definition 8 A *parameter-passing equation* is any descendant, except the last one, of a selected equation in [on]-step or [ov]-step. A *parameter-passing descendant* is either a parameter-passing equation or a descendant of a parameter-passing descendant.

Note that parameter-passing descendants are always oriented equations. To distinguish them from the other oriented equations, we mark parameter-passing descendants as $s \blacktriangleright t$.

We define $[\mathrm{on}]_r$ as follows.

$[\mathrm{on}]_r$ Restricted outermost narrowing at non-variable position.
If $f(\overline{l_n}) \to r$ is an $\overline{x_k}$-lifted rewrite rule of $\mathcal{R}$ and the selected equation $\lambda\overline{x_k}.f(\overline{s_n}) \trianglerighteq \lambda\overline{x_k}.t$ is not of the form $\lambda\overline{x_k}.f(\overline{s_n}) \blacktriangleright \lambda\overline{x_k}.X(\overline{y_k})$ then

$$\begin{aligned}(E_1, \lambda\overline{x_k}.f(\overline{s_n}) \trianglerighteq \lambda\overline{x_k}.t, E_2)\downharpoonright_W \Rightarrow_{[\mathrm{on}],\epsilon} (E_1, &\overline{\lambda\overline{x_k}.s_n \triangleright \lambda\overline{x_k}.l_n},\\ &\lambda\overline{x_k}.r \trianglerighteq \lambda\overline{x_k}.t, E_2)\downharpoonright_{W'}\end{aligned}$$

We replace [on] of HOLN_0 by $[\mathrm{on}]_r$, and let HOLN_1 be the resulting calculus.

We note that HOLN_1 simulates the rewriting in an outside-in manner by the combination of the lazy narrowing rules and the imitation rule.

We have for HOLN_1 the following result.

Theorem 3. *Let $\mathcal{R}$ be a confluent LEPRS. Calculus* HOLN_1 *is sound and complete.*

The proof of this theorem is based on the standardization theorem for confluent LEPRSs, which has been proved by van Oostrom [vO00]. This result is the higher-order version of the eager variable elimination for LNC.

3.4 HOLN_2

Calculus HOLN_2 is a specialization of HOLN_0 for confluent constructor LEPRSs, i.e., confluent LEPRSs consisting of rewrite rules $f(\overline{l_n}) \to r$ with no defined symbols in $l_1, \ldots, l_n$. We introduce

- a new inference rule [c] for parameter-passing descendants, and
- an equation selection strategy $\mathcal{S}_{\mathrm{left}}$.

We define inference rule [c] as follows:

[c] Constructor propagation.
If $\exists s \blacktriangleright \lambda\overline{x_k}.X(\overline{y_k}) \in E_1$ and $s' = \lambda\overline{y_k}.s(\overline{x_k})$ then

$$(E_1, \lambda\overline{z_n}.X(\overline{t_k}) \trianglerighteq \lambda\overline{z_n}.u, E_2)\downharpoonright_W \Rightarrow_{[\mathrm{c}],\epsilon} (E_1, \lambda\overline{z_n}.s'(\overline{t_k}) \trianglerighteq \lambda\overline{z_n}.u, E_2)\downharpoonright_{W'}.$$

We then define the calculus HOLN_2 as consisting of the inference rules of HOLN_0 and [c], where [c] has the highest priority. We can show that HOLN_2 does not produce parameter-passing descendants if it respects the equation selection strategy $\mathcal{S}_{\mathrm{left}}$.

We say that calculus C respects equation selection strategy S_{left} if every C-step is applied to the leftmost selectable equation. As a consequence, there is no non-determinism between inference rules [on] and [d] for parameter-passing descendants. Thus, solving parameter-passing descendants becomes more deterministic.

We have for HOLN$_2$ the following result.

Theorem 4. *Let $\mathcal{R}$ be a confluent constructor LEPRS. Calculus* HOLN$_2$ *that respects strategy S_{left} is complete. Moreover,* HOLN$_2$ *with strategy S_{left} is sound for goals consisting only of unoriented equations.*

3.5 HOLN$_3$

Suppose we have a HOLN$_0$-derivation

$$E\lfloor_W \Rightarrow_{\theta}^{*} (E_1, e, E_2)\lfloor_{W'}$$

and a solution γ' of $(E_1, E_2)\lfloor_{W'}$. If for any such γ' there exists a solution γ of $(E_1, e, E_2)\lfloor_{W'}$ such that $\theta\gamma\lceil_{\mathcal{V}ars(E)} = \theta\gamma'\lceil_{\mathcal{V}ars(E)}$, then e does not contribute to the solution of $E\lfloor_W$. We call such an equation *redundant*. One of redundant equations is $\lambda\overline{x_k}.s \blacktriangleright \lambda\overline{x_k}.X(\overline{y_k})$ in $(E_1, \lambda\overline{x_k}.s \blacktriangleright \lambda\overline{x_k}.X(\overline{y_k}), E_2)\lfloor_W$, where $\overline{y_k}$ is a permutation of the sequence $\overline{x_k}$, and $X \notin \mathcal{V}ars(E_1, \lambda\overline{x_k}.s, E_2)$.

Calculus HOLN$_3$ is a refinement of HOLN$_0$ for LEPRSs which avoids solving the above redundant equation. We introduce the following inference rule [rm] and define HOLN$_3$ as consisting of the inference rules of HOLN$_0$ and [rm] that has to be applied with the highest priority.

[rm] Removal of redundant equations.
If $\overline{y_k}$ is a permutation of the sequence $\overline{x_k}$, and $X \notin \mathcal{V}ars(E_1, \lambda\overline{x_k}.s, E_2)$ then

$$(E_1, \lambda\overline{x_k}.s \blacktriangleright \lambda\overline{x_k}.X(\overline{y_k}), E_2)\lfloor_W \Rightarrow_{[\text{rm}],\epsilon} (E_1, E_2)\lfloor_W .$$

We have for HOLN$_3$ the following result.

Theorem 5. *Let $\mathcal{R}$ be a confluent LEPRS. Calculus* HOLN$_3$ *is sound and complete.*

3.6 Rationale for Refinements

After presenting the family of HOLN, we can now see more clearly the rationale of our refinements. First, we noted that [ov] is the main source of non-determinism, since it allows us to compute approximations of any solution, be it $\mathcal{R}$-normalized or not. Our first investigation was how to restrict the application of [ov] without losing completeness. We achieved this by redefining the notion of goal. A goal of HOLN is a pair $E\lfloor_W$ where E is a sequence of equations, and W is a finite set of variables for which we want to compute $\mathcal{R}$-normalized bindings. By keeping track of W, we can restrict the application of [ov] without losing completeness. The resulted calculus is HOLN$_0$.

Next, we studied how to reduce the non-determinism between the inference rules applicable to the same selected equation. We found two possible refinements of HOLN_0 which reduce this source of non-determinism. HOLN_1 is a refinement inspired by the eager variable elimination method of LNC. This refinement reduces the non-determinism between the inference rules applicable to parameter-passing descendants of the form $\lambda\overline{x_k}.f(\ldots) = \lambda\overline{x_k}.X(\ldots)$, where f is a defined function symbol. More precisely, [on] is not applied to such an equation, when it is selected. This refinement is valid for confluent LEPRSs.

HOLN_2 is a refinement inspired by the refinement of LNC for left-linear constructor rewrite systems. The idea is to avoid producing parameter-passing descendants with defined function symbols in the right-hand side. As a consequence, there is no non-determinism between [d] and [on] when the selected equation is a parameter-passing descendant. Thus, the calculus is more deterministic. We achieved this property by introducing a new inference rule [c] which has the highest priority, and a suitable equation selection strategy (which is the higher-order version of the leftmost equation selection strategy). This refinement is valid for confluent constructor LEPRSs.

HOLN_3 is a refinement aiming at detecting redundant equations. It is useful to detect redundant equations, because they can be simply eliminated from the goal, thus saving time and resources to solve them. We identified a criterion to detect redundant equations and introduced [rm] which simply eliminates a redundant equation from the goal.

4 Conclusions and Future Work

We have shown the results of our investigation of reducing the search space in solving higher-order equations. We presented a family of sound and complete lazy narrowing calculi HOLN designed to compute solutions which are normalized with respect to a given set of variables. All the calculi have been implemented as part of our distributed constraint functional logic system CFLP [MIS00].

An interesting direction of research is to extend HOLN to conditional PRSs. A program specification using conditions is much more expressive because it allows the user to impose equational conditions under which rewrite steps are allowed. Such an extension is quite straightforward to design, but it introduces many complexities for proving completeness.

References

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

[Bou83] G. Boudol. Computational Semantics of Term Rewriting Systems. Technical report, INRIA, February 1983.

[Fay79] M. Fay. First-Order Unification in Equational Theories. In *Proceedings of the 4th Conference on Automated Deduction*, pages 161–167, 1979.

[HL92] G. Huet and J.-J. Lévy. Computations in Orthogonal Rewriting Systems, I and II. In *Computational Logic, Essays in Honor of Alan Robinson*, pages 396–443. The MIT Press, 1992.

[Hul80] J.-M. Hullot. Canonical Forms and Unification. In *Proceedings of the 5th Conference on Automated Deduction*, volume 87 of *LNCS*, pages 318–334. Springer, 1980.

[IN97] T. Ida and K. Nakahara. Leftmost Outside-in Narrowing Calculi. *Journal of Functional Programming*, 7(2):129–161, 1997.

[MIS00] M. Marin, T. Ida, and T. Suzuki. Cooperative constraint functional logic programming. In T. Katayama, T. Tamai, and N. Yonezaki, editors, *International Symposium on Principles of Software Evolution (ISPSE 2000)*, Kanazawa, Japan, November 1-2, 2000. IEEE Computer Society.

[MO98] A. Middeldorp and S. Okui. A Deterministic Lazy Narrowing Calculus. *Journal of Symbolic Computation*, 25(6):733–757, 1998.

[MOI96] A. Middeldorp, S. Okui, and T. Ida. Lazy Narrowing: Strong Completeness and Eager Variable Elimination. *Theoretical Computer Science*, 167(1,2):95–130, 1996.

[MSI01] M. Marin, T. Suzuki, and T. Ida. Refinements of lazy narrowing for left-linear fully-extended pattern rewrite systems. Technical Report ISE-TR-01-180, Institute of Information Sciences and Electronics, University of Tsukuba, Tsukuba, Japan, to appear, 2001.

[Pre98] C. Prehofer. *Solving Higher-Order Equations. From Logic to Programming.* Foundations of Computing. Birkhäuser Boston, 1998.

[SG89] W. Snyder and J. Gallier. Higher-order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8:101–140, 1989.

[Suz96] T. Suzuki. Standardization Theorem Revisited. In *Proceedings of 5th International Conference, ALP'96*, volume 1139 of *LNCS*, pages 122–134, 1996.

[vO00] V. van Oostrom. Personal communication, August 2000.

A Deterministic Lazy Conditional Narrowing Calculus

Mircea Marin
Institute of Information Sciences and Electronics
University of Tsukuba, Tsukuba 305-8573, Japan

Abstract. (We show the completeness of a deterministic lazy conditional narrowing calculus with leftmost selection for the class of eft-linear fresh deterministic constructor-based conditional rewrite systems. This class of rewrite systems permits extra variables in the right-hand sides and conditions of its rewrite rules. This result is relevant for the designers of suitable computational models for functional logic programming, where the reduction of search space for solutions of systems of equations in theories presented by conditional rewrite systems is of paramount importance.

AMS Subject Classification: 68T15, 68T20.

Keywords and phrases: E-unification, Lazy narrowing.

1 Introduction

Narrowing is a general method for solving unification problems in equational theories presented by confluent term rewriting systems (TRSs for short). More recently, narrowing was proposed as the computational mechanism of functional-logic programming languages and several new results concerning the completeness of various narrowing strategies and calculi have been obtained in the past few years. Here completeness means that for every solution to a given goal a solution that is at least as general is computed by the narrowing strategy. Since narrowing is a complicated operation, numerous calculi consisting of a small number of more elementary inference rules that simulate narrowing have been proposed (e.g. [3, 8, 12])

In this paper we consider the lazy conditional narrowing calculus LCNC of [9]. LCNC is the extension of the lazy unconditional calculus LNC [8, 7] to conditional term rewrite systems (CTRSs for short). The extension is motivated by the observation that CTRSs are much more expressive than unconditional TRSs for describing interesting problems in a natural and concise way. However, the additional expressive power raises two problems: (1) completeness results are much harder to obtain, and (2) the search space increases dramatically because the conditions of the applied rewrite rule are added to the current goal.

In [9] three completeness results for LCNC are presented: (a) LCNC with leftmost selection is complete with respect to normalized solutions for confluent CTRSs without extra variables, (b) LCNC is strongly complete whenever basic conditional narrowing is complete, and (c) LCNC is complete for terminating level-confluent conditional rewrite systems. Strong completeness means completeness with respect to any selection function and basic conditional narrowing is known to be complete for several classes of terminating CTRSs. So the only completeness result which does not assume some kind of termination assumption does not permit extra variables in the conditions and right-hand sides of the rewrite rules.

The main results of this paper are:

1. We propose a refinement LCNC$_{\rm d}$ of the calculus LCNC with leftmost selection which is deterministic in the choice of the inference rule which is applied to a selected equation,

2. We propose the class of left-linear fresh deterministic constructor-based CTRSs

and show that LCNC$_{\rm d}$ is a complete calculus. Note that this class of CTRSs is not necessarily terminating, it has a simple syntactic characterization, and hence it is useful for functional logic programming.

The paper is structured as follows. In Section 2 we fix our notation and terminology and recall some relevant properties of conditional term rewriting and lazy conditional narrowing. In Section 3 we recall some recent results about LCNC$_\ell$. In Section 4 we introduce the refinement LCNC$_{\rm d}$ of LCNC$_\ell$ and prove that it is sound and complete. We provide some examples in Section 5 to confirm the effectiveness of our refinements.

2 Preliminaries

Familiarity with term rewriting ([1]) will be helpful. A recent survey of conditional term rewriting can be found in [11].

We consider a set $\mathcal{T}(\mathcal{F},\mathcal{V})$ of terms built from a set of function symbols with given arities $\mathcal{F}$ and a countably infinite set of variables $\mathcal{V}$. We write root(t) for the root symbol of a term t and $\mathcal{V}\mathrm{ar}(t)$ for the set of variables which occur in t. An *equation* is either an *unoriented* equation $s \approx t$, an *oriented* equation $s \rhd t$, or the constant $\mathtt{true}$. We assume that $\approx, \rhd, \mathtt{true} \notin \mathcal{F}$. An equation between identical terms is called *trivial*. A *goal* is a sequence of equations. The empty sequence is denoted by $\Box$. A *proper* goal does not contain any occurrences of $\mathtt{true}$.

A CTRS $\mathcal{R}$ is a set of rewrite rules of the form $l \to r \Leftarrow c$ such that $l \notin \mathcal{V}$ and the conditional part c is a proper goal. We require that $\mathcal{V}\mathrm{ar}(r) \subseteq \mathcal{V}\mathrm{ar}(l,c)$. (So we deal with so-called 3-CTRSs in this paper.) We write $l \to r$ for $l \to r \Leftarrow \Box$. If the conditional part of every rewrite rule consists of unoriented (oriented) equations then $\mathcal{R}$ is called a *join* (*oriented*) CTRS. $\mathcal{R}$ is called *left-linear* if all left-hand sides of its rewrite rules are *linear terms*, i.e., they have no multiple occurrences of the same variable. $\mathcal{R}$ is a *constructor-based* TRS (CCS for short) if the left-hand sides of its rewrite rules have defined symbols only at root positions.

We define the rewrite relation $\to_\mathcal{R}$ associated with $\mathcal{R}$ on proper terms as follows: $s \to_\mathcal{R} t$ if there exists a rewrite rule $l \to r \Leftarrow c$ in $\mathcal{R}$, a position p in s, and a substitution θ such that $s|_p = l\theta$, $t = s[r\theta]_p$, and $\mathcal{R} \vdash e$ for all equations e in $c\theta$. The latter is defined as follows: $\mathcal{R} \vdash s \rhd t$ if $s \to^*_\mathcal{R} t$ and $\mathcal{R} \vdash s \approx t$ if there exists a term u such that $s \to^*_\mathcal{R} u$ and $t \to^*_\mathcal{R} u$. We extend $\to_\mathcal{R}$ to equations as follows: $s \approx t \to_\mathcal{R} e$ if one of the following three alternatives holds: (1) $e = \mathtt{true}$ and $s = t$, (2) $e = s' \approx t$ and $s \to_{\mathcal{R}} s'$, or (3) $e = s \approx t'$ and $t \to_{\mathcal{R}} t'$, and $s \rhd t \to_{\mathcal{R}} e$ if either $e = \mathtt{true}$ and $s = t$ or $e = s' \approx t$ and $s \to_{\mathcal{R}} s'$. So unoriented equations are interpreted as joinability statements. Furthermore, for goals G we define $G \to_{\mathcal{R}} G'$ if $G = G_1, e, G_2$, $G' = G_1, e', G_2$, and $e \to_{\mathcal{R}} e'$. It is well-known that $\mathcal{R} \vdash G$ if and only if $G \to^*_{\mathcal{R}} \top$ where $\top$ denotes any sequence of $\mathtt{true}$s. The set $\mathcal{F}_{\mathcal{D}}$ of defined function symbols of $\mathcal{R}$ is defined as $\{\mathrm{root}(l) \mid l \to r \Leftarrow c \in \mathcal{R}\}$. Function symbols in $\mathcal{F}_{\mathcal{C}} = \mathcal{F} \setminus \mathcal{F}_{\mathcal{D}}$ are called *constructors*.

A substitution θ is a solution of a goal G if $G\theta \to^*_{\mathcal{R}} \top$. Since confluence is insufficient [14] to guarantee that joinability coincides with the equality relation induced by the underlying conditional equational theory, we decided to drop confluence and use the above notion of solution. We say that θ is *X-normalized* for a subset X of $\mathcal{V}$ if every variable in X is mapped to a normal form with respect to $\mathcal{R}$.

The lazy conditional narrowing calculus LCNC$_\ell$ [5] consists of the following inference rules:

[o] *outermost narrowing*

$$\frac{f(s_1, \dots, s_n) \simeq t, G}{s_1 \rhd l_1, \dots, s_n \rhd l_n, c, r \simeq t, G} \qquad \simeq \, \in \{\approx, \rhd, \backapprox\}$$

if $f(l_1, \dots, l_n) \to r \Leftarrow c$ is a fresh variant of a rewrite rule in $\mathcal{R}$,

[i] *imitation*

$$\frac{f(s_1, \dots, s_n) \simeq x, G}{(s_1 \simeq x_1, \dots, s_n \simeq x_n, G)\theta} \qquad \simeq \, \in \{\approx, \rhd, \backapprox, \lhd\}$$

if $\theta = \{x \mapsto f(x_1, \dots, x_n)\}$ with $x_1, \dots, x_n$ fresh variables,

[d] *decomposition*

$$\frac{f(s_1, \dots, s_n) \simeq f(t_1, \dots, t_n), G}{s_1 \simeq t_1, \dots, s_n \simeq t_n, G} \qquad \simeq \, \in \{\approx, \rhd\}$$

[v] *variable elimination*

$$\frac{s \simeq x, G}{G\theta} \qquad \frac{x \simeq s, G}{G\theta} \; s \notin \mathcal{V}$$

if $x \notin \mathcal{V}\mathrm{ar}(s)$, $\simeq \, \in \{\approx, \rhd\}$, and $\theta = \{x \mapsto s\}$,

[t] *removal of trivial equations*

$$\frac{s \simeq s, G}{G} \qquad \simeq \, \in \{\approx, \rhd\}$$

In the above rules we use $s \backapprox t$ to denote the equation $t \approx s$ and $s \lhd t$ to denote the equation $t \rhd s$.

Compared to the definition of LCNC in [9], in LCNC$_\ell$ leftmost selection is built-in. Another difference is that the *parameter-passing* equations $s_1 \rhd l_1, \dots, s_n \rhd l_n$ created by the outermost narrowing rule [o] are regarded as oriented equations. This is in line with the earlier completeness proofs for LNC and LCNC [7, 8, 9]. Furthermore, the conditions c are placed before $r \simeq t$ whereas in [9] they are placed after $r \simeq t$. The relative locations of c and $r \simeq t$

are irrelevant for the completeness results reported in that paper, but our new completeness proofs do not go through if we stick to the order in [9]. If the variable elimination rule [v] is applied to an equation between two different variables, we can only eliminate the variable on the right-hand side. Finally, the removal of trivial equations rule [t] is usually restricted to equations between identical variables. The variation adopted here avoids some infinite derivations as well as refutations which eventually compute substitutions which are subsumed by the empty substitution of the [t]-step and therefore redundant.

If G_1 and G_2 are the upper and lower goal in the inference rule $[\alpha]$, we write $G_1 \Rightarrow_{[\alpha]} G_2$. The applied rewrite rule or substitution may be supplied as subscript, that is, we write things like $G_1 \Rightarrow_{[\mathsf{o}], l \to r \Leftarrow c} G_2$ and $G_1 \Rightarrow_{[\mathsf{i}], \theta} G_2$. A finite LCNC$_\ell$-derivation $G_1 \Rightarrow_{\theta_1} \cdots \Rightarrow_{\theta_{n-1}} G_n$ may be abbreviated to $G_1 \Rightarrow^*_\theta G_n$ where θ is the composition $\theta_1 \cdots \theta_{n-1}$ of the substitutions $\theta_1, \ldots, \theta_{n-1}$ computed along its steps. An LCNC$_\ell$-refutation is an LCNC$_\ell$-derivation ending in the empty goal $\square$.

3 Soundness and Completeness

Soundness of LCNC$_\ell$ is not difficult to prove.

Theorem 1 *Let $\mathcal{R}$ be an arbitrary CTRS and G a proper goal. If $G \Rightarrow^*_\theta \square$ then θ is a solution of G.* $\square$

It is known that the unconditional variant of LCNC$_\ell$ is complete for (confluent) TRSs and normalized solutions θ ([8, Cor. 40]). The following well-known example shows that extra variables in the rewrite rules may result in incompleteness.

Example 2 *Consider the confluent CTRS $\mathcal{R} = \{\mathsf{f}(x) \to \mathsf{a} \Leftarrow y \approx \mathsf{g}(y), \mathsf{b} \to \mathsf{g}(\mathsf{b})\}$. The substitution $\theta = \{x \mapsto \mathsf{a}\}$ is a normalized solution of the goal $G = \mathsf{f}(x) \approx \mathsf{a}$. Any LCNC$_\ell$-derivation of G must start with the [o]-step $G \Rightarrow_{[\mathsf{o}]} x \rhd x_1, y_1 \approx \mathsf{g}(y_1), \mathsf{a} \approx \mathsf{a}$. The newly generated goal contains the equation $y_1 \approx \mathsf{g}(y_1)$ whose descendants are always of the form $z \approx \mathsf{g}(z)$ with $z \in \mathcal{V}$. Hence LCNC$_\ell$ cannot solve the goal G.*

To recover completeness, we impose determinism for goals and CTRSs.

Definition 3 *Let X be a finite set of variables. A proper goal $G = e_1, \ldots, e_n$ is called X-*deterministic *if $\mathcal{V}\mathrm{ar}(s_i) \subseteq X \cup \bigcup_{j=1}^{i-1} \mathcal{V}\mathrm{ar}(e_j)$ if $e_i = s_i \rhd t_i$ and $\mathcal{V}\mathrm{ar}(e_i) \subseteq X \cup \bigcup_{j=1}^{i-1} \mathcal{V}\mathrm{ar}(e_j)$ if $e_i = s_i \approx t_i$, for all $1 \leqslant i \leqslant n$. A CTRS $\mathcal{R}$ is called deterministic if the conditional part c of every rewrite rule $l \to r \Leftarrow c$ in $\mathcal{R}$ is $\mathcal{V}\mathrm{ar}(l)$-deterministic.*

The CTRS of Example 2 is not deterministic since the variable y occurring in the condition of the rewrite rule $\mathsf{f}(x) \to \mathsf{a} \Leftarrow y \approx \mathsf{g}(y)$ does not occur in its left-hand side $\mathsf{f}(x)$. The following oriented CTRS $\mathcal{R}$ is a natural encoding of the efficient computation of Fibonacci numbers:

$$\begin{array}{rcl@{\qquad}rcl}
0 + y & \to & y & \mathsf{fst}(\langle x, y\rangle) & \to & x \\
\mathsf{s}(x) + y & \to & \mathsf{s}(x + y) & \mathsf{snd}(\langle x, y\rangle) & \to & y \\
\mathsf{fib}(0) & \to & \langle 0, \mathsf{s}(0)\rangle & & & \\
\mathsf{fib}(\mathsf{s}(x)) & \to & \langle z, y + z\rangle \;\Leftarrow\; \mathsf{fib}(x) \rhd \langle y, z\rangle & & &
\end{array}$$

Note that $\mathcal{R}$ is deterministic. It is also terminating. However, when we change the oriented condition $\mathsf{fib}(x) \rhd \langle y, z\rangle$ into an unoriented condition $\mathsf{fib}(x) \approx \langle y, z\rangle$ then we lose termination: Because $\mathsf{fib}(\mathsf{s}(0)) \to^{+} \langle \mathsf{s}(0), \mathsf{s}(0)\rangle$ we have $\mathsf{fib}(0) \downarrow \langle 0, \mathsf{fst}(\mathsf{fib}(\mathsf{s}(0)))\rangle$ and thus $\mathsf{fib}(\mathsf{s}(0)) \to \langle \mathsf{fst}(\mathsf{fib}(\mathsf{s}(0))), 0 + \mathsf{fst}(\mathsf{fib}(\mathsf{s}(0)))\rangle$, which gives rise to an infinite rewrite sequence. As a consequence, the completeness results for LCNC presented in [9], which deal only with join CTRSs, do not apply to this CTRS. The completeness result presented below does not have this problem.

Theorem 4 ([5]) *Let $\mathcal{R}$ be a deterministic CTRS and G an X-deterministic goal. If θ is an X-normalized solution of G then $G \Rightarrow^{*}_{\theta'} \Box$ for some substitution θ' with $\theta' \leqslant \theta$ $[\mathcal{V}\mathrm{ar}(G)]$.* $\Box$

The proof is based on a transformation of rewrite proofs of $G\theta$. It is based on the observation that determinism allows to identify suitable sets of variables X_1, X_2, ... such that each intermediate goal G_i in the refutation $G \Rightarrow G_1 \Rightarrow G_2 \Rightarrow \cdots \Rightarrow \Box$ that is being constructed is X_i-deterministic.

Note that any proper goal G has a minimum set of variables $X \subseteq \mathcal{V}\mathrm{ar}(G)$ for which it is X-deterministic. This set will be denoted by X_G in the following. An X_G-normalized solution of G will simply be called *normalized.*

4 The Calculus LCNC$_\mathrm{d}$

The calculus LCNC$_\ell$ is highly nondeterministic. Further refinements are needed if we want to make LCNC$_\ell$ an appealing computational model. For the unconditional version LNC of LCNC$_\ell$, such refinements have been successfully identified [7] by confining the computation to more restricted classes of rewrite systems and to equations with strict semantics. A careful investigation reveals that the refinements of LNC can be carried over to LCNC$_\ell$. The final result of our analysis is a refinement of LCNC$_\ell$ called LCNC$_\mathrm{d}$ which is completely deterministic w.r.t. the selection of the inference rule that applies to a selected equation.

We start with some useful notions which are useful in defining LCNC$_\mathrm{d}$.

Descendants for LCNC$_\ell$ are defined as follows. The selected equation $f(s_1, \dots, s_n) \simeq t$ in rule [o] has $r \simeq t$ as only one-step descendant. In rule [i] all equations $s_i\theta \approx x_i$ are one-step descendants of the selected equation $f(s_1, \dots, s_n) \simeq x$. The selected equation $f(s_1, \dots, s_n) \simeq f(t_1, \dots, t_n)$ in rule [d] has all equations $s_i \simeq t_i$ as one-step descendants. Finally, the selected equations in rules [v] and [t] have no one-step descendants. One-step descendants of non-selected equations are defined as expected. Descendants are obtained from one-step descendants by reflexivity and transitivity. Observe that every equation in an LCNC$_\ell$-derivation descends from either an equation in the initial goal, a parameter-passing equation, or an equation in the conditional part of the rewrite rule used in the [o] rule.

A CTRS is called *fresh* if every oriented equation $s \approx t$ in the conditional part c of every rewrite rule $l \to r \Leftarrow c$ satisfies $\mathcal{V}\mathrm{ar}(l) \cap \mathcal{V}\mathrm{ar}(t) = \varnothing$.

To distinguish parameter-passing descendants from other oriented equations we denote them by $s \blacktriangleright t$ instead of $s \rhd t$.

In the programming literature a substitution θ is called a *strict* solution of an equation $s \approx t$ if $s\theta$ and $t\theta$ rewrite to the same ground term without defined symbols. We do not require groundness. We say that θ is a strict solution of an equation e if

- $e = s \rhd t$, $s\theta$ rewrites to $t\theta$ and $t\theta$ is a term without defined symbols, or
- $e = s \approx t$, and both $s\theta$ and $t\theta$ rewrite to a common term without defined symbols.

These notions are easily incorporated into the definition of conditional rewriting. So when a conditional rewrite rule $l \to r \Leftarrow c$ is applied with substitution θ, we require that θ is a strict solution of the equations in c. Hence, as far as semantics is concerned, we do not distinguish equations in the initial goal from equations in the conditional parts of the rewrite rules.

Now we are ready to define our main refinement, the calculus LCNC_d. This calculus is defined for the left-linear fresh deterministic CCSs, and consists of the inference rules shown below.

[o] *outermost narrowing*

$$\frac{f(s_1,\dots,s_n) \simeq t, G}{s_1 \blacktriangleright l_1,\dots,s_n \blacktriangleright l_n, c, r \simeq t, G} \quad \simeq \in \{\approx, \backsimeq, \rhd, \blacktriangleright\}$$

if $f(l_1,\dots,l_n) \to r \Leftarrow c$ is a fresh variant of a rewrite rule in $\mathcal{R}$,

[i] *imitation*

$$\frac{g(s_1,\dots,s_n) \simeq x, G}{(s_1 \simeq x_1,\dots,s_n \simeq x_n, G)\theta} \quad \simeq \in \{\approx, \backsimeq, \rhd\} \qquad \frac{f(t_1,\dots,t_n) \blacktriangleright x, G}{(t_1 \blacktriangleright x_1,\dots,t_n \blacktriangleright x_n, G)\theta'}$$

if $g \in \mathcal{F}_\mathcal{C}$, $g(s_1,\dots,s_n) \notin \mathcal{T}(\mathcal{F}_\mathcal{C},\mathcal{V})$, $\theta = \{x \mapsto g(x_1,\dots,x_n)\}$, and $\theta' = \{x \mapsto f(x_1,\dots,x_n)\}$ with $x_1,\dots,x_n$ fresh variables,

[d] *decomposition*

$$\frac{g(s_1,\dots,s_n) \blacktriangleright g(t_1,\dots,t_n), G}{s_1 \simeq t_1,\dots,s_n \simeq t_n, G} \quad \begin{array}{l}\simeq \in \{\approx, \rhd\}\\ g \in \mathcal{F}_\mathcal{C}\end{array} \qquad \frac{f(s_1,\dots,s_n) \blacktriangleright f(t_1,\dots,t_n), G}{s_1 \blacktriangleright t_1,\dots,s_n \blacktriangleright t_n, G}$$

[v] *variable elimination*

$$\frac{x \blacktriangleright s, G}{G\theta} \quad s \notin \mathcal{V} \qquad \frac{s \blacktriangleright x, G}{G\theta}$$

$$\frac{x \simeq s, G}{G\theta} \quad s \in \mathcal{T}(\mathcal{F}_\mathcal{C},\mathcal{V}) \setminus \mathcal{V} \qquad \frac{s \simeq x, G}{G\theta} \quad s \in \mathcal{T}(\mathcal{F}_\mathcal{C},\mathcal{V})$$

if $x \notin \mathcal{V}\text{ar}(s)$, $\simeq \in \{\approx, \rhd\}$, and $\theta = \{x \mapsto s\}$,

[t] *removal of trivial equations*

$$\frac{s \simeq s, G}{G} \quad \begin{array}{l}\simeq \in \{\approx, \rhd\}\\ s \in \mathcal{T}(\mathcal{F}_\mathcal{C},\mathcal{V})\end{array} \qquad \frac{s \blacktriangleright s, G}{G}$$

	$s \approx t$				$s \rhd t$				$s \blacktriangleright t$		
$root(s)\backslash^{\mathrm{root}(t)}$	$\mathcal{F}_{\mathcal{C}}$	$\mathcal{F}_{\mathcal{D}}$	$\mathcal{V}$		$\mathcal{F}_{\mathcal{C}}$	$\mathcal{F}_{\mathcal{D}}$	$\mathcal{V}$		$\mathcal{F}_{\mathcal{C}}$	$\mathcal{F}_{\mathcal{D}}$	$\mathcal{V}$
$\mathcal{F}_{\mathcal{C}}$	[t]; [d]	[o]$_2$	[v]; [i]	$\mathcal{F}_{\mathcal{C}}$	[t]; [d]	$\times$	[v]; [i]	$\mathcal{F}_{\mathcal{C}}$	[t]; [d]	$\times$	[v]
$\mathcal{F}_{\mathcal{D}}$	[o]$_1$	[t]; [o]$_1$	[o]$_1$	$\mathcal{F}_{\mathcal{D}}$	[o]$_1$	[t]; [o]$_1$	[o]$_1$	$\mathcal{F}_{\mathcal{D}}$	[o]$_1$	[t]	[v]
$\mathcal{V}$	[v]; [i]	[o]$_2$	[t]; [v]	$\mathcal{V}$	[v]	$\times$	[t]; [v]	$\mathcal{V}$	[v]	[v]	[v]

Table 1: LCNC$_{\mathrm{d}}$: Selection of inference rule.

The calculus LCNC$_{\mathrm{d}}$ imposes the selection strategy given in Table 1. Here the symbol ";" separates (groups of) rules in decreasing order of priority. The subscript (1 or 2) in [o] indicates to which side (left or right) of the selected equation the rule is applied.

The following theorem summarizes the main result of this paper.

Theorem 5 *Let $\mathcal{R}$ be a left-linear fresh deterministic CCS and θ a normalized strict solution of G. There exists an* LCNC$_{\mathrm{d}}$*-refutation $G \Rightarrow^*_{\theta'} \Box$ such that $\theta' \leqslant \theta$ $[\mathcal{V}\mathrm{ar}(G)]$.*

4.1 Proof of Theorem 5

We start by defining some useful concepts.

For a substitution θ and a set of variables X, we denote $(X \setminus \mathcal{D}(\theta)) \cup \mathcal{I}_X(\theta)$ by $\mathcal{V}\mathrm{ar}_X(\theta)$. Here $\mathcal{D}(\theta) = \{x \in \mathcal{V} \mid \theta(x) \neq x\}$ denotes the domain of θ, which is always assumed to be finite, and $\mathcal{I}_X(\theta) = \bigcup_{x \in \mathcal{D}(\theta) \cap X} \mathcal{V}\mathrm{ar}(x\theta)$ the set of variables introduced by the restriction of θ to X.

The proof of the following lemma is straightforward.

Lemma 6 *If G is an X-deterministic goal and σ a substitution then $G\sigma$ is $\mathcal{V}\mathrm{ar}_X(\sigma)$-deterministic.* $\Box$

For a CTRS $\mathcal{R}$ we let $\mathcal{R}_+ = \mathcal{R} \cup \{x \approx x \to \mathtt{true}, x \rhd x \to \mathtt{true}\}$. We use the variant of conditional rewriting in which the list of instantiated conditions of the applied rewrite rule is explicitly added to the goal after every rewrite step. Formally, we use the *intermediate* rewrite relation $\rightarrowtail$ defined as follows: $G \rightarrowtail G'$ if $G = G_1, e, G_2$, $G' = G_1, c\theta, e', G_2$, and $e \to_{\mathcal{R}} e'$ by applying the rewrite rule $l \to r \Leftarrow c \in \mathcal{R}_+$ with substitution θ. The equation e' is called the *(immediate) descendant* of e in the intermediate rewrite step. The notion of descendant is generalized to arbitrary intermediate rewrite derivations in the obvious way. It is well-known that $\mathcal{R} \vdash G$ if and only if $G \rightarrowtail^* \top$.

Definition 7 *A state is a quadruple $\langle G, \theta, \Pi, X \rangle$ consisting of a proper goal G, an X-normalized solution of G, and an intermediate rewrite sequence Π: $G\theta \rightarrowtail^* \top$. $\langle G, \theta, \Pi, X \rangle$ is* deterministic *if G is X-deterministic.*

The intended meaning of a state $\langle G, \theta, \Pi, X \rangle$ is as follows:

Given a goal G with X-normalized solution θ and rewrite proof $\Pi\colon G\theta \rightarrowtail^* \top$, find an LCNC$_\ell$-refutation $G \Rightarrow_\sigma^* \Box$ such that $\sigma \leqslant \theta$ $[\mathcal{V}\mathrm{ar}(G)]$.

To solve this problem, we apply proof steps called *state transformations.* Each proof step takes as input a state $S = \langle G, \theta, \Pi, X\rangle$ and a finite set of variables W, and yields a simpler state $S' = \langle G', \theta', \Pi', X'\rangle$ and an LCNC$_\ell$ step $\pi\colon G \Rightarrow_\sigma G'$ such that $\theta = \sigma\theta'$ $[W]$. We denote this operation by $\langle S', \sigma\rangle = \phi_{\mathrm{LCNC}_\ell}(S, W)$ and depict it as

$$S \underset{W}{\overset{\sigma}{\dashrightarrow}} S'.$$

The notion of "simpler" is captured by the well-founded relation $\gg$ on states given by [9, Definition 5].. In this setting, a successful transformation is a finite sequence of proof steps

$$\langle G, \theta, \Pi, X\rangle \underset{W_1}{\overset{\sigma_1}{\dashrightarrow}} \langle G_1, \theta_1, \Pi_1, X_1\rangle \underset{W_2}{\overset{\sigma_2}{\dashrightarrow}} \cdots \underset{W_n}{\overset{\sigma_n}{\dashrightarrow}} \langle \Box, \theta_n, \Pi_n, X_n\rangle$$

which we abbreviate to

$$\langle G, \theta, \Pi, X\rangle \underset{W_1, W_2, \ldots, W_n}{\overset{\sigma_1\sigma_2\cdots\sigma_n}{\dashrightarrow}} \langle \Box, \theta_n, \Pi_n, X_n\rangle$$

where the sets $W_1, \ldots, W_n$ are chosen in such a way that $\theta = \sigma_1\sigma_2\cdots\sigma_n\theta_n$ $[\mathcal{V}\mathrm{ar}(G)]$. Then, by concatenating the LCNC$_\ell$-steps obtained along the transformation, we obtain the LCNC$_\ell$-refutation $G \Rightarrow_\sigma^* \Box$ with $\sigma = \sigma_1\sigma_2\cdots\sigma_n$, providing an answer to the problem posed by the initial state $\langle G, \theta, \Pi, X\rangle$.

We recall the notion of standard intermediate rewrite sequence. In the following we use $Pos_{\mathcal{F}}(t)$ to denote the set of non-variable positions in the term t.

Definition 8 *An intermediate rewrite sequence $G \rightarrowtail^* \top$ is called* standard *if $q\backslash p \in \mathcal{P}os_{\mathcal{F}}(l')$ whenever $G \rightarrowtail^* \top$ can be written as follows:*

$$G \rightarrowtail^* \top, e, G' \rightarrowtail_p \top, c\theta, e[r\theta]_p, G' \rightarrowtail^* \underbrace{\top, e[r\theta]_p, G' \rightarrowtail^* \top, e', G'}$$
$$\rightarrowtail_q \top, c'\theta', e'[r'\theta']_q, G' \rightarrowtail^* \top$$

where $l \to r \Leftarrow c$ and $l' \to r' \Leftarrow c'$ are the rewrite rules used in the $\rightarrowtail_p$ and $\rightarrowtail_q$-steps, such that $\epsilon < q < p$ and in the underbraced part no rewrite rule is applied to a position above p.

Theorem 9 *Let $\mathcal{R}$ be a left-linear fresh CTRS and G a goal. Every intermediate rewrite sequence $G \rightarrowtail^* \top$ can be transformed into a standard sequence from G to $\top$.*

The proof is essentially the same as Suzuki's proof of standardization for left-linear join CTRSs [13, Theorem 5.10]. The only complication is that we must show that the relation "$\Rightarrow$" for eliminating so-called anti-standard pairs is well-defined, which amounts to proving that rewriting can be performed at positions which are descendants of positions below the pattern position of a $\rightarrowtail$-step, cf. [13, Definition 5.3]. This follows from the simple observation that in fresh CTRSs there are no such descendants in the right-hand sides of oriented equations in the conditions. $\Box$

Definition 10 *A state $S = \langle G, \theta, \Pi, X \rangle$ is called standard if Π is standard and for every parameter-passing descendant $e = s \blacktriangleright t$ in G the following property holds: if a descendant of $e\theta$ is rewritten at position $1p$ and subsequent descendants are not rewritten at a position q with $\epsilon < q < 1p$ then $p \in \mathcal{P}os_{\mathcal{F}}(t)$. S is called* strict *if all parameter-passing descendants in G are of the form $s \blacktriangleright t$ with t a linear constructor-based term.*

Due to space restrictions, we omit the rather long but boring proof of the following lemma.

Lemma 11 *There exists a state transformation ϕ_{LCNC_d} which, for every standard strict state $S = \langle G, \theta, \Pi, X \rangle$ with $G \neq \Box$ and finite set of variables W yields a pair $\langle S', \pi \rangle$ of a standard state $S' = \langle G', \theta', \Pi', W' \rangle$ and an* LCNC$_d$*-step $\pi : G \Rightarrow_\sigma G'$ such that $\theta = \sigma\theta'$ $[W]$ and $W' = \mathcal{I}_W(\sigma)$.* □

Lemma 12 *If $S = \langle G, \theta, \Pi, X \rangle$ is a standard strict state such that $\langle S', W' \rangle = \phi_{\text{LCNC}_d}(S, W)$ then S' is a standard strict state too.*

We say that a goal G satisfies property $\mathcal{Q}_{strict}$ if all parameter-passing descendants in G are of the form $s \blacktriangleright t$ with t a linear constructor-based term.

It is sufficient to prove that property $\mathcal{Q}_{strict}$ is preserved by LCNC$_d$-steps. This follows immediately from a proof by case distinction. □

With this proviso, we are ready to give the proof of Theorem 5, i.e., that if θ is a strict solution of G, then there exists an LCNC$_d$-refutation $\Psi : G \Rightarrow^*_{\theta'} \Box$ such that $\theta' \leq \theta$ $[\mathcal{V}(G)]$.

Let $\mathcal{R}$ be a left-linear fresh deterministic CCS, θ a strict solution of G, and $S = \langle G, \theta, \Pi, X \rangle$ a corresponding state. By Theorem 9 we can assume that Π is a standard rewrite proof, and thus that S is a standard state. Since G is an initial goal, the state S is strict too. The construction of Ψ proceeds by induction w.r.t. the well-founded order $\gg$ on states. To make the induction work we prove $\theta' \ll \theta$ $[W]$ for a finite set of variables W. In the base case when $G = \Box$ we take Ψ the empty refutation. If $G \neq \Box$ we perform the following construction

$$\underbrace{\langle G, \theta, \Pi, X \rangle \overset{\sigma_1}{\underset{W}{\dashrightarrow}} \langle G_1, \theta_1, \Pi_1, X_1 \rangle}_{\text{Lemma 11}} \underbrace{\overset{\sigma_2}{\underset{W_1,\ldots}{\dashrightarrow}} \langle \Box, \theta_2, \Pi_2, X_2 \rangle}_{\text{induction hypothesis}}$$

where $W_1 = W \cup \mathcal{I}_W(\sigma_1)$ and $\sigma_2 \leqslant \theta_1$ $[W_1]$. Let $\theta' = \sigma_1\sigma_2$. Clearly $\theta' \leqslant \sigma_1\theta_1 = \theta$ $[W]$. In this way we obtain the LCNC$_d$-refutation $\Psi : G \Rightarrow_{\sigma_1} G' \Rightarrow^*_{\sigma_2} \Box$. □

5 Benchmarks

We compare the three calculi LCNC$_\ell$ and LCNC$_d$ on a small number of examples. In Table 2 we use the CTRS for computing Fibonacci numbers given in Section 3 and the goal $G = \mathsf{fib}(x) = \langle x, y \rangle$, which admits the two normalized solutions $\{x \mapsto 0, y \mapsto \mathsf{s}(0)\}$ and $\{x \mapsto \mathsf{s}(0), y \mapsto \mathsf{s}(0)\}$,

Table 2: The goals $\mathsf{fib}(x) = \langle x, y\rangle$ and G_n for $0 \leqslant n \leqslant 4$.

	LCNC$_\ell$		LCNC$_\text{d}$	
depth	N	R	N	R
5	27	0	14	0
6	39	2	17	1
14	743	24	65	2
21	10464	92	129	2
22	16087	92	140	2

	LCNC$_\ell$		LCNC$_\text{d}$	
n	N	R	N	R
0	12	1	8	1
1	38	4	16	2
2	92	11	24	3
3	202	26	32	4
4	424	57	40	5

Table 3: The goals H_1 and H_2.

H_1	LCNC$_\ell$		LCNC$_\text{d}$		H_2	LCNC$_\ell$		LCNC$_\text{d}$	
depth	N	R	N	R	depth	N	R	N	R
10	172	8	25	0	10	205	12	25	0
15	982	8	43	0	15	1740	18	57	0
20	4710	8	65	0	20	18332	18	93	0
23	8438	8	68	0	23	61640	18	125	0
59	?	?	160	1	94	?	?	325	1

as well as the goals $G_n = x + y \approx \mathsf{s}^n(0)$ for $0 \leqslant n \leqslant 4$. The search trees for the goals G_n are finite for all three calculi. Since the search tree for G is infinite, we give the number of nodes (N) and number of refutations (R) for given depths.

Next we consider the following specification of the quicksort algorithm:

$$\begin{array}{rcl}
\mathsf{app}([\,], y) & \to & y \\
\mathsf{app}([x|y], z) & \to & [x|\mathsf{app}(y, z)] \\
0 + y & \to & y \\
\mathsf{s}(x) + y & \to & \mathsf{s}(x + y) \\
\mathsf{split}(x, [y|z]) & \to & \langle u, [y|v]\rangle \Leftarrow \mathsf{split}(x, z) \rhd \langle u, v\rangle,\ (x \leq y) \rhd \mathsf{t} \\
\mathsf{split}(x, [y|z]) & \to & \langle [y|u], v\rangle \Leftarrow \mathsf{split}(x, z) \rhd \langle u, v\rangle,\ (x \leq y) \rhd \mathsf{f} \\
\mathsf{qsort}([\,]) & \to & [\,] \\
\mathsf{qsort}([x|y]) & \to & \mathsf{app}(\mathsf{qsort}(u), [x|\mathsf{qsort}(v)]) \Leftarrow \mathsf{split}(x, y) \rhd \langle u, v\rangle
\end{array}
\qquad
\begin{array}{rcl}
0 \leq y & \to & \mathsf{t} \\
\mathsf{s}(x) \leq 0 & \to & \mathsf{f} \\
\mathsf{s}(x) \leq \mathsf{s}(y) & \to & x \leq y \\
\mathsf{split}(x, [\,]) & \to & \langle [\,], [\,]\rangle
\end{array}$$

Note that these rules constitute a left-linear fresh deterministic CCS. We consider the goals $H_1 = \mathsf{qsort}([\mathsf{s}(0), 0]) \rhd x$ and $H_2 = \mathsf{qsort}([\mathsf{s}(0), 0, \mathsf{s}(\mathsf{s}(0))]) \approx x$. The results are shown in Table 3. Both H_1 and H_2 admit a single strict normalized solution as well as numerous non-normalized solutions. The two search trees for LCNC$_\text{d}$ are finite.

6 Conclusion and Future Research

We have identified a class of non-necessarily terminating CTRSs, the left-linear fresh deterministic CCSs, for which there exists a deterministic and complete lazy narrowing calculus

LCNC$_d$. This class is completely characterized by syntactic restrictions which can be easy to check and impose on functional logic programs. Determinism was introduced by Ganzinger [2] for oriented CTRS, and has proved to be very useful for the study of the (unique) termination behavior of well-moded Horn clause programs (cf. [10]). We have generalized the notion of determinism to the class of arbitrary CTRSs and identified a syntactic condition (freshness) which guarantees standardization for left-linear deterministic CTRSs (Theorem 9). Left-linear constructor-based TRSs are widely accepted in functional logic programming, and the freshness condition arises quite natural.

The work reported in this paper is a continuation of our efforts to identify suitable computational models for functional logic programming. Our previous research was targeted more to higher-order extensions of lazy narrowing [4, 6]. We claim that the refinement proposed here can be carried over to a conditional extension of the higher-order lazy narrowing calculus proposed in [6] without losing completeness.

References

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

[2] H. Ganzinger. Order-sorted completion: The many-sorted way. *Theoretical Computer Science*, 89:3–32, 1991.

[3] J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. Polymorphic types in functional logic programming. *Journal of Functional and Logic Programming*, 2001(1), 2001.

[4] M. Marin, T. Ida, and T. Suzuki. On Reducing the Search Space of Higher-Order Lazy Narrowing. In A. Middeldorp and T. Sato, editors, *FLOPS'99*, volume 1722 of *LNCS*, pages 225–240. Springer-Verlag, 1999.

[5] M. Marin and A. Middeldorp. New completeness results for lazy conditional narrowing. In *Sixth International Workshop on Unification (UNIF 2002)*, Copenhagen, Denmark, 2002.

[6] M. Marin, T. Suzuki, and T. Ida. Refinements of lazy narrowing for left-linear fully-extened pattern rewrite systems. Technical Report ISE-TR-01-180, Institute of Information Sciences and Electronics, University of Tsukuba, Japan, 2001.

[7] A. Middeldorp and S. Okui. A deterministic lazy narrowing calculus. *Journal of Symbolic Computation*, 25(6):733–757, 1998.

[8] A. Middeldorp, S. Okui, and T. Ida. Lazy narrowing: Strong completeness and eager variable elimination. *Theoretical Computer Science*, 167(1,2):95–130, 1996.

[9] A. Middeldorp, T. Suzuki, and M. Hamada. Complete selection functions for a lazy conditional narrowing calculus. *Journal of Functional and Logic Programming*, 2002(3), March 2002.

[10] E. Ohlebusch. Termination of logic programs: Transformational methods revisited. *Applicable Algebra in Engineering, Communication and Computing*, 12:73–116, 2001.

[11] E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002.

[12] C. Prehofer. *Solving Higher-Order Equations: From Logic to Programming*. Progress in Theoretical Computer Science. Birkäuser, 1998.

[13] T. Suzuki. Standardization theorem revisited. In *Proceedings of the 5th International Conference on Algebraic and Logic Programming*, volume 1139 of *LNCS*, pages 122–134, 1996.

[14] T. Yamada, J. Avenhaus, C. Loría-Sáenz, and A. Middeldorp. Logicality of conditional rewrite systems. *Theoretical Computer Science*, 236(1,2):209–232, 2000.

Equational Reasoning in Programming

Tetsuo Ida
Institute of Information Sciences and Electronics
University of Tsukuba
Tsukuba, 305-8573, Japan
email: ida@score.is.tsukuba.ac.jp

Abstract
Equality plays an important role in our life, and we practise equational reasoning everyday. We can take advantage of our ability of reasoning with equalities and make explicit the equational reasoning in programming and symbolic computation. Based on this observation we developed an equational programming system called CFLP (Constraint Functional Logic Programming system). We present various examples to show the importance of equations in programming.

1. Introduction

Modern life demands a certain level of mathematical maturity. One of the most important is the ability to reason with equalities. We do sophisticated reasoning with equalities in arithmetic in everyday life, although we are often unaware of it.

Let us take a concrete example. Suppose we buy an item priced at 975 yen. Since in Japan consumer tax is 5 %, we have to pay over 1000 yen. We calculate the exact amount of money that we have to pay, while at the same time we fumble in the pocket and try to find appropriate coins and notes so as to avoid filling up the pocket with many coins of change. We may decide to give a note of 5000 yen and three 10 yen coins. How do we decide?

The involved reasoning is complex; it is not mere simplification of numerical expressions. It involves equational reasoning. In the above example we need at least 6 steps of equational reasoning even if integer arithmetic is assumed. Most people can somehow perform this kind of mathematics. Indeed they master it at a fairly early age. They can comfortably handle reflexivity, symmetry and transitivity of equality relation defined over the domain of numbers.

In this paper we will show that reasoning with equality over various domains of objects is also important, and easy to practise if we are provided with appropriate tools for reasoning. One particular example that we are interested in here, and is relevant to mathematics education, is programming. All the programming examples including these texts are in *Mathematica* Notebook and executable.

2. Equality in Programming

We will first show a very basic programming system, a subset of *Mathematica*, and extend it to a system for equational programming. We begin by defining terms with which we construct a program. Our vocabulary $\mathcal{S}$, called *signature*, is given as a set of symbols $\{0, \mathrm{s}, \oplus, \otimes\}$. The symbols 0, s, $\oplus$ and $\otimes$ take 0, 1, 2 and 2 arguments, respectively. These numbers are called arity. Furthermore we will use unlimited number of variables, say $x, y, z, \ldots$. Then the syntax of terms is as follows:

0 is a term.

x is a term if x is a variable.

$f[t_1, \ldots, t_n]$ is a term if $t_1, \ldots, t_n$ are terms where n is the arity of f and $f \in \mathcal{S}$.

Nothing other than those constructed in this way is a term.

At this point we only have syntactic equality, i. e. terms are equal only if they look the same. Really interesting things emerge when we introduce rewriting rules on the domain of the terms. We define rewrite rules for the symbols $\oplus$ and $\otimes$. For readability we use those symbols as infix operators. The rewrite rules are specified in the following way in *Mathematica.*

■ Rewrite rules

```
0⊕y_ := y;
s[x_]⊕y_ := s[x⊕y] ;
0⊗y_ := 0;
s[x_]⊗y_ := (x⊗y)⊕y ;
```

In order to distinguish variables from non-variable symbols, *Mathematica* has the convention that the variables on the left hand side of the rewrite rule are marked by _ (underscore).

These four lines of rewrite rules define a term rewrite system $\mathcal{R}$. $\mathcal{R}$ induces the reduction relation $\rightarrow_{\mathcal{R}}$.

This means that for any terms s and t in $s := t \in \mathcal{R}$ and any substitution θ,
$s\,\theta \rightarrow_{\mathcal{R}} t\,\theta$ and for any context $C[\ ]$, $C[u] \rightarrow_{\mathcal{R}} C[v]$, if $u \rightarrow_{\mathcal{R}} v$.

Here, context $C[\ ]$ denotes a term that has a hole to be filled by some term.

The reflexive and transitive closure of $\rightarrow_{\mathcal{R}}$ is denoted by $\rightarrow^*_{\mathcal{R}}$. The reflexive, transitive and symmetric closure of $\rightarrow_{\mathcal{R}}$ is denoted by $\leftrightarrow^*_{\mathcal{R}}$. Finally we identify equality $==_{\mathcal{R}}$ with $\leftrightarrow^*_{\mathcal{R}}$.

A purely functional language is based on the notion of rewriting terms by $\rightarrow_{\mathcal{R}}$. The term which is no longer related by $\rightarrow_{\mathcal{R}}$ to any term is called a ($\rightarrow_{\mathcal{R}}$) normal form. For example, we see the term `s[s[0]]⊗s[s[s[0]]]` is reduced to its normal form.

```
s[s[0]]⊗s[s[s[0]]]

s[s[s[s[s[s[0]]]]]]
```

When $\mathcal{R}$ is understood from the context, we will drop subscript $\mathcal{R}$ in the above relations.

Handling equality is more difficult in general than reduction. To see if $s ==_{\mathcal{R}} t$, we have to show that s and t are related by $\rightarrow_{\mathcal{R}}$ via several terms. We do not know in advance in what way we should rewrite s and t so that the rewrites eventually lead to the same term. This seems a formidable problem, unless we know certain properties of the term rewrite system $\mathcal{R}$.

Furthermore, equality in programming is discussed in a more challenging context. Instead of proving $\forall x_1,\ldots,x_n.\ s ==_{\mathcal{R}} t$ as above, we want to prove $\exists x_1,\ldots,x_n.\ s ==_{\mathcal{R}} t$, where we are interested not only in the truth of the statement, but a substitution θ that makes $s\theta ==_{\mathcal{R}} t\theta$. The computation to find the substitution is called *solving* in this paper.

In *Mathematica* we have a special function called `Solve` which computes such substitutions.

▪ Solving equations: Cranes and Tortoise (cats and birds) problem

There are 32 legs and 10 heads of tortoises and cranes. The number of tortoises and cranes is found by applying `Solve` to equations:

```
Solve[{cranes + tortoises == 10, 2 cranes + 4 tortoises == 32},
  {cranes, tortoises}]
```

```
{{cranes → 4, tortoises → 6}}
```

We obtain the substitution {cranes→4, tortoises→6} as the answer.

▪ Solving equations on the domain of terms

Then, we will try to solve a similar problem over the domain of terms that we have defined.

```
Solve[{x⊕y == s[s[s[0]]]}, {x, y}]

Solve::dinv :
 The expression x⊕y involves unknowns in more than one
   argument, so inverse functions cannot be used.

Solve[{x⊕y == s[s[s[0]]]}, {x, y}]
```

Mathematica does not give solutions that we would expect. Readers are invited to find the reason why it does not give the solutions.

An important to note is that the symbols that we are using should be uninterpreted, i.e. they should be used as mere symbols. The symbol 0, for instance, is not an integer zero in this setting. We do not assume any mathematical properties on the symbols except that we define rewrite relations and equality induced by $\to_{\mathcal{R}}$. The situation is very different from the case of solving linear equations.

3. Narrowing

Fortunately, we already have a method called *narrowing* for solving equations over the domain of terms. Narrowing is a procedure to prove existentially quantified equations by presenting values that bind the existential variables Formally, given a rewrite system $\mathcal{R}$ and a sequence of equations $s_1 == t_1, \ldots, s_n == t_n$, by narrowing we can compute a substitution θ such that $s_k\theta ==_{\mathcal{R}} t_k\theta$ for $k = 1, \ldots, n$. The basic idea is similar to rewriting; rewrite the terms of both sides of an equation repeatedly until they become the same term using the rewrite rules and the substitutions. In narrowing, the substitutions are used not only to substitute terms for the variables in rewrite rules, but also for the variables in equations to be solved.

This method can be formalized as a calculus, which we call *lazy narrowing calculus*. The calculus is called *lazy* because we incorporate in the calculus an algorithm for systematically identify and rewrite a certain preferred parts of equations. The lazy narrowing calculus is the core of the interpreter of the programming language, which we discuss in the next section.

4. Language for solving equations

We now define the language for our equational programming. The signature consists of two disjoint sets of function symbols; the set of constructors and the set of defined function symbols. In our previous example, s is a constructor symbol, and $\oplus$ and $\otimes$ are defined function symbols. The defined function symbols are associated with rewrite rules, whereas the constructor symbols are not. The syntax of the terms is as given before except that they are (sometimes implicitly) typed. With that syntax we will represent a simply typed λ-terms, e.g. $\lambda[\{x, y\}, \mathrm{plus}[x, y]]: \mathrm{int} \to \mathrm{int} \to \mathrm{int}$.

The equations to be solved is often called a *goal*. To avoid confusion of a *Mathematica*'s built-in equation $s == t$, we hereafter denote our equation by $s \approx t$.

The program is given by a higher-order rewrite system called pattern rewrite system. With higher-order rewrite system we can treat functions systematically. Moreover, we can find higher-order solutions, i.e. functions, wherever possible.

A pattern rewrite system is a set of unconditional rewrite rules

$$f[t_1, \ldots, t_n] \to t,$$

or conditional rewrite rules

$$f[t_1, \ldots, t_n] \to t \Leftarrow E.$$

We have certain restrictions on the syntactic structure and the types of the terms that can be used to form a pattern rewrite system (See [1] for technical details).

5. Solving equations over the domain of terms

■ Solving first-order equations

We return to the problem of solving the equation in Section 2 with our system [2]. The system is called Constraint Functional Logic Programming system (CFLP for short). CFLP is implemented as a package of *Mathematica*. After loading CFLP package, we start a CFLP session by declaring the signature. ℤ denotes some first-order basic type, in this case type integer.

```
<< FrontendCFLP.m
```

```
DataConstructor[s : ℤ → ℤ]
```

`CirclePlus` and `CircleTimes` are actual function names of ⊕ and ⊗, respectively.

```
(* We clear previous definitions
  about CirclePlus and CircleTimes *)
Clear[CirclePlus, CircleTimes];
```

```
DefinedSymbol[CirclePlus : ℤ → ℤ → ℤ, CircleTimes : ℤ → ℤ → ℤ]
```

The following is our program assigned to a variable R. All the rewrite rules in this example turn out to be unconditional. In CFLP we have to declare all function symbols. Hence the other symbols are automatically recognized as variables. So we do not have to mark variables with _.

```
R = FLPProgram[{
     0⊕y → y, s[x]⊕y → s[x⊕y],
     0⊗y → 0, s[x]⊗y → (x⊗y)⊕y}];
```

This is the goal to be solved.

```
G1 = {exists[{X, Y}, X⊕Y ≈ s[s[s[0]]]]}

{∃{X:ℤ,Y:ℤ} X⊕Y ≈ s[s[s[0]]]}
```

We specify the solver to solve the above goal. Our system is designed to be general, in that we can also specify solvers for the problem and the strategy to apply various solvers. We omit the explanation of the following two lines of program, as we discuss the solver collaboration in later sections. The reader can see that we use Lazy Narrowing solver (LNSolver) for solving our problem. Actually, the lazy narrowing solver consists of several narrowing calculi, and we will use the one called LCNCd tailored to the first-order solving.

```
(* solver *)
flp = MkLocalSolver["LNSolver`"];
```

```
ConfigSolver[flp, {Program → R, Calculus -> "LCNCd"}]
```

Computation, i.e., solving the equation, starts when we apply the solver LNSolver to the goal.

```
ApplyCollaborative[flp, G1]
```

Then LNSolver yields the following solution:

```
LNSolver` yields
```

```
{{X → 0, Y → s[s[s[0]]]}, {X → s[0], Y → s[s[0]]},
  {X → s[s[0]], Y → s[0]}, {X → s[s[s[0]]], Y → 0}}
```

We can ask ourselves the following questions. Are these correct? Are these all the solutions? The former is concerned with so-called soundness, and the latter with completeness. There are several ways that we can convince ourselves of the affirmative of these questions. Concerning the soundness, we apply the substitutions to the goal, and rewrite both sides of the equation. In our case the right hand side is already a normal form. So all we have to do is to reduce the left hand side of the equation. Another way of convincing ourselves is by a model. Interpret 0 as the natural number zero, s as a function of increment by one over the natural numbers. We will see that $\oplus$ and $\otimes$ are addition and multiplication operators on natural numbers. Hence the goal is actually $x + y = 3$, and we solve for x and y in the domain of natural numbers. It is easy to see (by simple enumeration) that we have 4 solutions, i.e., (x, y) = (0,3), (2,1), (1,2), (3,0). These correspond to the solutions that we obtained.

Proving soundness and completeness of the calculus in general setting requires deeper investigation [3].

■ Solving higher-order equations

The next example shown below in G2 is more difficult. It involves higher-order, i.e. function, variables. Even in high school mathematics, we often encounter higher-order mathematical objects, and equations involving higher-order objects. However, mostly problems are restricted to obtain first-order quantities. In the following example, we want to solve for higher-order variables, i.e. we want to obtain a function as a solution.

As in the previous example, *Mathematica* is able to give solutions of certain kind of equations for certain domains. For example, *Mathematica*'s built-in function `DSolve` gives solutions to differential equations.

```
DSolve[f'[x] == x, {f}, {x}]
```

$$\left\{\left\{f \to \text{Function}\left[\{x\},\ \frac{x^2}{2} + C[1]\right]\right\}\right\}$$

In the domain of terms, however, this is not the case. Solving for higher-order variables is non-trivial task. For certain class of programs, we do have a method for solving equations over the domain of terms. The method is generically called *higher-order narrowing.* As in the first-order case, we can formalize the higher-order narrowing as a narrowing calculus. So in this case let us use higher-order lazy narrowing calculus HOLN (Higher-Order Lazy Narrowing calculus). Our problem is as follows:

```
G2 = {exists[{F, Y : ℤ → ℤ},
    λ[{x}, F[x, Y[x]]] ≈ λ[{x}, x⊕s[s[s[0]]]]]}

{∃(F:ℤ→ℤ→ℤ, Y:ℤ→ℤ) λ[{x : ℤ}, F[x, Y[x]]] ≈ λ[{x : ℤ}, x⊕s[s[s[0]]]]}
```

We want to solve this equation for higher-order variables. For those who are not familiar with the lambda calculus, read λ as Function.

```
holn = MkLocalSolver["LNSolver`"];
ConfigSolver[holn, {Program → R, Calculus -> "HOLN"}]
```

```
ApplyCollaborative[holn, G2]
```

Then, CFLP returns the following solutions.

```
LNSolver` yields
```

```
{{F→λ[{x1 : ℤ, x2 : ℤ}, x1⊕s[s[s[0]]]]},
 {F→λ[{x1 : ℤ, x2 : ℤ}, x1⊕s[s[s[x2]]]], Y→λ[{x57 : ℤ}, 0]},
 {F→λ[{x1 : ℤ, x2 : ℤ}, x1⊕s[s[x2]]], Y→λ[{x45 : ℤ}, s[0]]},
 {F→λ[{x1 : ℤ, x2 : ℤ}, x1⊕s[x2]], Y→λ[{x31 : ℤ}, s[s[0]]]},
 {F→λ[{x1 : ℤ, x2 : ℤ}, x1⊕x2], Y→λ[{x19 : ℤ}, s[s[s[0]]]]},
 {F→λ[{x1 : ℤ, x2 : ℤ}, x2⊕s[s[s[0]]]], Y→λ[{x9 : ℤ}, x9]},
 {F→λ[{x3 : ℤ, x4 : ℤ}, x4], Y→λ[{x7 : ℤ}, x7⊕s[s[s[0]]]]}}
```

Let us take a closer look at the second solution: {`F→λ[{x1:ℤ,x2:ℤ}, x1⊕s[s[s[x2]]]], Y→λ[{x57:ℤ},0]`}. In order to solve the higher-order equation, the system needs type information. The solution is attached types by the system. Let us ignore the types temporarily and describe the answer in the more familiar mathematical representation.

```
F[x, y] =x⊕s[s[s[y]]]
    Y[x] = 0
```

It is easy to see that those are indeed solutions.

6. Solving equations over various domains

Many scientific problems are modeled as a set of equations. Specialized algorithms have been developed for solving various equations over various domains. The domain of terms that we discussed in the previous sections is very important since it is in this domain that equational programs are treated. Equations and rewrite rules are regarded as programs. These programs are different from those of procedural programming languages such as JAVA and C.

The next challenge is whether we can combine several solvers to make a single framework in which specific solvers are called for specific problems automatically. CFLP is actually designed to work in this way. CFLP is at present equipped with four solvers including LNSolver. The system coordinates those solvers to work on a given goal. A programmer can either use a default combination of the solvers or can program the collaboration of the solvers using a simple coordination language.

Below we explicitly declare solvers and put the references to solvers in variables `holn`, `elim`, `deriv` and `polyn`. The latter three variables hold the references to the solver for a system of linear equations which implements Gaussian elimination method, the solver for partial and differential equations, and the solver for general polynomial equations which implements Gröbner basis algorithm, respectively.

```
(* solvers *)
holn = MkLocalSolver["LNSolver`"];
elim = MkLocalSolver["ElimSolver`"];
deriv = MkLocalSolver["DerivSolver`"];
polyn = MkLocalSolver["PolynSolver`"];
```

Using these elementary solvers we can define a new solver that combines these solvers. The following defines a new solver which applies solvers LNSolver, ElimSolver, DerivSolver and PolynSolver sequentially (seq) in this order. This application is repeated until the goal becomes fixed point (which means the goal is solved).

```
newSolver = repeat[seq[{holn, elim, deriv, polyn}]];
```

Finally we apply the new solver to the goal

```
ApplyCollaborative[newSolver, G2];
```

Of course, newSolver returns the same solution in this case since solvers other than LNSolver does not change the goal. We will see in the next section that the collaboration of solvers can solve more sophisticated problems.

7. Examples from geometry

Our final examples are taken from elementary geometry. We give these examples since many geometrical properties are stated declaratively, i.e., without resort to describing a concrete method to realize those propertied. If declarative statements are given in equations, it is easy to make them run on the computer. Those statements can be regarded programs.

Let us consider parallelism of lines. We first represent a line $a\,x + b\,y + c\ = 0$ by `line[`a, b, c`]` using constructor `line`. The following one taught in a high school is easy to see.

```
line[a1,b1,c1] ∥ line[a2,b2,c2] → True ⇐ a1 b2 - a2 b1 ≈ 0
```

It says that two lines are parallel if the coefficients of the equations of each line satisfies `a1 b2 - a2 b1` ≈ 0. In our language we can omit "`→ True`", and write simply

```
line[a1,b1,c1] ∥ line[a2,b2,c2] ⇐ a1 b2 - a2 b1 ≈ 0
```

We will give the definitions of several geometric functions below. Our language requires function declarations with types. In this paper we omit the explanation, but the declaration is necessary for our examples to run.

```
Constructor[TyLine[α] = line[α, α, α]];
Constructor[TyPoint[α] = point[α, α]];
Constructor[TySegment[α] = segment[TyPoint[α], TyPoint[α]]]
```

```
Instance[{α : Reals} ⇒ TyLine[α] : Eq];
Instance[{α : Reals} ⇒ TyPoint[α] : Eq];
Instance[{α : Reals} ⇒ TySegment[α] : Eq]
```

```
DefinedSymbol[
 DoubleVerticalBar : TyLine[ℝ] → TyLine[ℝ] → 𝔹,
 UpTee : TyLine[ℝ] → TyLine[ℝ] → 𝔹,
 DownRightVector : TyPoint[ℝ] → TyLine[ℝ] → 𝔹,
 SegmentToLine : TySegment[ℝ] → TyLine[ℝ],
 PerpendicularBisector :
  TyPoint[ℝ] → TyPoint[ℝ] → TyLine[ℝ],
 MidPoint : TyPoint[ℝ] → TyPoint[ℝ] → TyPoint[ℝ],
 BrTh : TyPoint[ℝ] → TySegment[ℝ] → TyPoint[ℝ] → TyLine[ℝ]
]
```

Then we have the following function definitions.

```
R2 = FLPProgram[{
     line[a1, b1, c1] ∥ line[a2, b2, c2] ⇐ a1 b2 - a2 b1 ≈ 0,
     line[a1, b1, c1] ⊥ line[a2, b2, c2] ⇐ a1 a2 + b1 b2 ≈ 0,
     (point[x, y] ⇁ line[a, b, c]) ⇐ a x + b y + c ≈ 0,
     SegmentToLine[segment[point[x1, y1], point[x2, y2]]] →
      line[y2 - y1, x1 - x2, x2 y1 - x1 y2],
     MidPoint[point[px, py], point[qx, qy]] →
      point[(px + qx) / 2, (py + qy) / 2],
     PerpendicularBisector[P, Q] → m ⇐
      {m ⊥ SegmentToLine[segment[P, Q]], MidPoint[P, Q] ⇁ m},
     BrTh[P, seg, Q] → n ⇐ {PerpendicularBisector[P, R] ≈ n,
       Q ⇁ n, R ⇁ SegmentToLine[seg]}}
  ];
```

```
ConfigSolver[flp, {Program → R2, Calculus → "LCNCd"}];
```

The functions ⊥, ⇁ , `MidPoint`, `SegmentToLine`, `PerpendicularBisector` are defined as auxiliary functions to function `BrTh`. Similar to the parallelism of lines, the condition of the perpendicularity of two lines are asserted by :

`line[a1,b1,c1]⊥ line[a2,b2,c2]  ⇐ a1 a2 + b1 b2 ≈ 0`

We next represent by `Point` [*x*, *y*] a point whose *x*- and *y*-coordinates are *x* and *y* respectively. The fact that point *P* is on line *m* is expressed by the notation *P* ⇁ *m*.

We represent a segment whose end points are `Point`[*x1*, *y1*] and `Point`[*x2*, *y2*] by `Segment[-Point`[*x1*, *y1*], `Point`[*x2*,*y2*]]. Then the program `SegmentToLine` which transforms a segment to a line is given as above.

`PerpendicularBisector`[*P*, *Q*] returns a line that bisects and is perpendicular to `Segment`[*P*, *Q*].

After the definition of the program, we can issue a question like:

```
G3 = exists[{l}, {line[2, 3, 5] ⊥ l}];
```

```
ApplyCollaborative[seq[{flp, elim}], {G3}]
```

Then the system returns the following answer.

```
ApplyCollaborative[seq[{flp, elim}], {G3}]

LNSolver` yields
```

```
{∃{a2$2268:R,b2$2268:R} {l → line[a2$2268, b2$2268, c2$2268]} &&
   2 a2$2268 + 3 b2$2268 == 0}
```

```
ElimSolver` yields
```

```
                 3 b2$2268
{{l → line[- ─────────── , b2$2268, c2$2268]}}
                     2
```

Note that a2$2268, b2$2268 and c2$2268 are internally generated variables.

With these preparations we can give a program of one of six basic Origami folds, known as Huzita's axioms [4, 5]. The axiom (O5) says that given two points *P* and *Q*, and a line *m* , we can make a fold that places *P* onto *m* and passes through *Q*.

BrTh[*P*, *s*, *Q*] computes a line *n* that passes through *Q*, such that the fold along the line *n* brings *P* onto *s*. BrTh is read as "bring *P* onto *s* along the line through *Q*".

We will try to solve the goal defined below:

```
G4 = exists[{l}, l ≈ BrTh[point[3 / 2, 1 / 2],
     segment[point[0, 0], point[2, 2]], point[1, -2]]]
```

```
ApplyCollaborative[seq[{flp, polyn}], {G4}]
```

We finally obtain the solutions of the goal.

```
{{l → line[0, 0, 0]}, {l → line[a1$2625, 0, -a1$2625]},
              3 b1$2625              b1$2625
 {l → line[ ─────────── , b1$2625, ─────────── ]}}
                 2                      2
```

In this example, we have three solutions.

8. Conclusion

In this paper we have shown the following:

- Equality plays an important role in programming.
- Equality can be defined in many domains, but the equality defined over the domain of terms is essential one in programming.
- Many properties defined in terms of equations naturally turn into algorithms when solvers for equations are developed.
- Solvers can be combined to make a more versatile and powerful solver. Collaboration of solvers are important for solving real-life problems.
- In summary, equations are (one of) bridges between mathematics and computer science, especially programming.

References

[1] Tetsuo Ida, Mircea Marin and Taro Suzuki, Higher-order Lazy Narrowing Calculus: a Solver for Higher-order Equations, Conference on Computer Aided Systems (EUROCAST 2001)}, Lecture Notes in Computer Science 2178, Las Palmas de Gran Canaria, Spain, pp. 478-493, 2001

[2] Tetsuo Ida, Mircea Marin and Norio Kobayashi, An Open Environment for Cooperative Scientific Problem Solving, Proceedings of the fourth International Mathematica Symposium (IMS 2001), Chiba Japan, pp. 71--78, 2001

[3] Mircea Marin, Taro Suzuki and Tetsuo Ida, Higher-Order Lazy Narrowing Calculi for Pattern Rewrite Systems, Technical Report, ISE-TR-01-180, Institute of Information Sciences and Electronics, University of Tsukuba, 2001

[4] Humiaki Huzita, Axiomatic Development of Origami Geometry, Proceedings of the First International Meeting of Origami Science and Technology, pp. 143--158, 1989

[5] Thomas Hull, Origami and Geometric Constructions, http://web.merrimack.edu/~thull/geoconst.html, 1997

PAPER

Collaborative Constraint Functional Logic Programming System in an Open Environment

Norio KOBAYASHI[†], Mircea MARIN[††], *Nonmembers, and* Tetsuo IDA[††], *Regular Member*

SUMMARY In this paper we describe collaborative constraint functional logic programming and the system called Open CFLP that supports this programming paradigm. The system solves equations by collaboration of various equational constraint solvers. The solvers include higher-order lazy narrowing calculi that serve as the interpreter of higher-order functional logic programming, and specialized solvers for solving equations over specific domains, such as a polynomial solver and a differential equation solver. The constraint solvers are distributed in an open environment such as the Internet. They act as providers of constraint solving services. The collaboration between solvers is programmed in a coordination language embedded in a host language. In Open CFLP the user can solve equations in a higher-order functional logic programming style and yet exploit solving resources in the Internet without giving low-level programs of distributions of resources or specifying details of solvers deployed in the Internet.

key words: equational solving, functional logic programming, solver collaboration, constraint solving, CORBA

1. Introduction

In our previous work [12], we presented a constraint functional logic programming system called CFLP, where the higher-order lazy narrowing calculus [11] and built-in constraint solvers collaborate to solve equational goals. Constraint functional logic programming is an integration of constraint programming, and functional and logic programming (FLP for short).*

An important observation to be made here is that FLP is a paradigm of solving equations over the domain of terms. In other words, FLP itself is also constraint programming aiming at solving constraints over domains described by rewrite systems. FLP holds a distinguished position in programming, since it manipulates terms directly without giving special meaning to terms. FLP alone, however, is not sufficient to solve equations modelling practical scientific problems. In real world applications, equations are often defined over several domains with terms interpreted specially in each domain. Each equation defined over a specific domain requires a dedicated constraint solver. Here the paradigm of constraint programming, which was originally proposed to extend the capability of logic programming, naturally comes into play. Constraint FLP is thus general constraint programming, where various constraint solvers interplay.

The language of FLP consists of equational goals and a set of rewrite rules. From programming language point of view, we need no extra linguistic constructs for constraint programming. In addition, constraint FLP requires constructs which specify how the constraint solvers collaborate to solve equations. These constructs add a new dimension to programming, *i.e.*, programming of collaboration of solvers. This fits very well in the advocated slogan [5]: programming = computation + coordination, where in our case coordination is collaboration of constraint solvers. A similar argument focusing on constraint solving has been made in [13].

In this paper, we present a new version of CFLP running in an open environment, where we program collaborations of solvers in addition to FLP. We call the new system Open CFLP since solvers are not fixed a priori and should be obtained dynamically from the open environment. Our work is based on the service model where solvers are distributed over the network and that they are not downloadable to the clients. The reasons for taking this premise are as follows:

- solvers evolve over time; sophisticated algorithms may require long time efforts for their perfection,
- solvers may be big and require specialized resources which are not downloadable,
- solvers are willing to provide services, but not necessarily willing to allow the clients to copy the programs to protect the intelligent and copy rights,
- solvers and the clients that use those services are independent and only communicate with the standardized protocol, and
- solving services may be deployed dynamically, independent of the plan of the potential clients.

The focus of the paper is the programming paradigm with Open CFLP, and on the architecture of Open CFLP. The rest of this paper is structured as follows. In Sect. 2 we describe a programming example that illustrates collaborative constraint programming. In Sect. 3 we describe the languages of our system.

Manuscript received January 4, 2002.
Manuscript revised September 10, 2002.

[†]Doctoral Program in Engineering, University of Tsukuba, Japan.
[††]Institute of Information Sciences and Electronics, University of Tsukuba, Japan.

*Functional and Logic programming is also an integration of functional programming and logic programming, but in this paper we take this integration for granted as the paradigm of functional and logic programming is amply discussed elsewhere [6].

In Sect. 4 and 5 we describe the architecture of Open CFLP. Finally, in Sect. 6 we draw conclusions.

2. Motivating example

We begin by a small motivating example to illustrate how we solve a problem with Open CFLP. The example is taken from [8]. The following presentation may appear a bit contrived to make clear the essence of our collaborative constraint functional logic programming. Let us consider the problem of solving the equations

$$y'(t) = k\ y(t),\ y(0) = 1,\ y(2) = 3,\ y(T) = 5$$

for variables y, k and T. Let us assume that we are functional programmers, and we decide to use the definition of map to make the program succinct. We further assume that map is not a built-in function, and so we write down its definition.

Our system is built on top of the Mathematica system [14], and we write the program in the language of Mathematica in a Mathematica notebook. Function FLPProgram, shown below, is a special function that treats its arguments as a FLP program. The result of the evaluation of the function call is the internal representation of the FLP program. The language of CFLP [12] is that of (conditional) pattern rewrite systems expressed in the syntax of Mathematica. The terms are simply-typed λ terms in $\beta\eta^{-1}$ normal form. The symbols on the left-hand side of the rewrite rules are underlined if they are free variables.

```
(* FLP program declaration *)
R = FLPProgram[
        {map[λ[{t},f[t]],{}] → {},
         map[λ[{t},f[t]],[H | T]] →
            [f[H] | map[λ[{t},f[t]],T]]},
        Signature →
         {DefinedSymbols →
            {map : (ℝ → ℝ) × TyList[ℝ] → TyList[ℝ]}}]
```

Next we specify the goal to be solved. Logically a goal is an existentially quantified formula of conjunction of equations $\exists x_1 \cdots x_m . e_1 \wedge \cdots \wedge e_n$, where $e_1, \ldots, e_n$ are equations. In the language of CFLP, we write it as exists[{x₁, …, xₘ}, {e₁, …, eₙ}]. There are three kinds of equations; oriented equation s ▷ t, unoriented equation s == t and strict unoriented equation s === t. The former two are relevant to the present discussion. If the equation s ▷ t or s == t is in solved form, it is written as s ↦ t. The variables occurring in the formula can be type-annotated. Thus, the goal to be solved is as follows.

```
(* Goal specification *)
G = exists[{y : ℝ → ℝ, k : ℝ, T : ℝ},
        {λ[{t}, y'[t]] == λ[{t}, k y[t]],
         map[λ[{t}. y[t]], {0, 2, T}] == {1, 3, 5}}]
```

We now have to find solvers in the open environment. We may have solvers in our local computer, but here let us assume that the necessary solvers are on the Internet. Our computing model is that we are not allowed to download the solvers, but are allowed only to use the service of the solvers. We only know the names of the services that are available somewhere on the network. In order to uniquely identify the solving service, we use URI to name the service. For example, the URI http://www.score.is.tsukuba.ac.jp/OCFLP/HOLN is the name of the service of solving equations over the domain of higher-order terms using Higher-Order Lazy Narrowing Calculus HOLN [11]. FindSolvers["http://www.score.is.tsukuba.ac.jp/OCFLP/HOLN"] will find the solvers that offer the service of HOLN by the help of the broker of Open CFLP (*cf.* Sect. 5.2). Similarly, we will find the solvers for systems of differential equations and systems of linear equations.

```
(* Find solvers *)
aHOLN = FindSolvers[
  "http://www.score.is.tsukuba.ac.jp/OCFLP/HOLN"]
aDeriv = FindSolvers[
  "http://www.score.is.tsukuba.ac.jp/OCFLP/Deriv"]
aLinear = FindSolvers[
  "http://www.score.is.tsukuba.ac.jp/OCFLP/Linear"]
```

FindSolvers returns the entity *elementary collaborative*. An elementary collaborative is a collection of basic solvers that perform the same solving service. In Sect. 5.2, we will explain the mechanism of finding such solvers. An elementary collaborative can be configured by ConfigSolvers to work on specific problems more effectively by supplying additional parameters. To some solvers, configuration is essential to equip them with necessary solving power. For example, in the case of HOLN solvers, we need to supply a FLP program R to solve equations with respect to R. Thus, we configure the HOLN solvers as follows.

```
(* Configure solvers *)
aHOLN = ConfigSolvers[aHOLN, Prog → R]
```

The solvers of other two services need not be configured since we use their standard services.

The next step of programming is the important one in Open CFLP. We program a collaborative solver. In CFLP, collaboration of solvers is fixed and is *hard-wired* to CFLP system. For this example, we define a collaborative solver in which elementary collaboratives aHOLN, aDeriv and aLinear collaborate. First, we want to apply the goal to these elementary collaboratives sequentially in the order of aHOLN, aDeriv and aLinear. This is programmed as seq[{aHOLN, aDeriv, aLinear}]. Furthermore, we apply the goal to seq[{aHOLN, aDeriv, aLinear}] repeatedly until the goal reaches the fixed-point. Thus we have a collaborative definition repeat[seq[{aHOLN, aDeriv, aLinear}]]. The definition is given to the system using a special function NewCollaborative.

```
(* Collaborative specification *)
aCollabo = NewCollaborative[
              repeat[seq[{aHOLN, aDeriv, aLinear}]]]
```

NewCollaborative returns the entity *collaborative*.

Finally, we apply the goal G to the collaborative aCollabo:

```
(* Invoke collaborative *)
ApplyCollaborative[aCollabo, {G}]
```

and obtain the following solution.

$$\begin{aligned}&\{\{\mathtt{y} \mapsto \lambda[\{\mathtt{t}\}, \mathtt{e}^{\mathtt{Log[3]\ t/2}}], \mathtt{k} \mapsto \mathtt{Log[3]/2},\\ &\quad \mathtt{T} \mapsto \mathtt{2\ Log[5]/Log[3]}\}\}\end{aligned}$$

The following transformation of goals took place during the solving process of the initial goal G.

$$\begin{aligned}&\{\mathtt{G}\} \Rightarrow_{\mathtt{aHOLN}} \{G_1\} \Rightarrow_{\mathtt{aDeriv}} \{G_2\} \Rightarrow_{\mathtt{aLinear}} \{G_3\}\\ &\quad \Rightarrow_{\mathtt{aHOLN}} \{G_3\} \Rightarrow_{\mathtt{aDeriv}} \{G_3\} \Rightarrow_{\mathtt{aLinear}} \{G_3\}\end{aligned}$$

where [†]

$$\begin{aligned}\{G_1\} &= \{\mathtt{exists}[\{\mathtt{h}, \mathtt{T_1}\}, \{\mathtt{y} \mapsto \lambda[\{\mathtt{t}\}, \mathtt{h[t]}],\\ &\qquad \mathtt{T} \mapsto \mathtt{T_1}, \lambda[\{\mathtt{t}\}, \mathtt{h'[t]}] == \lambda[\{\mathtt{t}\}, \mathtt{k\ h[t]}],\\ &\qquad \mathtt{h[0] == 1, h[2] == 3, h[T_1] == 5}, \ldots\}]\},\\ \{G_2\} &= \{\mathtt{exists}[\{\mathtt{c}, \mathtt{k}, \mathtt{T_1}\}, \{\mathtt{h} \mapsto \lambda[\{\mathtt{t}\}, \mathtt{c\ e^{k\ t}}],\\ &\qquad \mathtt{T} \mapsto \mathtt{T_1}, \mathtt{y} \mapsto \lambda[\{\mathtt{t}\}, \mathtt{c\ e^{k\ t}}],\\ &\qquad \mathtt{c == 1, c\ e^{2\ k} == 3, c\ e^{k\ T_1} == 5}, \ldots\}]\},\\ \{G_3\} &= \{\{\mathtt{c} \mapsto \mathtt{1}, \mathtt{T_1} \mapsto \mathtt{2\ Log[5]/Log[3]}, \mathtt{k} \mapsto \mathtt{Log[3]/2},\\ &\qquad \mathtt{h} \mapsto \lambda[\{\mathtt{t}\}, \mathtt{e}^{\mathtt{Log[3]\ t/2}}], \mathtt{y} \mapsto \lambda[\{\mathtt{t}\}, \mathtt{e}^{\mathtt{Log[3]\ t/2}}],\\ &\qquad \mathtt{T} \mapsto \mathtt{2\ Log[5]/Log[3]}, \ldots\}\}.\end{aligned}$$

Note that the collaborative operates on a list of goals and returns a list of goals and that $\{G_1\}$ can be obtained if we evaluate ApplyCollaborative[NewCollaborative[aHOLN], {G}].

3. Languages of Open CFLP

From the example in the previous section we see that collaborative constraint functional logic programming proceeds in the following steps:

1. Define a FLP program R.
2. Define a goal G to be solved.
3. Obtain sets $C_1, \ldots, C_n$ of elementary collaboratives.
4. Define a new collaborative C by specifying a collaboration of elementary collaboratives $C_1, \ldots, C_n$.
5. Apply the collaborative C to G.

[†]HOLN introduces extra variables that are irrelevant in this illustration. We have abbreviated their bindings in G_1, G_2 and G_3 by ….

In the scientific problem solving the above process is often interactive. For instance the obtained solution may become new goals after inspecting and editing it. Furthermore even if the FLP program R is completed, the steps 2 to 5 are repeated. This requires the collaboration of solvers to be re-programmed since some combination of solvers may not deliver a desired solution. Therefore, our language should also have the power of modern programming languages such as graphics and interactive debugging. This observation leads to the following design decision.

3.1 Language $\mathcal{M}$

The language of Open CFLP is built on top of Mathematica. We decided to use Mathematica as a base language since we need to make use of its built-in mathematical knowledge and symbolic processing capability. Let us denote our language by $\mathcal{M}$ in this paper. Since the syntax of the language of Mathematica is universal in that a form of functional application like $f[s_1, \ldots, s_n]$ is a basic building block, extending the functionalities that we have shown in the previous section can be made simply by providing special functions to Mathematica. We have seen functions like FindSolvers, NewCollaborative and ApplyCollaborative in our example. Functions FindSolvers, NewCollaborative and ApplyCollaborative are by no means trivially implemented by Mathematica programs. Rather they are realized by sophisticated software components that we have built for Open CFLP.

A collaborative is programmed in a coordination language embedded in $\mathcal{M}$. The language is denoted by $\mathcal{L}$ in this paper. For the definition of a collaborative in $\mathcal{M}$, we use function NewCollaborative whose argument is a collaborative expression in $\mathcal{L}$.

3.2 Coordination Language $\mathcal{L}$

Open CFLP system has a software component called *coordinator*. The coordinator, as the name suggests, coordinates the activities of solvers. The coordinator evaluates the program of $\mathcal{L}$. An element of $\mathcal{L}$ is collaborative. The collaborative C and the list $\overline{G}$ of goals are sent from the user frontend of the system (*cf.* Sect. 4.1) in the form of a function application ApplyCollaborative[$C, \overline{G}$].

Language $\mathcal{L}$ allows us to define a new solver by combining various solvers using collaboration combinators seq, if, choice and repeat. A collaborative C is inductively defined as follows:

$C ::= \mathcal{B}$	elementary collaborative
$\mid\ \mathcal{X}$	locally defined collaborative
$\mid\ \mathtt{seq}[\{C_1, \ldots, C_n\}]$	sequential
$\mid\ \mathtt{if}[\phi, C_1, C_2]$	conditional
$\mid\ \mathtt{choice}[\psi, \{C_1, \ldots, C_n\}]$	concurrency and choice
$\mid\ \mathtt{repeat}[C]$	repetition

A collaborative receives a list of goals. All the solutions of the goals are collected and is returned as the solution of the given list of the goals.

We will give informal semantics of the language. Suppose a collaborative C and a list $\overline{G}$ of goals are given initially.

- When C is an elementary collaborative, C is applied to $\overline{G}$ directly.
- When C is a locally defined collaborative, where 'locally defined' means that C is defined as a Mathematica function, the definition of C is interpreted by the user frontend.
- When C is seq[$\{C_1, \ldots, C_n\}$], we distinguish the following two cases. If $n = 1$, seq[$\{C_1\}$] is the same as C_1. Otherwise, seq[$\{C_2, \ldots, C_n\}$] is applied to the result of application of C_1 to $\overline{G}$.
- When C is if[ϕ, C_1, C_2], the following takes place. $\phi(\overline{G})$ is computed first. Function ϕ probes the list of goals $\overline{G}$ and checks whether each goal of $\overline{G}$ possesses a certain property, *e.g.*, size or linearity. $\phi(\overline{G})$ returns a list $\{\overline{G_T}, \overline{G_F}\}$, where $\overline{G_T}$ is a list of goals which satisfy the property, and $\overline{G_F}$ is a list of goals which do not satisfy it. Finally the collaboratives C_1 and C_2 are applied to $\overline{G_T}$ and $\overline{G_F}$ respectively.
- When C is choice[$\psi, \{C_1, \ldots, C_n\}$], the collaboratives $C_1, \ldots, C_n$ are applied to $\overline{G}$ simultaneously and ψ selects one of the results.
- When C is repeat[C], the collaborative C is repeatedly applied to the goal list (initially $\overline{G}$) until no further transformation of the goal list is possible by C.

Language $\mathcal{L}$ is designed to be a small language with primitives for coordination of solvers. It is a very basic language for two reasons. First, $\mathcal{L}$ is a language embedded in a powerful language $\mathcal{M}$. Secondly, additional functionalities that may be desired for full-fledged coordination languages can be provided by $\mathcal{M}$. For example, repeat can be easily programmed in $\mathcal{M}$ using the built-in higher-order function FixedPoint, as follows.

```
Repeat[collabo_] := Function[GList,
  FixedPoint[Function[x,
    ApplyCollaborative[collabo, x]], GList]]
```

For efficiency reasons, repeat is provided as primitive, however.

Similarly, we show that the sequential version of conditional collaboration CondCollabo can be defined as a Mathematica function.

```
CondCollabo[Choice_, collabo1_, collabo2_]
  := Function[GList, Union[
       ApplyCollaborative[collabo1,
         Choice[GList][[1]]],
       ApplyCollaborative[collabo2,
         Choice[GList][[2]]]]]:
```

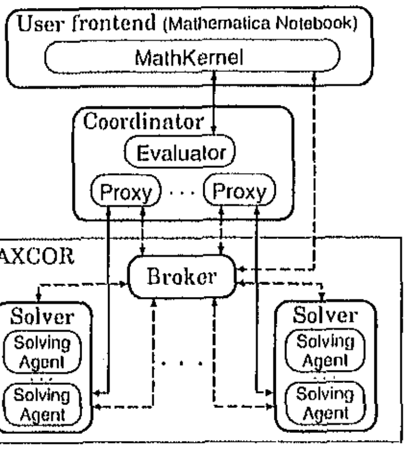


Fig. 1 The architecture of Open CFLP

```
aCondCollabo_M
  := CondCollabo[aChoice, aCollabo1, aCollabo2]
```

delivers the same result as the collaborative defined by

```
aCondCollabo_L
  = NewCollaborative[
      if[aChoice, aCollabo1, aCollabo2]],
```

although aCondCollabo$_L$ is more efficiently executed, because expression aCondCollabo$_M$[GList] is interpreted by the user frontend, whereas ApplyCollaboarive[aCondCollabo$_L$, GList] is interpreted by the coordinator.

If the choice function aChoice in aCondCollabo$_L$ is provided as the primitive in $\mathcal{L}$, running in the coordinator, the performance difference is even greater. However, there are cases where simplicity and flexibility outweigh the execution efficiency, in which case CondCollabo defined as the higher-order function of Mathematica would be preferable. For example, aChoice that checks whether each goal is a system of linear equations can be programmed much more easily with Mathematica.

Language $\mathcal{L}$ can be seen as a core of coordination languages that have been studied by several researchers (*cf.* [9], for a good survey).

4. Architecture of Open CFLP

The discussion about the program in Sect. 2 has revealed the following ingredients of Open CFLP:

- user frontend,
- coordinator,
- open framework for solver collaboration, and
- solvers.

These ingredients are realized by software components called *user frontend*, *coordinator*, *broker* and *solver*.

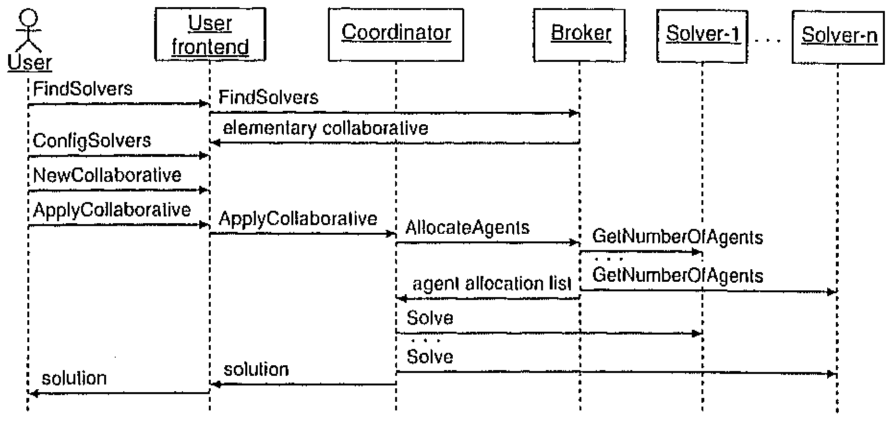


Fig. 2 Interaction between components of Open CFLP

Figure 1 shows the architecture of Open CFLP. Figure 2 is a sequence diagram that shows how the components interact one another. Functionally, the broker and the solvers are grouped together to form a framework called MAXCOR(MAth eXchange for CORBA). Since MAXCOR has many functionalities, we discuss the detail of MAXCOR separately in Sect. 5. In the following subsections we will explain each component in more detail.

4.1 User frontend

The user frontend (frontend for short) is a user interface to Open CFLP. It is a Mathematica notebook equipped with the MathKernel and allows the user to define CFLP programs in $\mathcal{M}$, as well as to interact with the Mathematica system. The MathKernel executes locally defined collaboratives as well as Mathematica programs that run with Open CFLP. The interface component called palette is also provided to facilitate the input of CFLP expressions and commands.

4.2 Solver

A solver is a solving service provider. It implements a solving algorithm such as higher-order lazy narrowing, Gaussian elimination method and Gröbner basis algorithm. When the operation `Solve` is invoked by the coordinator (*cf.* Fig. 2), the solver creates (possibly multiple) solving processes. We call the solving processes *solving agents.* The solving agents of a solver execute the same solving algorithm in parallel.

4.3 Coordinator

The coordinator is in charge of evaluating the program of $\mathcal{L}$. The evaluation consists in creating the proxies for the solvers and in interpreting the collaboratives as described in Sect. 3.2. The main component of the coordinator is the *evaluator*. It communicates with the frontend and the broker. When the evaluator receives a collaborative C and a list $\overline{G}$ of goals from the frontend, it creates a buffer U where the solutions are stored and then executes the procedure `Collabo` given in Appendix. When the execution of the procedure `Collabo` is completed, the evaluator fetches the solution stored on U and returns it to the frontend.

We will explain the procedure `Collabo` when C is an elementary collaborative. The procedure in other cases is a straightforward translation of the informal semantics explained in Sect. 3.2.

The important design issue here is how to exploit the parallelism. Note that C is a collection of basic solvers $s_1, \ldots, s_m$ and $\overline{G}$ is a list of goals $G_1, \ldots, G_n$. Each solver s_i can spawn τ_i solving agents. Ideally, n goals are solved in parallel by n solving agents. The procedure `Collabo` will exploit parallelism afforded in the open environment in the following way.

1. Contact the broker and obtain the number τ_i' of currently assignable solving agents for each solver s_i, $i = 1, \ldots, m$ such that $\tau_i' \leq \tau_i$ and $\tau_1' + \cdots + \tau_m' = n' \leq n$.
2. Create a proxy of each solver s_i if $\tau_i' \neq 0$, for $i = 1, \ldots, m$.
3. Distribute n goals among n' solving agents via the proxies.
4. Configure the solving agents if necessary and then trigger the n' solving agents to solve the given goals via their proxies.
5. Probe the states of the computations via the proxies. Delete the proxies if all the computations are completed, and send the request to the broker to de-allocate all the n' involved solving agents.
6. Return the solution to the frontend.

Here, The procedure `Collabo` exploits n'-fold parallelism. Figure 2 shows the sequence of actions taken by the coordinator, the broker and the solvers.

We illustrate the behavior of the coordinator by the example of Sect. 2. When `ApplyCollaborative[`

aCollabo, {G}] that is the operation defined in the coordinator is evaluated, aCollabo is first evaluated to repeat[seq[{aHOLN, aDeriv, aLinear}]] then ApplyCollaborative[repeat[seq[{aHOLN, aDeriv, aLinear}]], {G}] is sent to the coordinator.

1. The evaluator creates a buffer U_0.
2. The evaluator calls the procedure Collabo with the parameters repeat[seq[{aHOLN, aDeriv, aLinear}]], {G} and U_0.
3. Collabo creates a buffer U_1 and recursively calls Collabo with the parameters seq[{aHOLN, aDeriv, aLinear}]], {G} and U_1.
   - 3.1. Collabo creates a buffer U_2.
   - 3.2. Collabo obtains a solving agent HOLN from the broker and creates the proxy of the solver of HOLN.
   - 3.3. The proxy sends the FLP program R for configuring this solver and sends {G} to this solver by invoking Solve operation.
   - 3.4. HOLN agent solves {G} and appends the solution $\{G_1\}$ to U_2.
   - 3.5. Similarly, Deriv agent and Linear agent solve their goals in this order.
4. The result $\{G_3\}$ of Step 3 is appended to the buffer U_1, and the evaluator compares U_1 and {G}.
5. Steps 2–4 are obeyed once more with the parameters repeat[seq[{aHOLN, aDeriv, aLinear}]], $\{G_3\}$ and U_0.
6. Finally, the solution $\{G_3\}$ is appended to U_0.

We assume that a solving agent returns finite number of solutions, since the evaluation is call-by-value, following the default evaluation mode of Mathematica. The coordinator can display the progress of solving. In the case the computation appears be non-terminating, the user can interrupt the computation.

Furthermore, for efficiency the coordinator has a cache of solving agents, although the description of the cache is not explicit in the procedure Collabo. The cached solving agents are reused until the CFLP session is over.

5. MAXCOR

MAXCOR is designed to be a framework which realizes transparent communication between solvers. It consists of object wrappers for those solvers and the broker.

We often want to use various existing solvers such as HOLN and polynomial solvers integrated to Mathematica and integer programming solver CPLEX [2]. These solvers may be heterogeneous, *i.e.*, they are implemented in different programming languages, run on different platforms and have different data formats for describing constraints. Object wrappers are used to hide the heterogeneity of such solvers. To realize this functionality, we need a common data format for communication and a common communication protocol. We adopted MathML for the common data format and CORBA for the communication protocol. An object wrapper thus has the functionalities of data conversion between MathML and the solvers' own data formats, and of mediation of solvers' operations.

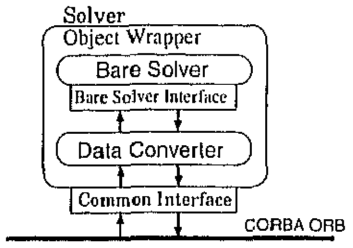


Fig. 3 The architecture of the object wrapper

An object wrapper alone is not sufficient to realize the intended transparent communication with the coordinator. MAXCOR has to provide the facilities for finding the solvers. We are going to discuss in more detail about these components.

5.1 Object Wrappers

Figure 3 shows the architecture of the object wrapper. It provides CORBA compliant common interface for solvers and realizes a solver using a bare solver. The object wrapper transmits MathML data, more specifically valuetype objects of XML DOM (Document Object Model) [1], the standard allowing solvers to transmit mathematical formulas written in MathML on CORBA ORB. The object wrapper is equipped with the data converter between MathML and the bare solver's own internal representation of mathematical formulas. We have implemented object wrappers for Mathematica and CPLEX.

5.2 Broker

The scheme of broker–provider coordination is a well understood design pattern [10], and we follow the scheme for broker–solver coordination. Solvers are providers that offer solving services and the broker finds the appropriate service to its clients. Here the clients are proxies created by the coordinator.

The broker communicates with the frontend and the coordinator. It receives the command FindSolvers[*uri*] from the frontend, where *uri* is a name of the service. Recall that the service of solving a goal is given a unique name in the format of URI. From the coordinator, it receives the request of the process allocation for running the solving agent.

(1) Search of solvers

When the broker receives FindSolvers[*uri*], it accesses a *service table* and then by simple look-up with *uri* it finds a list of solvers associated with *uri*. This list will be returned to the frontend. The broker maintains the service table with entries of the name of the service and a list of solvers whose agents perform the service. A solver published the availability of service by requesting the broker the registration of the solver with the name of the service in the service table.

(2) Agent allocation

When the coordinator requests n solving agents for the elementary collaborative C by invoking the operation AllocateAgents$[C, n]$(*cf.* Fig. 2), the broker returns at most n solving agents in the following way:

1. The broker asks each solver $s_1, \ldots, s_m$ collected in C the number of solving agents it can give to the coordinator by invoking the operation GetNumberOfAgents.
2. Solver s_i returns the number τ_i, for $i = 1, \ldots, m$.
3. Based on τ_i, the broker distributes n solving agents among m solvers according to its load balancing policy. Let $n' = \min(n, \tau_1 + \cdots + \tau_n)$. The broker decides τ'_i for each s_i such that $\tau'_i \leq \tau_i$ and $\tau'_1 + \cdots + \tau'_m = n'$.
4. The broker informs each solver $s_1, \ldots, s_m$ that the broker reserves $\tau'_1, \ldots, \tau'_m$ solving agents.
5. The broker returns an agent allocation list $\{\{s_1, \tau'_1\}, \cdots, \{s_m, \tau'_m\}\}$ to the coordinator.

5.3 Features of MAXCOR

MAXCOR is designed as an application of CORBA. The broker and the solvers are CORBA servers, and the frontend and the proxies in the coordinator are CORBA clients. Combined with the features of CORBA, we have achieved the following properties:

- portability—language and platform independence among solvers,
- data interoperability—MathML documents,
- operation interoperability among solvers,
- scalability and modularity—solvers implemented as CORBA servers, and
- location independence of solvers.

Furthermore, we can easily extend our system with rich CORBA common services such as the security service.

In addition, we implemented the broker that CORBA does not offer to achieve dynamic solver deployment. It has the functionalities of searching solvers and allocating solving agents. Furthermore, our broker implements a load balancing policy for the allocation of solving agents, hence realizes efficient use of distributed resources.

6. Conclusions

We have described collaborative constraint functional logic programming and a system that supports this paradigm. The system is open in the sense that it can access via a brokering service the constraint solving resources available in an open environment.

The language of Open CFLP is multi-tiered. The notions of constraint solving, coordination programming and symbolic computation are separable in developing programs, and such a separation leads to a succinct and modular programming style. Yet, each activity is supported by existing programming capabilities. Our languages $\mathcal{M}$ and $\mathcal{L}$ are embedded into the language of Mathematica. This flexibility allows us to make our coordination language $\mathcal{L}$ small. $\mathcal{L}$ is essentially a core of the coordination language BALI [3], [13].

In [3], [13], an implementation model of BALI with MANIFOLD is discussed. Our coordinator and MANIFOLD are based on a control-driven coordination model such as ConCoord [7] and TOOLBUS [4]. The distinctive feature of Open CFLP is that it has been implemented using middleware technology CORBA and further has realized broker-provider scheme needed for open collaborative constraint solving. By this approach, we have achieved openness and extensibility of the system. As far as we know, a collaborative constraint system fully implemented with open technologies, CORBA and MathML, with clear design goals of scientific problem solving is our new contribution in this field of collaborative constraint programming.

References

[1] http://cgi.omg.org/xml.

[2] Using the CPLEX Callable Library, CPLEX Optimization, Inc., USA, 1995.

[3] F. Arbab and E. Monfroy, "Coordination of heterogeneous distributed cooperative constraint solving,", Applied Computing Review, SIGAPP. ACM, vol.6, pp.4–17, 1998.

[4] J. Bergstra and P. Klint, "The TOOLBUS coordination architecture," in 1st Int'l Conf. Coordination Models, Languages and Applications (Coordination'96), ed. P. Ciancarini and C. Hankin, LNCS, vol.1061, pp.75–88, Springer-Verlag, 1996.

[5] D. Gelernter and N. Carriero, "Coordination languages and their significance: From theory to practice," Commun. ACM, vol.35, no.2, pp.97–107, 1992.

[6] M. Hanus, "The integration of functions into logic programming: From theory to practice," Journal of Logic Programming, no.19&20, pp.583–628, 1994.

[7] A. Holzbacher, "A software environment for concurrent coordinated programming," in 1st Int'l Conf. Coordination Models, Languages and Applications (Coordination'96), ed. P. Ciancarini and C. Hankin, LNCS, vol.1061, pp.249–266, Springer-Verlag, 1996.

[8] H. Hong, "RISC-CLP(CF) Constraint logic programming over complex functions," Logic Programming and Automated Reasoning, Proc. of the 5th Int'l Conf., LPAR'94, ed. F. Pfennig, pp.99–113, Springer-Verlag, 1994.

[9] A. Omicini and F. Zambonelli,ed., Coordination of Internet Agents, Springer-Verlag, 2001.

[10] M. Klusch and K. Sycara, "Brokering and matchmaking for coordination of agent societies: A survey," in Coordina-

tion of Internet Agents, ed. A. Omicini and F. Zambonelli, pp.197–224, Springer-Verlag, 2001.

[11] M. Marin, T. Ida, and T. Suzuki, "Higher-order lazy narrowing calculus: A solver for higher-order equations," in Proc. 8th Int'l Conf. Computer Aided Systems (EuroCAST 2001), LNCS, vol.2178, pp.478–493, Springer-Verlag, 2001.

[12] M. Marin, T. Ida, and W. Schreiner, "CFLP: Distributed constraint solving system," The Mathematica Journal, vol.8, no.2, pp.287–300, 2001.

[13] E. Monfroy and F. Arbab, "Constraints solving as the coordination of inference engines," in Coordination of Internet Agents, ed. A. Omicini and F. Zambonelli, pp.399–419, Springer-Verlag, 2001.

[14] S. Wolfram, The Mathematica Book, 4th Edition, Wolfram Media Inc. Champaign, Illinois, USA, and Cambridge University Press, 1999.

Appendix: Procedure Collabo

```
procedure Collabo($C, L, U$)

    {Inputs are collaborative $C$, a list $L$ of $n$
    goals and a buffer $U$.}

  case $C$ is an elementary collaborative
    begin
      Obtain the agent allocation list
      $\{\{s_1, \tau_1'\}, \ldots, \{s_m, \tau_m'\}\}$ from the broker, where
      $s_1, \ldots, s_m$ are solvers of $C$;
      for $i = 1, \ldots, m$ do
        if $\tau_i' \neq 0$ then Create a proxy of $s_i$;
      Distribute $L$ to the proxies;
      Send configuration requests if $C$ includes
      parameters for configuration, and solving
      requests to the proxies;
      Obtain the solutions from the proxies;
      Append each solution to $U$;
      Delete the proxies and issue a request to the
      broker to de-allocate the solving agents;
    end;
  case $C$ is a locally defined collaborative $X$
    begin
      Send $L$ to the frontend;
      {The frontend evaluates X[L].}
      Append to $U$ the result sent from the frontend;
    end;
  case $C$ = seq[$\{C_1, \ldots, C_k\}$]
    if $k = 1$ then call Collabo($C_1, L, U$);
    else
      begin
        Create a buffer $U'$;
        call Collabo($C_1, L, U'$);
        call Collabo(seq[$\{C_2, \ldots, C_k\}$], $U', U$);
      end;
  case $C$ = if[$\phi, C_1, C_2$]
    begin
      Let $\{L_T, L_F\} = \phi(L)$;
      call Collabo($C_1, L_T, U$) and
      call Collabo($C_2, L_F, U$) simultaneously;
    end;
  case $C$ = choice[$\psi, \{C_1, \ldots, C_k\}$]
    begin
      Create $k$ buffers $U_1, \ldots, U_k$;
      call Collabo($C_1, L, U_1$), ...,
      call Collabo($C_k, L, U_k$) simultaneously;
      Append $\psi(U_1, \ldots, U_k)$ to $U$;
    end;
  case $C$ = repeat[$C_r$]
    begin
      Create a buffer $U'$;
      call Collabo($C_r, L, U'$);
      if the contents of the buffer $U'$ and $L$ are the
      same then Append $L$ to $U$;
      else call Collabo($C, U', U$);
    end;
end of collabo;
```

Norio Kobayashi is currently a doctor course graduate student in engineering, University of Tsukuba. His research interests include scientific equational solving and open computing model for collaborative equational solving. He received the B.Sc. degree in physics from Science University of Tokyo and M.E. degree in information sciences and electronics from University of Tsukuba in 1997 and 1999, respectively.

Mircea Marin is a visiting researcher at the University of Tsukuba. His current research interests include integration of functional logic programming with constraint solving, and coordination models for collaborative constraint solving. He holds a Ph.D. in Computer Science from the Johannes Kepler University of Linz, Austria.

Tetsuo Ida is a professor at the University of Tsukuba, where he leads a research group of symbolic computation (SCORE) in the institute of information sciences and electronics. His research includes distributed symbolic computation, integration of functional and logic programming and term rewriting. He is an editor of the Journal of Symbolic Computation and the Journal of Functional and Logic Programming. He is a member of the IEICE, the IPSJ, the JSSST, the ACM and the IEEE Computer Society. He received a Doctor of Science from the University of Tokyo.