Jean-Louis Giavitto
Pierre-Etienne Moreau (Eds.)

Rule-Based Programming

4th International Workshop, RULE 2003
Valencia, Spain, June 9, 2003
Proceedings

Volume Editors

Jean-Louis Giavitto
Université d'Evry Val d'Essone, Evry, FRANCE
Email: giavitto@lami.univ-evry.fr

Pierre-Etienne Moreau
INRIA Lorraine & LORIA, Villers-lès-Nancy, FRANCE
Email: Pierre-Etienne.Moreau@loria.fr

Proceedings of the 4th International Workshop on Rule-Based Programming, RULE'03
Valencia, Spain, June 9, 2003

ISBN:

Depósito Legal:

Impreso en España.

Technical Report DSIC-II/11/03,
http://www.dsic.upv.es
Departamento de Sistemas Informáticos y Computación,
Universidad Politécnica de Valencia, 2003.

Foreword

This volume contains the pre-proceedings of the Fourth International Workshop on Rule-Based Programming (RULE2003).

This year, RULE 2003 is part of a federation of colloquia known as the Federated Conference on Rewriting, Deduction and Programming (RDP 2003) which includes the 14th International Conference on Rewriting Techniques and Applications (RTA), the 6th Conference on Typed Lambda Calculi and Applications (TLCA), the 5th Workshop on First-order Theorem Proving (FTP), as well as several other workshops. The colloquia and affiliated workshops will run from June 8 to June 14, 2003 and will be held in Valencia, Spain. Details about the affiliated conferences and workshops will appear at the URL www.dsic.upv.es/~rdp03

Previous RULE meeting were held in Montréal (2000), Firenze (2001) and Pittsburgh (2002); their proceedings were published by Elsevier and by the ACM/SIGPLAN. The final version of this volume will be published as volume 86.2 in the series Electronic Notes in Theoretical Computer Science (ENTCS). This series is published electronically through the facilities of Elsevier Science B.V. and its auspices. The volumes in the ENTCS series can be accessed at URL www.elsevier.nl/locate/entcs.

The rule-based programming paradigm is characterized by the repeated, localized transformation of a shared data object such as a term, graph, proof, or constraint store. The transformations are described by rules which separate the description of the sub-object to be replaced (the pattern) from the calculation of the replacement. Optionally, rules can have further conditions that restrict their applicability. The transformations are controlled by explicit or implicit strategies.

The basic concepts of rule-based programming appear throughout computer science, from theoretical foundations to practical implementations. Term rewriting is used in semantics in order to describe the meaning of programming languages, as well as in the implementation of program transformation systems. It is used implicitly or explicitly to perform computations, e.g., in Mathematica, OBJ, or ELAN, or to perform deductions, e.g., by using inference rules to describe or implement a logic, theorem prover or constraint solver. Extreme examples of rule-based programming include the mail system in Unix which uses rules in order to rewrite mail addresses to canonical forms, or the transition rules used in model checkers.

Rule-based programming is currently experiencing a renewed period of growth with the emergence of new concepts and systems that allow a better understanding and better usability. On the theoretical side, after the in-depth study of rewriting concepts during the eighties, the nineties saw the emergence of the general concepts of rewriting logic and of the rewriting calculus. On the practical side, new languages such as ASM, ASF+SDF, BURG, Claire, ELAN,

Maude, and Stratego, new systems such as LRR and commercial products such as Ilog Rules and Eclipse have shown that rules are a useful programming tool.

The practical application of rule-based programming prompts research into the algorithmic complexity and optimization of rule-based programs as well as into the expressivity, semantics and implementation of rules-based languages.

The purpose of this workshop is to bring together researchers from the various communities working on rule-based programming to foster fertilisation between theory and practice, as well as to favour the growth of this programming paradigm.

Program Committee

Program co-chairs:

- Jean-Louis Giavitto (CNRS & Université d'Evry Val d'Essone)
- Pierre-Etienne Moreau (LORIA & INRIA-Lorraine)

Program Committee:

- James Cordy (Queen's University at Kingston, Canada)
- Olivier Danvy (BRICS, Denmark)
- Steven Eker (SRI International, USA)
- Thom Fruehwirth (Ulm University, Germany)
- Berthold Hoffmann (Bremen University, Germany)
- Herbert Kuchen (Muenster University, Germany)
- Oege de Moor (Oxford University Computing Laboratory, England)
- Przemek Prusinkiewicz (Calgary University, Canada)
- Patrick Viry (ILOG, France)

Acknowledgements

We would like to thank the program committee members for their help in evaluating the papers and also Marc Aiguier (LaMI) and Guy Narbonni (IMPLEX) that have done additional reviews and Olivier Michel for its help in the preparation of this document. Furthermore, we would like to thank the RDP organizing committee for taking care of the local organization of our workshop. We thank Elsevier for publishing these proceedings in the Electronic Notes in Theoretical Computer Science (ENTCS) and Professor Michael Mislove for providing and adapting the style files for ENTCS.

We are also grateful to our respective institution for their support: CNRS, INRIA Lorraine, LORIA, and Université d'Evry.

Jean-Louis Giavitto,
Pierre-Etienne Moreau.

Contents

OnDemandOBJ: A Laboratory for Strategy Annotations [1]

María Alpuente[†] [2] Santiago Escobar[†] [3] Salvador Lucas[†] [4]

[†] *DSIC, UPV, Camino de Vera s/n, 46022 Valencia, Spain.*

Abstract

Strategy annotations are used in rule-based programming languages such as OBJ2, OBJ3, CafeOBJ, and Maude to improve efficiency and/or reduce the risk of nontermination. Syntactically, they are given either as lists of natural numbers or as lists of integers associated to function symbols whose (absolute) values refer to the arguments of the corresponding symbol. A positive index enables the evaluation of an argument whereas a negative index means "evaluate on-demand". In this paper, we present OnDemandOBJ, an implementation of strategy-guided on-demand evaluation, which improves previous mechanizations that were lacking satisfactory computational properties.

1 Introduction

Eager rule-based programming languages such as Lisp, OBJ*, CafeOBJ, ELAN, or Maude evaluate functional expressions by innermost rewriting. Since nontermination is a known problem of innermost reduction, *syntactic annotations* (generally specified as sequences of integers associated to function arguments, called *local strategies*) have been used in OBJ2 [9], OBJ3 [11], CafeOBJ [10], and Maude [6] to improve efficiency and (hopefully) avoid nontermination. A local strategy for a k-ary symbol $f \in \mathcal{F}$ is a sequence $\varphi(f)$ of integers taken from $\{-k, \ldots, -1, 0, 1, \ldots, k\}$ which are given in parentheses. Local strategies are used in OBJ programs [5] for guiding the *evaluation strategy* (abbr. E-strategy): when considering a function call $f(t_1, \ldots, t_k)$, if annotation i appears in the local strategy, then the subterm at

[1] Work partially supported by CICYT TIC2001-2705-C03-01 and MCYT grants HA2001-0059 and HU2001-0019.

[2] Email:alpuente@dsic.upv.es, URL:http://www.dsic.upv.es/users/elp/alpuente.html

[3] Email:sescobar@dsic.upv.es, URL:http://www.dsic.upv.es/users/elp/sescobar.html

[4] Email:slucas@dsic.upv.es, URL:http://www.dsic.upv.es/users/elp/slucas.html

[5] As in [11], by OBJ we mean OBJ2, OBJ3, CafeOBJ, or Maude.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

argument i is evaluated. If 0 is found, then the evaluation of f is attempted. A mapping φ that associates a local strategy $\varphi(f)$ to every $f \in \mathcal{F}$ is called an E-strategy map [17,18]. Whenever the user provides no local strategy for a given symbol, the (Maude, OBJ*, CafeOBJ) interpreter automatically assigns a *default* E-strategy. We adopt the default local strategy of Maude which associates $(1\ 2 \cdots k\ 0)$ to each k-ary symbol f having no explicit strategy, i.e. all arguments are marked as evaluable.

Example 1.1 Consider the following Maude program `pi` which codifies the well-known infinite series expansion to approximate number π:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

```
obj PI is
  sorts Nat LNat Recip LRecip .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op posrecip : Nat -> Recip .
  op negrecip : Nat -> Recip .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat .
  op rnil : -> LRecip .
  op rcons : Recip LRecip -> LRecip .
  op from : Nat -> LNat .
  op seriepos : Nat LNat -> LRecip .
  op serieneg : Nat LNat -> LRecip .
  op pi : Nat -> LRecip .
  vars N X Y : Nat . var Z : LNat .
  eq from(X) = cons(X,from(s(X))) .
  eq seriepos(0,Z) = rnil .
  eq seriepos(s(N),cons(X,cons(Y,Z))) =
                              rcons(posrecip(Y),serieneg(N,Z)) .
  eq serieneg(0,Z) = rnil .
  eq serieneg(s(N),cons(X,cons(Y,Z))) =
                              rcons(negrecip(Y),seriepos(N,Z)) .
  eq pi(X) = seriepos(X,from(0)) .
endo
```

A term[6] `pi(2)` approximates the number $\pi/4$ using 2 elements of the series expansion, i.e. the intended behavior is

$$\texttt{pi(2)} \rightarrow^* \texttt{rcons(posrecip(1),rcons(negrecip(3),rnil))}$$

where `posrecip(`n`)` denotes the positive reciprocal $1/n$ and `negrecip(`n`)` denotes $-1/n$. Since the Maude interpreter associates a default local strategy

[6] Naturals 1, 2, ... are used as a shorthand to numbers $\mathtt{s}^n(0)$ where $n = 1, 2, \ldots$.

$(1\ 2 \cdots k\ 0)$ to each k-ary symbol f having no explicit strategy, all arguments are marked as evaluable. However, this program is non-terminating since innermost evaluation diverges:

```
pi(2) → seriepos(2,from(0))
      → seriepos(2,cons(0,from(1)))
      → seriepos(2,cons(0,cons(1,from(2)))) → ···
```

In order to avoid non-termination, annotation 2 should be removed from the (default) local strategy `(1 2 0)` for symbol `cons`.

Example 1.2 After removing annotation 2 from the local strategy for symbol `cons` in Example 1.1, the program becomes terminating under innermost rewriting with such restriction. The unique change in the program of Example 1.1 is:

```
op cons : Nat LNat -> LNat [strat (1)] .
```

Unfortunately, this restriction of rewriting can have a negative impact in the ability to compute normal forms.

Example 1.3 The evaluation of `pi(2)` using the program of Example 1.2 yields:

```
pi(2) → seriepos(2,from(0)) → seriepos(2,cons(0,from(1)))
```

The evaluation stops at this point since reductions on the second argument of `cons` are disallowed. Note that a further step

```
seriepos(2,cons(0,from(1))) → seriepos(2,cons(0,cons(1,from(2))))
```

is required in order to apply the second rule of `seriepos` and be able to obtain the intended normal form of Example 1.1.

The handicaps of using only positive annotations regarding correctness and completeness of computations are discussed in [1,2,14,15,18,19]: essentially, the problem is that the absence of some indices in the local strategies can have a negative impact in the ability of such strategies to compute normal forms.

In [18,19], *negative* indices are proposed to indicate those arguments that should be evaluated only 'on-demand', where the 'demand' is an attempt to match an argument term with the left-hand side of a rewrite rule [7,11,19]. For instance, subterm `from(1)` in Example 1.2 is *demanded* by the second rule of `seriepos`. Thus, `(1 -2)` would be the apt local strategy for `cons` as pointed out in [18]; i.e. the first argument can be always evaluated but the second argument can be evaluated only "on-demand". The evaluation of `cons` under strategy `(1 -2)` is able to normalize `pi(2)` to its intended normal form without entering in a non-terminating evaluation, whereas evaluation only with positive annotations enters an infinite derivation or does not provide the intended normal form.

On-demand strategy annotations have not been implemented to date: even if negative annotations are (syntactically) accepted in current OBJ

implementations, namely OBJ3 and Maude, unfortunately they do not have the expected (on-demand) effect over the computations.

Example 1.4 Consider the program of Example 1.1 where the local strategy for `cons` includes the on-demand annotation `-2`. The unique change to the program is:

```
op cons : Nat LNat -> LNat [strat (1 -2)] .
```

The OBJ3 interpreter does not implement negative (on-demand) annotations though does *accept* this program and the evaluation of `pi(2)` surprisingly delivers the very same result as in Example 1.3. That is, the negative annotation is just disregarded by the OBJ3 interpreter (which, in this case, causes loss of completeness). On the other hand, the Maude interpreter does not implement negative annotations but also accepts this program and the evaluation of the same expression diverges as in Example 1.1. This is because the negative annotation `-2` is interpreted as a positive one thus resulting in non-termination.

On the other hand, CafeOBJ is able to deal with negative annotations using the on-demand evaluation model of [18] and is able to compute the intended value `rcons(posrecip(1),rcons(negrecip(3),rnil))` of Example 1.1. However, in [1] we discussed a number of problems of the on-demand evaluation model of [18,19], as shown in the following example.

Example 1.5 [1] Consider the following OBJ program:

```
obj LENGTH is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat  [strat (1)] .
  op from : Nat -> LNat .
  op length : LNat -> Nat     [strat (0)] .
  op length' : LNat -> Nat    [strat (-1 0)] .
  vars X Y : Nat . var Z : LNat .
  eq from(X) = cons(X,from(s(X))) .
  eq length(nil) = 0 .
  eq length(cons(X,Z)) = s(length'(Z)) .
  eq length'(Z) = length(Z) .
endo
```

The expression `length'(from(0))` is rewritten (in one step) to `length(from(0))`. No evaluation is demanded on the argument of `length'` for enabling this step (the negative annotation `-1` is included for `length'` but the corresponding rule includes a variable at the first argument of `length'`) and no further evaluation on `length(from(0))` should be performed (due to the local strategy `(0)` of `length` which forbids evaluation on any argument of `length`). However, the annotation `-1` of function `length'` is treated in such a way by the operational model of [19,18] that the on-demand evaluation of the expression

`length'(from(0))` yields an infinite evaluation sequence (see [1] for a more detailed explanation). For instance, CafeOBJ[7] ends with a stack overflow:

```
LENGTH> red length'(from(0)) .
-- reduce in LENGTH : length'(from(0))
Error: Stack overflow (signal 1000)
```

In [1] we proposed a solution to these problems in order to cope with on-demand strategy annotations, which is based on a suitable extension of the E-evaluation strategy of OBJ-like languages which only considers annotations given as natural numbers. Our strategy incorporates a better treatment of demandness and also enjoys good computational properties; in particular, we show how it can be used for computing (head-)normal forms and we prove it is conservative w.r.t. other on-demand strategies: *lazy rewriting* [8] and *on-demand rewriting* [14]. A program transformation for proving termination of the on-demand evaluation strategy was also formalized, which relies on standard techniques.

In this paper, we address the implementation of on-demand evaluation strategy of [1] together with different techniques related to managing on-demand strategy annotations. This system is called OnDemandOBJ.

2 OnDemandOBJ

In order to demonstrate the practicality of our ideas, an interpreter of the computational model described in [1] has been implemented in Haskell (using GHC 5.04.1). The system is called OnDemandOBJ and is publicly available at

`http://www.dsic.upv.es/users/elp/soft.html`

2.1 Programs

The prototype implements a subset of the Maude and CafeOBJ syntax, i.e. admits programs typed in either one of the two syntaxes. The BNF grammars associated to such syntax subsets are included in the distribution. Default strategy annotations are considered as in Maude, i.e. the default local strategy associated to a k-ary symbol f, is `(1 2 ... k 0)`. The prototype does not provides a prelude set of functions or operators as in Maude or CafeOBJ, i.e. `if_then_else` function is not directly available.

2.2 Evaluation

The evaluation of an expression according to the computational model described in [7,17] is available through the command `red`. This command is also available in OBJ2, OBJ3, CafeOBJ, or Maude.

[7] We use the CafeOBJ interpreter (version 1.3.1) available at `http://www.ldl.jaist.ac.jp/Research/CafeOBJ/system.html`.

If reductions on some arguments are constrained by means of strategy annotations, command `red` can fail to obtain the desired normal forms. Expressions obtained by `red` are called E-normal forms. Conditions ensuring that E-normal forms are (at least) head-normal forms have been investigated in [14,18]. In order to be able to obtain normal forms once head-normal forms are obtained, the OnDemandOBJ prototype provides a novel command `norm` which calculates normal forms following the *normalization via μ-normalization* process described in [14]. Informally speaking, once the E-normal forms have been obtained, the evaluation process starts on those positions which were not allowed for reduction. The following example explains how normalization works.

Example 2.1 Consider the problem of selecting a collection of prime numbers. The following program codifies such problem where the expression `primes` is intended to arbitrarily approximate the list of prime numbers (see [12]).

```
obj SEL-FIRST-PRIMES is
  sorts Nat LNat .
  op 0     : -> Nat .
  op s     : Nat -> Nat .
  ops nil serieprimes : -> LNat .
  op cons  : Nat LNat -> LNat [strat (1)] .
  op first : Nat LNat -> LNat .
  op nats primes : Nat -> LNat .
  op sieve : LNat -> LNat .
  op filter : LNat Nat Nat -> LNat .
  vars X Y M N : Nat . var Z : LNat .
  eq filter(cons(X,Z),0,M) = cons(0,filter(Z,M,M)) .
  eq filter(cons(X,Z),s(N),M) = cons(X,filter(Z,N,M)) .
  eq sieve(cons(0,Z)) = sieve(Z) .
  eq sieve(cons(s(N),Z)) = cons(s(N),sieve(filter(Z,N,N))) .
  eq nats(N) = cons(N,nats(s(N))) .
  eq serieprimes = sieve(nats(s(s(0)))) .
  eq first(0,Z) = nil .
  eq first(s(X),cons(Y,Z)) = cons(Y,first(X,Z)) .
  eq primes(N) = first(N,serieprimes) .
endo
```

The intended behavior is `primes(3)` $\rightarrow^*$ `cons(2,cons(3,cons(5,nil)))`. Note that in order to avoid non-termination, the strategy for symbol `cons` does not include annotation 2, as in Example 1.3. The program is not complete and some normalizations are not available. For instance, expression `primes(3)` is evaluated as follows:

```
Maude> red primes(s(s(s(0)))).
reduce in SEL-FIRST-PRIMES : primes(s(s(s(0)))) .
rewrites: 5 in -1ms cpu (0ms real) (~ rewrites/second)
result LNat: cons(s(s(0)),first(s(s(0)),sieve(...,s(0),s(0))))
```

Note also that annotation `-2` for symbol `cons` does not solve this problem

since the second argument of `cons` is a variable in every lhs of the program. However, when `norm` is used the evaluation is restarted on the maximal non-evaluated subterms in order to produce an actual normal form, i.e. on subterm `first(s(s(0)),sieve(...,s(0),s(0)))`. For instance, when the previous expression is evaluated using OnDemandOBJ, we obtain:

```
SEL-FIRST-PRIMES> norm primes(s(s(s(0)))).
Normal form: cons(s(s(0)),cons(s(s(s(0))),cons(s(s(s(s(s(0))))),nil)))
{ 0.0000 sec., 29 rewrites }
```

2.3 Transformations

In the following, we recall two program transformations integrated into OnDemandOBJ.

2.3.1 Removing negative annotations

In [3] we introduced an automatic, semantics–preserving program transformation which produces a program (without negative annotations) which can be then correctly executed by typical OBJ interpreters. The idea is to encode the 'on-demand' strategy instrumented by the negative annotations within new function symbols (and corresponding program rules) that only use positive strategy annotations. Command `trNeg` of the OnDemandOBJ prototype applies this program transformation to eliminate on-demand annotations (negative indices) from an annotated program (see [3]) and then loads the new transformed program for evaluation. The following example explains how this program transformation works.

Example 2.2 Consider the program of Example 1.4. The program obtained after the transformation of [3] is the following:

```
obj PINoNeg is
  sorts Nat LNat Recip LRecip .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op posrecip : Nat -> Recip .
  op negrecip : Nat -> Recip .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat [strat (0)] .
  op cons-+2 : Nat LNat -> LNat [strat (2)] .
  op rnil : -> LRecip .
  op rcons : Recip LRecip -> LRecip .
  op from : Nat -> LNat .
  op seriepos : Nat LNat -> LRecip .
  op seriepos-+2 : Nat LNat -> LRecip [strat (2 0)] .
  op serieneg : Nat LNat -> LRecip .
  op serieneg-+2 : Nat LNat -> LRecip [strat (2 0)] .
  op pi : Nat -> LRecip .
  op quoteNat : Nat -> Nat [strat (0)] .
  op quoteLNat : LNat -> LNat [strat (0)] .
```

```
  op quoteRecip : Recip -> Recip [strat (0)] .
  op quoteLRecip : LRecip -> LRecip [strat (0)] .
  vars N X Y : Nat . var Z : LNat . var R : Recip . var L : LRecip .
  eq from(X) = quoteLNat(cons(X,from(s(X)))) .
  eq seriepos(0,Z) = quoteLRecip(rnil) .
  eq seriepos(s(N),cons(X,Z)) = seriepos-+2(s(N),cons-+2(X,Z)) .
  eq seriepos-+2(s(N),cons-+2(X,cons(Y,Z))) =
        quoteLRecip(rcons(posrecip(Y),serieneg(N,Z))) .
  eq serieneg(0,Z) = quoteLRecip(rnil) .
  eq serieneg(s(N),cons(X,Z)) = serieneg-+2(s(N),cons-+2(X,Z)) .
  eq serieneg-+2(s(N),cons-+2(X,cons(Y,Z))) =
        quoteLRecip(rcons(negrecip(Y),seriepos(N,Z))) .
  eq pi(X) = quoteLRecip(seriepos(X,from(0))) .
  eq quoteNat(0) = 0 .
  eq quoteNat(s(N)) = s(N) .
  eq quoteRecip(posrecip(N)) = posrecip(N) .
  eq quoteRecip(negrecip(N)) = negrecip(N) .
  eq quoteLNat(nil) = nil .
  eq quoteLNat(cons(X,Z)) = cons(quoteNat(X),Z) .
  eq quoteLNat(from(X)) = from(X) .
  eq quoteLRecip(rnil) = rnil .
  eq quoteLRecip(rcons(R,L)) = rcons(quoteRecip(R),quoteLRecip(L)) .
  eq quoteLRecip(seriepos(X,Z)) = seriepos(quoteNat(X),quoteLNat(Z)) .
  eq quoteLRecip(serieneg(X,Z)) = serieneg(quoteNat(X),quoteLNat(Z)) .
  eq quoteLRecip(pi(X)) = pi(X) .
endo
```

Informally, new symbols `seriepos-+2`, `serieneg-+2` and `cons-+2` are introduced to enable the evaluation of the second argument of `cons` in those positions which could be eventually demanded. Note that the rules for symbols `seriepos` and `serieneg` in Example 1.4 are the only one which could demand the evaluation of the second argument of `cons`. The extra symbols `quote` are introduced to preserve correctness w.r.t. reductions with positive indices. Now, term `pi(2)` is correctly evaluable using the Maude interpreter (simulating the on-demand evaluation of [1])

```
Maude> red quoteLRecip(pi(s(s(0)))).
reduce in PINoNeg : quoteLRecip(pi(s(s(0)))) .
rewrites: 33 in -1ms cpu (0ms real) (~ rewrites/second)
result LRecip: rcons(posrecip(s(0)), rcons(negrecip(s(s(s(0)))), rnil))
```

In OnDemandOBJ, the execution of the program transformation and the reduction of term `pi(2)` works as follows:

```
PI> trNeg
Module PI successfully transformed
Module PINoNeg loaded
PINoNeg> red quoteLRecip(pi(s(s(0)))).
Normal form: rcons(posrecip(s(0)),rcons(negrecip(s(s(s(0)))),rnil))
{ 0.0020 sec., 33 rewrites }
```

2.3.2 Ensuring constructor normal forms

In [2] we defined a program transformation methodology for (correct and) complete evaluations which applies to OBJ-like languages. We ascertain the conditions (on an strategy φ) ensuring that OBJ programs using strategy annotations do compute the value (i.e., the constructor normal form) of any given expression.

The following example explains how this program transformation works.

Example 2.3 [2] Consider the following OBJ program:

```
obj EXAMPLE is
  sorts Nat LNat .
  op 0     : -> Nat .
  op s     : Nat -> Nat .
  op nil   : -> LNat .
  op cons  : Nat LNat -> LNat [strat (1)] .
  op from  : Nat -> LNat .
  op sel   : Nat LNat -> Nat .
  op first : Nat LNat -> LNat .
  vars X Y : Nat . var Z : LNat .
  eq sel(s(X),cons(Y,Z)) = sel(X,Z) .
  eq sel(0,cons(X,Z)) = X .
  eq first(0,Z) = nil .
  eq first(s(X),cons(Y,Z)) = cons(Y,first(X,Z)) .
  eq from(X) = cons(X,from(s(X))) .
endo
```

The evaluation of expression $t =$ `first(s(0),from(0))` of sort `LNat` yields (we use the OBJ3 interpreter –version 2.0–):

```
Maude> reduce first(s(0),from(0)) .
reduce in EXAMPLE : first(s(0), from(0)) .
rewrites: 2 in -10ms cpu (0ms real) (~ rewrites/second)
result LNat: cons(0, first(0, from(s(0))))
```

Note that `cons(0,first(0,from(s(0))))` is not a normal form. However, $t \to^*$ `cons(0,nil)` $\in \mathcal{T}(\mathcal{C}, \mathcal{X})$, i.e., `cons(0,nil)` is a value of t which cannot be obtained by using the OBJ interpreter. The application of the program transformation of [2] produces the following program:

```
obj EXAMPLE-TR is
  sorts Nat LNat .
  ops 0 0' : -> Nat .
  ops s s' : Nat -> Nat .
  ops nil nil' : -> LNat .
  op cons : Nat LNat -> LNat [strat (1)] .
  op cons' fcons : Nat LNat -> LNat .
  op from : Nat -> LNat .
  ops sel sel' : Nat LNat -> Nat .
  ops first first' : Nat LNat -> LNat .
  op quoteNat : Nat -> Nat [strat (0)] .
  op unquoteNat : Nat -> Nat .
  op quoteLNat : LNat -> LNat [strat (0)] .
```

```
  op unquoteLNat : LNat -> LNat .
  vars X Y : Nat . var Z : LNat .
  eq sel(s(X),cons(Y,Z)) = sel(X,Z) .
  eq sel(0,cons(X,Z)) = X .
  eq first(0,Z) = nil .
  eq first(s(X),cons(Y,Z)) = cons(Y,first(X,Z)) .
  eq from(X) = cons(X,from(s(X))) .
  eq sel'(s(X),cons(Y,Z)) = sel'(X,Z) .
  eq sel'(0,cons(X,Z)) = quoteNat(X) .
  eq first'(0,Z) = nil' .
  eq first'(s(X),cons(Y,Z)) = cons'(quoteNat(Y),first'(X,Z)) .
  eq quoteNat(0) = 0' .
  eq quoteLNat(cons(X,Z)) = cons'(quoteNat(X),quoteLNat(Z)) .
  eq quoteLNat(nil) = nil' .
  eq quoteNat(s(X)) = s'(quoteNat(X)) .
  eq quoteNat(sel(X,Z)) = sel'(X,Z) .
  eq quoteLNat(first(X,Z)) = first'(X,Z) .
  eq unquoteNat(0') = 0 .
  eq unquoteNat(s'(X)) = s(unquoteNat(X)) .
  eq unquoteLNat(nil') = nil .
  eq unquoteLNat(cons'(X,Z)) = fcons(unquoteNat(X),unquoteLNat(Z)) .
  eq fcons(X,Z) = cons(X,Z) .
endo
```

Informally, all constructors and defined symbols which participate in the sort `LNat` of the goal term are duplicated in order to enable reduction on the second argument of `cons`. Symbols `quote` translate from original symbols to duplicated symbols and symbol `unquote` translates back to original symbols. Now, the evaluation of `unquoteLNat(quoteLNat(first(s(0),from(0))))` yields:

```
Maude> reduce unquoteLNat(quoteLNat(first(s(0), from(0)))) .
reduce in EXAMPLE-TR : unquoteLNat(quoteLNat(first(s(0), from(0)))) .
rewrites: 10 in -10ms cpu (0ms real) (~ rewrites/second)
result LNat: cons(0, nil)
```

This procedure is implemented in OnDemandOBJ by using command `eval` which applies the transformation of [2] and automatically quotes/unquotes the input expression.

```
EXAMPLE> eval first(s(0),from(0)).
Normal form: cons(0, nil)
{ 0.0000 sec., 10 rewrites }
```

3 Experiments

Tables 5.1, 5.2, and 3 show the runtimes [8] and the number of rewrite steps of a selection of benchmarks for the different OBJ-family systems. These experimental results, together with the OBJ source programs, are available at

`http://www.dsic.upv.es/users/elp/ondemandOBJ/experiments`

CafeOBJ is developed in Lisp at the Japan Advanced Inst. of Science and Technology (JAIST); OBJ3, also written in Lisp, is maintained by the University of California at San Diego; Maude is developed in C++ and maintained by the Computer Science Lab at SRI International. OBJ3 and Maude provide only computations with positive annotations whereas CafeOBJ provides computations with negative annotations as well, using the on-demand evaluation of [18,19]. OnDemandOBJ computes with negative annotations using the on-demand evaluation strategy provided in this paper. Note that CafeOBJ and OBJ3 implement sharing of variables whereas Maude and OnDemandOBJ do not. It is worth noting that mark *overflow* indicates that the execution raised a memory overflow and normal form was not achieved; whereas mark *unavailable* indicates that the program can not be executed in such OBJ implementation.

The benchmark `pi` codifies the program of Example 1.4. It is worth noting that uses negative annotations to obtain a terminating and complete example, which can not be obtained by using only positive annotations. Termination of the program can be formally proved using the technique of [1]. Also, by using the results in [2], we can guarantee that every expression such as `pi(`n`)` for some n of sort `Nat` produces (as expected) a completely evaluated expression of sort `LRecip`. Table 5.1 compares the evaluation of expression [9] `pi(square(square(3)))` using existing OBJ implementations. It demonstrates that negative annotations are actually useful in practice and that the implementation of the on-demand evaluation strategy in other systems is quite promising.

On the other hand, Table 5.2 illustrates the interest of using negative annotations to improve the behavior of programs: The benchmark `msquare_eager` codifies the functions `square`, `minus`, `times`, and `plus` over natural numbers using only positive annotations. This benchmark is called `eager` because every k-ary symbol f is given a strategy $(1\ 2\ \cdots k\ 0)$ (this corresponds to *default* strategies in OnDemandOBJ). Note that the program is terminating as a TRS (i.e., without any annotation). The benchmark `msquare_apt` is similar to `msquare_eager`, but *canonical positive* strategies are provided: the i-th argument of a symbol f is annotated with a positive index if there is

[8] The average of 10 executions measured in a Pentium III 350 Mhz machine with 256 Mbytes running RedHat 7.2.

[9] Rules for function `square` are not included in Example 1.4 but can be found at `http://www.dsic.upv.es/users/elp/ondemandOBJ/experiments`.

ms./rewrites	`pi`
OnDemandOBJ	25/364
CafeOBJ	30/364
OBJ3	*unavailable*
Maude	*unavailable*

Table 1
Execution of call `pi(square(square(3)))`

an occurrence of f in the left-hand side of a rule having a non-variable i-th argument; otherwise, the argument is not annotated (see [4]). The benchmark `msquare_neg` is similar to `msquare_apt`, though *canonical arbitrary* strategies are provided: now (from left-to-right), the i-th argument of a defined symbol f is annotated with a positive index i if all occurrences of f in the left-hand side of the rules contain a non-variable i-th argument; if all occurrences of f in the left-hand side of the rules have a variable i-th argument, then the argument is not annotated; in any other case, annotation $-i$ is given to f (see [4]). Then, for instance, program `msquare_neg` runs in less time and requires a smaller number of rewrite steps than `msquare_eager` or `msquare_apt`, which do not include negative annotations. Note the difference in rewrite steps of benchmarks `msquare_eager` and `msquare_apt`, which is due to sharing of variables (remember that OnDemandOBJ and Maude do not implement sharing of variables and this is only meaningful in lazy evaluation).

Finally, Table 3 compares the execution of typical functional programs with canonical arbitrary strategies in OnDemandOBJ and in CafeOBJ, and demonstrates that there are clear advantages in using our implementation of the on-demand evaluation. We have used benchmarks `quicksort`, `minsort`, `mod`, and `average` which are borrowed from [5], and use canonical arbitrary strategies. Benchmark `mod'` is similar to `mod` but extra annotations are provided in order to avoid differences due to sharing (again, OnDemandOBJ does not implemente sharing of variables).

4 Conclusions

In this paper we have addressed the implementation of the on-demand evaluation strategy of [1] together with different techniques related to managing on-demand strategy annotations. For instance, we provide a novel command `norm` which calculates normal forms following the *normalization via μ-normalization* process described in [14] (see Section 2.2) and two program transformations integrated into OnDemandOBJ. One program transformation for encoding the on-demand strategy instrumented by the negative annotations within new function symbols (and corresponding program rules) that only use positive

ms./rewrites	msquare_eager	msquare_apt	msquare_neg
OnDemandOBJ	33/ 715 40/ 914	62/ 1640 78/ 1992	0/ 1 80/ 1992
CafeOBJ	40/ 715 50/ 914	50/ 715 60/ 914	0/ 1 60/ 914
OBJ3	20/ 715 30/ 914	*overflow* *overflow*	*unavailable* *unavailable*
Maude	0/ 715 0/ 914	0/ 1640 3/ 1992	*unavailable* *unavailable*

Table 2
Execution of terms `minus(0,square(square(5)))` and `minus(square(square(5)),square(square(3)))`

ms./rewrites	quicksort	minsort	mod	mod'	average
OnDemandOBJ	55/1373	87/1649	540/13661	135/3117	70/1399
CafeOBJ	42/ 658	*overflow*	180/ 3117	175/3117	130/1399

Table 3
Comparison of CafeOBJ and OnDemandOBJ

strategy annotations (see Section 2.3.1) and the other program transformation for ensuring (correct and) complete evaluations within OBJ programs using strategy annotations to compute the value (i.e., the constructor normal form) of any given expression (see Section 2.3.2). This new features apply to OBJ-like languages in general.

References

[1] M. Alpuente, S. Escobar, B. Gramlich, and S. Lucas. Improving on-demand strategy annotations. In M. Baaz and A. Voronkov, editors, *Proc. 9th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'02)*, volume 2514 of *Lecture Notes in Computer Science*, pages 1–18, Tbilisi, Georgia, 2002. Springer-Verlag, Berlin.

[2] M. Alpuente, S. Escobar, and S. Lucas. Correct and complete (positive) strategy annotations for OBJ. In *Electronic Notes in Theoretical Computer Science*, volume 71. Elsevier Sciences Publisher, 2002.

[3] M. Alpuente, S. Escobar, and S. Lucas. On-demand evaluation by program transformation. In *Electronic Notes in Theoretical Computer Science*, volume 86.2. Elsevier Sciences Publisher, 2003.

[4] S. Antoy and S. Lucas. Demandness in rewriting and narrowing. In *Proc. of 11th International Workshop on Functional and (Constraint) Logic Programming WFLP'02*, volume 76 of *Electronic Notes in Theoretical Computer Science*. Elsevier Sciences, 2002.

[5] T. Arts and J. Giesl. A collection of examples for termination of term rewriting using dependency pairs. Technical report, AIB-2001-09, RWTH Aachen, Germany, 2001.

[6] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In *Electronic Notes in Theoretical Computer Science*, volume 4, pages 65–89. Elsevier Sciences Publisher, 1996.

[7] S. Eker. Term rewriting with operator evaluation strategies. In *Electronic Notes in Theoretical Computer Science*, volume 15. Elsevier Sciences Publisher, 2000.

[8] W. Fokkink, J. Kamperman, and P. Walters. Lazy rewriting on eager machinery. *ACM Transactions on Programming Languages and Systems*, 22(1):45–86, 2000.

[9] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proc. of 12th Annual ACM Symp. on Principles of Programming Languages*, pages 52–66. ACM Press, New York, 1985.

[10] K. Futatsugi and A. Nakagawa. An overview of CAFE specification environment – an algebraic approach for creating, verifying, and maintaining formal specification over networks –. In *1st International Conference on Formal Engineering Methods*, 1997.

[11] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000.

[12] J. Kennaway and F. deVries. Infinitary rewriting. In TeReSe, editor, *Term Rewriting Systems*, chapter 11. Cambridge University Press, 2003.

[13] S. Lucas. Termination of on-demand rewriting and termination of obj programs. In *Proc. of 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP'01*, pages 82–93. ACM Press, New York, 2001.

[14] S. Lucas. Context-sensitive rewriting strategies. *Information and Computation*, 178(1):294–343, 2002.

[15] S. Lucas. Lazy rewriting and context-sensitive rewriting. In *Electronic Notes in Theoretical Computer Science*, volume 64. Elsevier Sciences Publisher, 2002.

[16] T. Nagaya. *Reduction Strategies for Term Rewriting Systems*. PhD thesis, School of Information Science, Japan Advanced Institute of Science and Technology, March 1999.

[17] M. Nakamura and K. Ogata. The evaluation strategy for head normal form with and without on-demand flags. In *Electronic Notes in Theoretical Computer Science*, volume 36. Elsevier Sciences Publisher, 2001.

[18] K. Ogata and K. Futatsugi. Operational semantics of rewriting with the on-demand evaluation strategy. In *Proc. of 2000 International Symposium on Applied Computing, SAC'00*, pages 756–763. ACM Press, New York, 2000.

Translating *Combinatory Reduction Systems* into the *Rewriting Calculus*

Clara Bertolissi, and Horatiu Cirstea, and Claude Kirchner

INPL & University Nancy II & INRIA & LORIA
615, rue du Jardin Botanique, BP-101
54602 Villers-lès-Nancy 54506 France
{Clara.Bertolissi,Horatiu.Cirstea,Claude.Kirchner}@loria.fr

Abstract

The last few years have seen the development of the *rewriting calculus* (or rho-calculus, ρCal) that extends first order term rewriting and λ-calculus. The integration of these two latter formalisms has been already handled either by enriching first-order rewriting with higher-order capabilities, like in the *Combinatory Reduction Systems*, or by adding to λ-calculus algebraic features. The different higher-order rewriting systems and the rewriting calculus share similar concepts and have similar applications, and thus, it seems natural to compare these formalisms. We analyze in this paper the relationship between the Rewriting Calculus and the Combinatory Reduction Systems and we present a translation of *CRS*-terms and rewrite rules into rho-terms and we show that for any *CRS*-reduction we have a corresponding rho-reduction.

1 Introduction

Lambda calculus and term rewriting provide two fundamental computational paradigms that had a deep influence on the development of programming and specification languages, and on proof environments. Starting from Klop's ground-breaking work on higher-order rewriting, and because of their complementarity, many frameworks have been designed with a view to integrate these two formalisms.

This integration has been handled either by enriching first-order rewriting with higher-order capabilities or by adding to λ-calculus algebraic features. In the first case, we find the works on CRS [15] and other higher-order rewriting systems [22,19], in the second case the works on combination of λ-calculus with term rewriting [1,4,11] to mention only a few.

For example, the *Combinatory Reduction Systems* (*CRS*), introduced by J.W.Klop [14] are an extension of first order rewrite systems with a mechanism

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

of bound variables like in the λ-calculus. The meta-language of *CRS*, *i.e.* the language in which the notions of substitution and rewrite step are expressed, is based on λ-calculus and higher-order matching (matching modulo the β-rule).

In the same line, the Rewriting Calculus (also called ρCal) [6,7,9], extends first order term rewriting and λ-calculus. Its main design concept is to make all the basic ingredients of rewriting explicit objects, in particular the notions of rule *application* and *result*. By making the rule application explicit, the calculus emphasises on one hand the fundamental role of matching and on the other hand the intrinsic higher-order nature of rewriting. The Rho Calculus offers a broad spectrum of applications due to the two fundamental parameters of the calculus: the theory modulo which matching is performed and the structure under which the results of a rule application are returned. Adjusting these parameters to various situations permits us to easily describe in a uniform but still appropriately tuned manner different calculi, like, for example, lambda calculus, term rewriting and object calculi.

Since the different higher-order rewriting systems and the rewriting calculus share similar concepts and have similar applications, it seems natural to compare these formalisms. The comparison between different higher-order formalisms, like for example between *Combinatory Reduction Systems* and *Higher-order Rewrite Systems*, has already been done [21].

This paper is concerned with the analysis of the relation between the rewriting calculus and higher order rewriting. In particular we study the representation of *Combinatory Reduction Systems* in the rewriting calculus.

This work of analysis and comparison between the ρCal and the *CRS* is meant to better understand the behavior of these systems, in particular how the rewrite rules are applied to terms and how term reductions are performed in the *CRS*. The main contribution of this paper is the definition of a translation of the various *CRS* concepts, like terms, assignments, matching etc. to the corresponding notions of the ρCal, and, using this translation, the proof that every reduction of a *CRS*-term can be reproduced in the ρCal.

The paper is structured as follows: In Section 2 we briefly present the ρCal through its components. In Section 3 we give a description of *CRS*s and we give some examples. In Section 4 we present a translation from *CRS*-terms into ρ-terms and we state a theorem on the correspondence between *CRS*-reductions and ρ-reductions. Section 5 concludes the paper and gives some perspectives to this work.

2 The rewriting calculus

We briefly present in what follows the syntax and the semantics of the ρCal. For a more detailed presentation the reader can refer to [7,9] and for a typed version of the calculus to [3,8].

$$
\begin{array}{lll}
(\rho)\ (t_1 \rightarrowtriangle t_2)t_3 & \rightarrow_\rho & [t_1 \ll t_3].t_2 \\
(\sigma)\ [t_1 \ll t_3].t_2 & \rightarrow_\sigma & \sigma_1(t_2), \ldots, \sigma_n(t_2), \ldots \\
 & & \textit{where } \sigma_i \in \mathit{Sol}(t_1 \lessdot_{\mathbb{T}} t_3) \\
(\delta)\ \ (t_1, t_2)\, t_3 & \rightarrow_\delta & t_1\, t_3, t_2\, t_3
\end{array}
$$

Fig. 1. Small-step reduction semantics

2.1 Syntax

In this paper, the symbols $t, u, \ldots$ range over the set $\mathcal{T}$ of terms, the symbols $X, Y, Z, \ldots, x, y, z \ldots$ range over the infinite set $\mathcal{X}$ of variables (we usually use capitals for free variables) and the symbols $f, g, \ldots$ range over the infinite set $\mathcal{F}$ of constants. All symbols can be indexed. The set of ρ-terms is defined as follows:

$$\mathcal{T} ::= \mathcal{X} \mid \mathcal{F} \mid \mathcal{T} \rightarrowtriangle \mathcal{T} \mid [\mathcal{T} \ll \mathcal{T}].\mathcal{T} \mid \mathcal{T}\ \mathcal{T} \mid \mathcal{T}, \mathcal{T}$$

We assume that the (hidden) application operator to the left, while the other operators associate to the right. The priority of the application operator is higher than that of "$[\ll].$" which is higher than that of the "$\rightarrowtriangle$" which is, in turn, of higher priority than the ",". By abuse of notation, function application is denoted by $t_0(t_1 \cdots t_n)$ instead of $t_0\ t_1 \cdots t_n$.

To support the intuition, we mention here that the application of an abstraction $t_1 \rightarrowtriangle t_3$ to a term t_2 always "fires" and produces as result the term $[t_1 \ll t_2].t_3$ which represents a constrained term where the matching equation is "put on the stack". The body of the constrained term will be evaluated or delayed according to the result of the corresponding matching problem. If a solution exists, the delayed matching constraint self-evaluates to $\sigma(t_3)$, where σ is the solution of the matching between t_1 and t_2. Finally, terms can be grouped together into *structures* built using the symbol ",".

As in any calculus involving binders, we work modulo the "*α-convention*" of Church [5], and modulo the "*hygiene-convention*" of Barendregt [2].

We should mention that there are two operators binding variables: in $t_1 \rightarrowtriangle t_2$, the free variables of t_1 are bound in t_2 and in $[t_1 \ll t_2].t_3$, the free variables of t_1 are bound in t_3 but not in t_2.

2.2 Semantics

The small-step reduction semantics is defined by the reduction rules presented in Figure 1. The central idea of the (ρ) rule of the calculus is that the application of a term $t_1 \rightarrowtriangle t_2$ to a term t_3, reduces to the delayed matching constraint $[t_1 \ll t_3].t_2$, while the application of the (σ) rule consists in solving (modulo the theory $\mathbb{T}$) the matching equation $t_1 \lessdot_{\mathbb{T}} t_3$, and applying the obtained result to the term t_2. The rule (δ) deals with the distributivity of

the application on the structures built with the "," constructor.

According to the matching theory specified [7], the application of the (σ) rule can produce an infinite number of substitutions as result. In the following we will restrict to matching theories leading to a finite (and even unitary) set of substitutions. The substitutions obtained as solution of a matching problem has the form $\sigma = \{x_1/t_1 \dots x_m/t_m\}$ where $Dom(\sigma) = \{x_1, \dots, x_m\}$. The application of a substitution σ to a term t, denoted by $\sigma(t)$ or $t\sigma$, can be straightforwardly adapted to deal with the new forms of constrained terms introduced in the ρCal(see [9]).

As usual, we introduce the classical notions of one-step, many-steps, and congruence relation of $\to_{\rho\sigma\delta}$. Let $\mathsf{Ctx}[-]$ be any context with a single hole, and let $\mathsf{Ctx}[t]$ be the result of filling the hole with the term t. The one-step evaluation $\mapsto_{\rho\sigma\delta}$ is defined by the following inference rules:

$$(\mathsf{Ctx}[-]) \quad \frac{t_1 \to_{\rho\sigma\delta} t_2}{\mathsf{Ctx}[t_1] \mapsto_{\rho\sigma\delta} \mathsf{Ctx}[t_2]} \qquad \forall\ t_1, t_2 \in \mathcal{T}$$

where $\to_{\rho\sigma\delta}$ denotes one of the top-level rules of ρCal. The many-step evaluation $\mapsto\!\!\!\!\to_{\rho\sigma\delta}$ is defined as the reflexive and transitive closure of $\mapsto_{\rho\sigma\delta}$. The congruence relation $=_{\rho\sigma\delta}$ is the symmetric closure of $\mapsto\!\!\!\!\to_{\rho\sigma\delta}$.

Example 2.1 [Small-step reductions] We take the terms

$$(f(\mathcal{X}) \twoheadrightarrow (3 \twoheadrightarrow 3)\mathcal{X})\, f(3) \quad \text{and} \quad (f(\mathcal{X}) \twoheadrightarrow (3 \twoheadrightarrow 3)\mathcal{X})\, f(4)$$

and we show one possible reduction for each of them (corresponding redexes are underlined and only the used rule (*e.g.* $\mapsto_\rho$) is shown instead of $\mapsto_{\rho\sigma\delta}$):

(i) $\underline{(f(\mathcal{X}) \twoheadrightarrow (3 \twoheadrightarrow 3)\mathcal{X})\, f(3)} \mapsto_\rho \underline{(f(\mathcal{X}) \twoheadrightarrow [3 \ll \mathcal{X}].3)\, f(3)} \mapsto_\rho$
$\underline{[f(\mathcal{X}) \ll f(3)].([3 \ll \mathcal{X}].3)} \mapsto_\sigma \underline{[3 \ll 3].3} \mapsto_\sigma 3$

(ii) $\underline{(f(\mathcal{X}) \twoheadrightarrow (3 \twoheadrightarrow 3)\mathcal{X})\, f(4)} \mapsto_\rho \underline{[f(\mathcal{X}) \ll f(4)].((3 \twoheadrightarrow 3)\, \mathcal{X})} \mapsto_\sigma$
$\underline{(3 \twoheadrightarrow 3)\, 4} \mapsto_\rho [3 \ll 4].3$

2.3 Two version of the rewriting calculus

The general ρCal can be instantiated to simpler versions when the syntax is restricted and the matching theory used in the (σ) rule is specified [6]. For example, when all the ρ-rules are of the form $\mathcal{X} \twoheadrightarrow \mathcal{T}$ and a syntactic matching is used, a version similar to the lambda-calculus, denoted ρCal_λ, is obtained. We denote the reductions in the obtained calculus by $\to_{\rho_\lambda}$. Starting from the corresponding congruence relation $=_{\rho_\lambda}$, we can define the matching theory $\mathbb{T}_\lambda$ such that $\mathbb{T}_\lambda \models t_1 = t_2$ *iff* $t_1 =_{\rho_\lambda} t_2$.

More powerful versions of the ρCal can be obtained using more elaborated (decidable) matching theories like, for example, pattern matching [16]. Our goal is to define a version of the ρCal with an evaluation mechanism similar to that of CRSs and thus using a (pattern) higher-order matching. Therefore, we introduce now the notion of ρ-pattern directly inspired from the *CRS*-patterns (defined in Section 3).

Definition 2.2 [ρ-pattern] A ρ-term p is called a *ρ-pattern* if any of its free variables Z appears in a sub-term of p of the form $Z\ x_1 \dots\ x_n$ where the variables $x_1, \dots, x_n$, $n \geqslant 0$, are distinct and all bound in p.

For example, $x \twoheadrightarrow f(Z\ x)$ is a ρ-pattern while $x \twoheadrightarrow (Z\ x\ x)$ and $g(x \twoheadrightarrow x, Z\ x)$ are not.

The $\rho Cal_{\P}$ is then obtained as the ρCal whose rules are of the form $\mathcal{P} \twoheadrightarrow \mathcal{T}$, with $\mathcal{P}$ the set of ρ-patterns, and using a matching modulo $\mathbb{T}_\lambda$ (denoted $\prec_{\P}$). Since pattern matching is decidable and unitary [16] the matching involved in the application of the evaluation rule (σ) of the $\rho Cal_{\P}$ yields a single substitution.

3 Combinatory Reduction Systems

The *Combinatory Reduction Systems* (*CRS*), introduced by J.W.Klop in 1980 [14], are a generalization of first-order term rewrite systems with a mechanism of bound variables like in the λ-calculus. For defining the components of a *CRS* we refer to [15].

3.1 Syntax

In what follows the symbols $A, B, \dots L, P, R, \dots$ range over the set $\mathcal{MT}$ of metaterms, the symbols $t, u, \dots$ range over the set $\mathcal{T}_{\mathcal{CRS}}$ of terms, the symbols $x, y, z, \dots$ range over the set $\mathcal{X}$ of variables, the symbols $X, Y, Z \dots$ range over the set $\mathcal{Z}$ of metavariables of fixed arity and the (functional) symbols $f, g, \dots$ range over the set $\mathcal{F}$ of symbols of fixed arity . All symbols can be indexed. We denote by $\equiv$ the syntactic identity of metaterms or substitutions.

The set of *CRS*-metaterms $\mathcal{MT}$, is defined as follows:

$$\mathcal{MT} ::= \mathcal{X} \mid \mathcal{F}(\mathcal{MT}, \dots, \mathcal{MT}) \mid \mathcal{Z}(\mathcal{MT}, \dots, \mathcal{MT}) \mid [\mathcal{X}]\mathcal{MT}$$

The set $\mathcal{T}_{\mathcal{CRS}} \subset \mathcal{MT}$ of *CRS*-terms is composed of all the metaterms without metavariables.

Comparing to first-order rewrite systems, in the syntax of a *CRS* we have two new concepts: the symbol $[\_]\_$ and the metavariables. The operator $[\_]\_$ denotes an abstraction similar to the λ-abstraction of the λ-calculus such that, in $[x]t$, the variable x is bound in t. The variables bound by $[\_]\_$ may be renamed by α-conversion. Metavariables (in *CRS* rewrite rules) behave as free variables of first-order rewrite systems. Metavariables cannot be bound by the abstraction operator, but can depend on bound variables by means of their arguments. The set of metavariables of a metaterm A is written $\mathcal{MV}(A)$. A metaterm is called *closed* if all its variables occur bound.

A *CRS* rewrite rule is a couple of metaterms. Their metavariables define the reduction schemes since they can be instantiated with the value

of all possible terms. We consider as left-hand side of the rules only the *CRS*-metaterms satisfying the pattern definition:

Definition 3.1 [*CRS*-pattern] A *CRS*-metaterm P is said to be a *CRS pattern* if any of its metavariables Z appears in a sub-metaterm of P of the form $Z(x_1, \ldots, x_n)$ where the variables $x_1, \ldots, x_n$, $n \geqslant 0$, are distinct and all bound in P.

Moreover, the usual conditions used in first-order rewriting are imposed and thus, for a *CRS* rewrite rule $L \to R$ we have: L and R are closed, L has the form $f(A_1, \ldots, A_n)$ with $A_1, \ldots, A_n$ metaterms and $f \in \mathcal{F}$ of arity n, $\mathcal{MV}(L) \supseteq \mathcal{MV}(R)$ and L is a *CRS* pattern. The last restriction ensures the decidability and the uniqueness of the solution of the matching inherent to the application of the *CRS*-rules.

Example 3.2 [β-rule in *CRS*] The β-rule of λ-calculus $(\lambda x.t)u \longrightarrow_\beta t\{x/u\}$ corresponds in *CRS* to the rewrite rule $BetaCRS$:

$$App(Ab([x]Z(x)), Z_1) \to Z(Z_1)$$

where $App \in \mathcal{F}$ of arity 2 and $Ab \in \mathcal{F}$ of arity 1 are the encodings for the application operator and the abstraction operator respectively.

3.2 Semantics

Given a rewrite rule $L \to R$ and a substitution σ (also called assignment, as defined below), we have $\sigma(L) \mapsto_{L\to R} \sigma(R)$ if $\sigma L, \sigma R \in \mathcal{T}_{\mathcal{CRS}}$. The left-hand side and the right-hand side of a *CRS* rewrite rule are metaterms, but the rewrite relation induced by the rule is a relation on terms.

Given a set of *CRS* rewrite rules $\mathcal{R}$, the corresponding one-step relation $\mapsto_{\mathcal{R}}$ (denoted also $\mapsto_{L\to R}$ if we want to specify the applied rule) is the context closure of the the relation induced (as above) by the rules in $\mathcal{R}$. The multi-step evaluation $\mapsto\!\!\!\!\to_{\mathcal{R}}$ is defined as the reflexive and transitive closure of $\mapsto_{\mathcal{R}}$.

The application of substitutions to metavariables is defined at the meta-level of the calculus and uses $\underline{\lambda}$-calculus as meta-language (just for distinguishing it from classical "lambda"). Unintended bindings of variables by the $\underline{\lambda}$-abstractor operator are avoided using α-conversion. The reduction of $\underline{\lambda}$-redexes is performed by the $\underline{\beta}$-rule of the $\underline{\lambda}$-calculus. We denote by $t{\downarrow_{\underline{\beta}}}$ the $\underline{\beta}$-normal form of the term t. We should point out that a *CRS*-term is necessarily in $\underline{\beta}$-normal form.

Performing a substitution in a *CRS* corresponds to applying an *assignment* (and consequently a set of substitutes) to a *CRS*-metaterm.

An n-ary *substitute* [13] is an expression of the form $\xi = \underline{\lambda} x_1 \ldots x_n.u$ where $x_1, \ldots, x_n$ are distinct variables and u is a *CRS*-term ad its application to an n-tuple of *CRS*-terms $(t_1, \ldots, t_n)$ yields the simultaneous substitution $(\underline{\lambda} x_1 \ldots x_n.u)(t_1, \ldots, t_n){\downarrow_{\underline{\beta}}} = u\{x_1 := t_1, \ldots, x_n := t_n\}$

Definition 3.3 [Assignment] An *assignment* $\sigma = \{(Z_1, \xi_1), \ldots, (Z_n, \xi_n)\}$, is a finite set of pairs (metavariable, substitute) such that $\text{arity}(Z_i) = \text{arity}(\xi_i)$ $\forall i \in \{1, \ldots, n\}$. The application of an assignment σ to a *CRS*-metaterm t, denoted $\sigma(t)$ or $t\sigma$, is inductively defined in the following way:

$$\sigma([x]t) \triangleq [x]\sigma(t) \qquad \sigma(f(t_1, \ldots, t_n) \triangleq f(\sigma(t_1), \ldots, \sigma(t_n))$$

$$\sigma(Z_i) \triangleq \xi_i \quad \text{if } (Z_i, \xi_i) \in \sigma \qquad \sigma(Z_i(t_1, \ldots, t_n)) \triangleq \sigma(Z_i)(\sigma(t_1), \ldots, \sigma(t_n))\downarrow_{\underline{\beta}}$$

Therefore the instantiation of rewrite rules in order to obtain an actual rewrite step is defined by replacing each metavariable by a $\underline{\lambda}$-term and by reducing all residuals of $\underline{\beta}$-redexes that are present in the initial term, i.e. performing a development on $\underline{\lambda}$-terms. Since in λ-calculus all developments are finite, the *CRS* substitution is well-defined. Note that the result of the application of an assignment to a metaterm is indeed a term.

To determine the assignment that applied to a metaterm leads to a given term, the concept of matching is used. *CRS* uses higher-order matching but the way the assignment is obtained has not been clearly specified until now.

Example 3.4 Given the *CRS*-term $f(t)$ with $t = App(Ab([x]f(x)), a)$. We apply to the sub-term t the *BetaCRS* rule (Example 3.2). A solution of the corresponding matching problem is the assignment $\sigma = \{(Z, \underline{\lambda}y.fy), (Z_1, a)\}$ since when applying it to the the left-hand side L of the rule *BetaCRS*, we have $\sigma(L) = \sigma(App(Ab([x]Z(x)), Z_1)) = App(Ab([x]\sigma(Z)(x), \sigma(Z_1)) = App(Ab([x](\underline{\lambda}y.fy)(x), a))\downarrow_{\underline{\beta}} = App(Ab([x]f(x)), a) = t$.

We obtain $\sigma(R) = \sigma(Z(Z_1)) = (\sigma(Z))(\sigma(Z_1)) = (\underline{\lambda}y.fy)(a) \downarrow_{\underline{\beta}} = f(a)$ where R is the right-hand side of the rule *BetaCRS*. Therefore we have $t \mapsto_{L \to R} f(a)$ and thus $f(t) \mapsto_{L \to R} f(f(a))$.

4 Translating the *CRS* into the Rewriting Calculus

We propose in this section a translation of *CRS*-(meta)terms into ρ-terms and we show that to *CRS*-reductions correspond ρ-reductions. The assignment application used for performing term reductions in a *CRS* (and thus the matching the *CRS*-reduction relies on) is based on λ-calculus. Consequently, to encode all the expressiveness of *CRS* into the rewriting calculus, we need a greater matching power than the syntactic matching and for this reason we use the $\rho Cal_{\P}$ as target calculus.

In the following we suppose that the set of constants of the considered $\rho Cal_{\P}$ contains the set of functional symbols of the corresponding *CRS* and the set of variables of $\rho Cal_{\P}$ contains the variables and metavariables of the corresponding *CRS*.

Definition 4.1 [Translation] The translation of a *CRS*-metaterm t into a ρ-term, denoted $\overline{t}$ or $< t >$, is inductively defined as follows:

- *CRS*-terms into ρ-terms

$$\overline{x} \triangleq x \qquad \overline{f(t_1,\ldots,t_n)} \triangleq f(\overline{t_1}\ldots\overline{t_n})$$
$$\overline{[x]t} \triangleq x \twoheadrightarrow \overline{t} \qquad \overline{Z(t_1,\ldots,t_n)} \triangleq Z\ \overline{t_1}\ldots\overline{t_n}$$

- *CRS* rewrite rules into ρ-terms:

$$\overline{L \to R} \triangleq \overline{L} \twoheadrightarrow \overline{R}$$

- *CRS* substitutes into ρ-terms:

$$\overline{\underline{\lambda}x_1\ldots x_n.u} \triangleq x_1 \twoheadrightarrow (x_2 \twoheadrightarrow (\ldots(x_n \twoheadrightarrow \overline{u})\ldots))$$

- *CRS* assignments into ρ-substitutions:

$$\overline{\{\ldots,(Z,\underline{\lambda}x_1\ldots x_n.u),\ldots\}} \triangleq \{\ldots, Z/\ x_1 \twoheadrightarrow (x_2 \twoheadrightarrow (\ldots(x_n \twoheadrightarrow \overline{u})\ldots)),\ldots\}$$

We can observe that the translation of the *CRS*-abstraction operator "[ ]" corresponds to the ρ-abstraction operator "$\twoheadrightarrow$". An n-ary *CRS*-metavariable (function) corresponds in the $\rho Cal_{\P}$ to a variable (constant) applied to n ρ-terms.

Since in ρCal rewrite rules are first class objects, a *CRS* rewrite rule is translated into a ρ-term and more precisely into a ρ-rule.

The $\underline{\lambda}$-abstraction operator defined at the meta-level of *CRS* is translated into the ρ-abstraction operator "$\twoheadrightarrow$". This means that reductions performed in *CRS* at the meta-level (using $\underline{\lambda}$) correspond in the $\rho Cal_{\P}$ to explicit reductions corresponding to the application of the abstraction operator "$\twoheadrightarrow$".

The translation of the abstraction operator and of the rewrite rules of a *CRS* into the same abstraction operator of the ρCal corresponds to the uniform treatment of first and higher-order rewriting in the ρCal.

Example 4.2 We have already seen how the β-rule of λ-calculus can be translated into a *CRS*. When translating this *BetaCRS* rule into the rewriting calculus the following term is obtained:

$$\overline{BetaCRS} \triangleq App(Ab(x \twoheadrightarrow (Z\ x)), Z_1) \twoheadrightarrow Z\ Z_1$$

where $App, Ab \in \mathcal{F}$ and $x, Z, Z_1 \in \mathcal{X}$.

The *CRS*-abstraction operator is never directly applied to a *CRS*-term, since we have no application symbols in the syntax of *CRS*. Nevertheless it is translated into the ρ-abstraction operator ensuring that the corresponding variables are bound in the translation and thus, the preservation of the pattern condition by the translation.

Lemma 4.3 *Given a* CRS*-metaterm L. If L satisfies the* CRS *pattern definition, then $\overline{L}$ satisfies the ρ-pattern definition.*

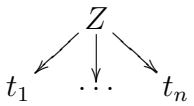
Proof. Follows immediately from Definition 4.1 and Definition 2.2. □

Moreover, as a consequence of the definition of the translation, we have that every ρ-term obtained from a *CRS*-metaterm does not contain any redexes and thus, it is in normal form.

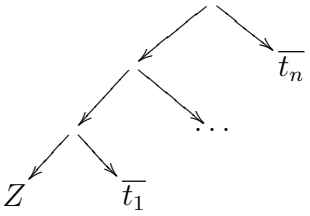
We show in the following that we can express *CRS*-derivations in the rewriting calculus using the translation we have proposed above. We state some lemmas and the main theorems and we give an example of *CRS*-reduction and its corresponding one in the $\rho Cal_{\P}$.

First of all we need some notation facilities. For any metaterm A we denote by $\mathcal{P}os(A)$ the set of all possible positions in A. We call ϵ the head position of A. $A(\omega)$ is the symbol at the position ω in A. A sub-metaterm of A at the position $\omega \in \mathcal{P}os(A)$ is denoted $A_{|\omega}$ and defined by $\forall \omega.\omega' \in \mathcal{P}os(A), \omega' \in \mathcal{P}os(A_{|\omega}), A_{|\omega}(\omega') = t(\omega.\omega')$. We use the notation $A_{\lceil B \rceil_\omega}$ to signify that A has a sub-metaterm B at the position ω. To simplify the notation we write $A(\omega.i^n.\omega')$ for $A(\omega.\underbrace{(i \ldots i)}_{n}.\omega')$.

The position ω of a sub-metaterm B in a *CRS*-metaterm $A_{\lceil B \rceil_\omega}$ and the position of its translation $\overline{B}$ in the ρ-term $\overline{A}$ are not the same. This is due to the different use of (meta)variables and functional symbols in the *CRS*- and ρ-(meta)terms. Indeed, since an n-ary metavariable and an n-ary function are translated in the $\rho Cal_{\P}$ as a variable or function applied to n ρ-terms, the n-ary tree of the *CRS*-metaterm



becomes a binary tree in ρCal



The following function $tr(\_,\_)$ defines the changing of position after the translation.

Definition 4.4 [Position transformation] Let A be a *CRS*-metaterm and $\omega \in \mathcal{P}os(A)$ a position in the metaterm A. The transformation function $tr(\omega, A) : \mathcal{P}os(A) \times \mathcal{MT} \mapsto \mathcal{P}os(\overline{A})$ is inductively defined in the following way:

- $tr(\epsilon, A) = \epsilon$
- $tr(n - i.\omega, Z(A_1, \ldots, A_n)) = 1^i.2.tr(\omega, A_{n-i})$ where $i \in \{0, \ldots, n-1\}$
- $tr(n - i.\omega, f(A_1, \ldots, A_n)) = 1^i.2.tr(\omega, A_{n-i})$ where $i \in \{0, \ldots, n-1\}$
- $tr(i.\omega, [x]A) = 1.\epsilon$ if $i = 1$ and $\omega = \epsilon$
  $= i.tr(\omega, A)$ if $i = 2$
  $= wrong$ in all other cases

The correctness of this function w.r.t. the translation can be shown by structural induction on the*CRS*-metaterm:

Lemma 4.5 *Let $A_{\lceil B \rceil_\omega}$ be a* CRS*-metaterm with a sub-term B at the position ω. Then:*

$$\overline{A_{\lceil B \rceil_\omega}} = \overline{A}_{\lceil \overline{B} \rceil_{tr(\omega,A)}}$$

We show that the translation "preserves" the matching solution, *i.e.* for every assignment σ permitting the application of a *CRS* rewrite rule to a *CRS*-term, the translation of the rule into the $\rho Cal_{\P}$ can be applied to the translation of the corresponding term using the corresponding substitution.

Lemma 4.6 *Let A be a* CRS*-metaterm and σ an assignment. Then*

$$\overline{\sigma}(\overline{A}) \ \mapsto\!\!\!\!\!\rightarrow_{\rho_\lambda} \ \overline{\sigma(A)}$$

Proof. By structural induction on the term A. □

As an immediate consequence we obtain as well $\overline{\sigma}(\overline{A}) \ \mapsto\!\!\!\!\!\rightarrow_{\rho\delta} \ \overline{\sigma(A)}$ and $\overline{\sigma}(\overline{A}) =_{\rho_\lambda} \overline{\sigma(A)}$. This result is important since $\mathbb{T}_\lambda$ is the matching theory of the $\rho Cal_{\P}$ and thus, the $\mapsto\!\!\!\!\!\rightarrow_{\rho_\lambda}$ reductions are considered when testing whether a substitution is the solution of a given matching problem.

The $\mathbb{T}_\lambda$ matching theory is effectively used when the *CRS*-metaterm A contains metavariables of arity different from zero. In that case the application of the assignment involves $\underline{\beta}$-reduction steps performed at the meta-level of the *CRS*-reduction and these steps correspond to explicit ρ-reductions.

Using the above lemmas we can show that starting from a *CRS*-reduction a corresponding reduction in $\rho Cal_{\P}$ can be obtained. For this, a ρ-term encoding the *CRS*-derivation trace is constructed. When only a one-step *CRS*-reduction is considered, *i.e.* only one *CRS*-rule is applied, the corresponding ρ-term depends on the initial term to be reduced and on the applied rewrite rule.

Furthermore, we show that every possible ρ-derivation resulting from the correct initial ρ-term terminates and converges to the correct result.

Theorem 4.7 *Let t_0, t_1 be two* CRS*-terms, $L \to R$ a* CRS*-rule and σ an assignment such that $t_{0\lceil \sigma(L) \rceil_\omega}$ and $t_1 \equiv t_{0\lceil \sigma(R) \rceil_\omega}$. Given the ρ-term $u_0 \triangleq \overline{t_0}_{\lceil x \rceil_{tr(\omega,t_0)}} \twoheadrightarrow \overline{t_0}_{\lceil \overline{L \to R} \ x \rceil_{tr(\omega,t_0)}}$, then, every derivation resulting from $u_0 \ \overline{t_0}$ terminates and converges to $\overline{t_1}$:*

$$(u_0 \ \overline{t_0}) \ \mapsto\!\!\!\!\!\rightarrow_{\rho\delta_{\P}} \ \overline{t_1}$$

Proof. Thanks to the form of the ρ-term u_0 that permits to apply the rewrite rule exactly at the needed position, the ρ-reduction follows relatively easily. For the sake of readability we only show the case $\omega = \epsilon$.

$$(u_0 \ \overline{t_0}) \triangleq (x \twoheadrightarrow ((\overline{L \to R}) \ x)) \ \overline{t_0} \triangleq (x \twoheadrightarrow ((\overline{L \to R}) \ x)) \ \overline{\sigma(L)}$$
$$\triangleq (x \twoheadrightarrow ((\overline{L} \twoheadrightarrow \overline{R}) \ x)) \ \overline{\sigma(L)} \text{ (By Definition 4.1)}$$

$\mapsto_\rho [x \ll \overline{\sigma(L)}].((\overline{L} \twoheadrightarrow \overline{R})\ x)$

$\mapsto_\sigma (\overline{L} \twoheadrightarrow \overline{R})\ \overline{\sigma(L)}$

$\mapsto_\rho [\overline{L} \ll \overline{\sigma(L)}].\overline{R}$

$= [\overline{L} \ll \overline{\sigma}(\overline{L})].\overline{R}$ (By Lemma 4.6)

$\mapsto_\sigma \overline{\sigma}(\overline{R})$

$\twoheadmapsto_{\rho\sigma} \overline{\sigma(R)}$ (By Lemma 4.6)

$\triangleq \overline{t_1}$

We use a little abuse of notation when stating that $[\overline{L} \ll \overline{\sigma(L)}].\overline{R} = [\overline{L} \ll \overline{\sigma}(\overline{L})].\overline{R}$ due to the fact that, by Lemma 4.6, the two corresponding matching problems $\overline{L} \prec\!\!\prec_{\P} \overline{\sigma(L)}$ and $\overline{L} \prec\!\!\prec_{\P} \overline{\sigma}(\overline{L})$ have the same solution. The reduction for $\omega \neq \epsilon$ is similar, the only difference being at the matching level. In this case we also use Lemma 4.5 when matching against the translation of the left-hand side of the rule.

Another possible reduction is the following one:

$(u_0\ \overline{t_0}) \triangleq (x \twoheadrightarrow ((\overline{L \to R})\ x))\ \overline{t_0} \triangleq (x \twoheadrightarrow ((\overline{L \to R})\ x))\ \overline{\sigma(L)}$

$\triangleq (x \twoheadrightarrow ((\overline{L} \twoheadrightarrow \overline{R})\ x))\ \overline{\sigma(L)}$ (By Definition 4.1)

$\mapsto_\rho (x \twoheadrightarrow [\overline{L} \ll x].\overline{R})\ \overline{\sigma(L)}$

$\mapsto_\rho [x \ll \overline{\sigma(L)}].([\overline{L} \ll x].\overline{R})$

$\mapsto_\sigma [\overline{L} \ll \overline{\sigma(L)}].\overline{R}$

$= [\overline{L} \ll \overline{\sigma}(\overline{L})].\overline{R}$ (By Lemma 4.6)

$\mapsto_\sigma \overline{\sigma}(\overline{R})$

$\twoheadmapsto_{\rho\sigma} \overline{\sigma(R)}$ (By Lemma 4.6)

$\triangleq \overline{t_1}$

These are the only possible derivations since the translations of *CRS*-terms contain no redexes and the matching problem $\overline{L} \prec\!\!\prec_{\P} x$ in the latter derivation has no solution since L cannot be a variable. □

We can notice that we have a longer derivation scheme in the $\rho Cal_{\P}$ than in the *CRS*. For every rewrite step in the *CRS*, in the $\rho Cal_{\P}$ we have two (ρ)-rule steps plus two (σ)-rule steps for the application of the rewrite rule and some additional steps corresponding to the β-reduction steps performed at the meta-level of the *CRS*-reduction. These latter steps are performed at the object-level of the $\rho Cal_{\P}$ and their number depends on the arity of the *CRS*-metavariables in the right-side of the considered rewrite rule. We should point out that the matching performed at the meta-level of the $\rho Cal_{\P}$ may involve some derivations but this time performed in ρCal_{λ}.

We can generalize the theorem above and built, using the derivations of a

term t_0 in a *CRS*, a ρ-term with a reduction similar to the one of t_0 in the *CRS*.

Lemma 4.8 *Let t_0, t_n be two* CRS*-terms such that $t_0 \mapsto\!\!\!\!\to_{\mathcal{R}} t_n$ and $u_0, \ldots, u_n$ the corresponding derivation trace terms in the ρCal. Then every derivation resulting from $(u_n \ \ldots \ (u_0 \ \overline{t_0}))$ terminates and converges to $\overline{t_n}$.*

Proof. By induction on the number of derivation trace terms $u_0, \ldots, u_n$. □

We can state thus that we have a complete and correct translation of *CRS*-reductions to ρ-reductions. Given the simplicity of the translation the properties of the rewrite system, like for example the orthogonality, are preserved. As far as it concerns the corresponding *CRS*-reductions, properties like the termination and the confluence are also preserved due to the direct correspondence with the ρ-reductions.

The main difference between the two systems lays in the fact that rewrite rules and consequently their control (application position) are defined at the object-level of the ρCal while in the *CRS* the reduction strategy is left implicit. The possibility to control the application of rewrite rules is particularly useful when the rewrite system is not confluent or terminant. Moreover, while in the *CRS* the β-reduction is implicitly included in the application of the assignment, in the $\rho Cal_{\P}$ the corresponding reductions are performed explicitly.

The ρ-terms u_i can be built automatically starting from the *CRS*-reduction steps as stated in Theorem 4.7. It is obviously interesting to give a method for constructing this terms without knowing *a priori* the derivation from t_0 to t_n but only the set of rewrite rules to be applied. This needs the definition of iteration strategies and of strategies for the generic traversal of terms. This has been done for the initial version of the ρCal either by enriching the calculus with a new operator [6] or by adding an "exception handling mechanism" to the calculus [10]. We conjecture that these approaches that have already been used for encoding first order rewriting can be used for the *CRS* translation as well.

More recently, we have been working on a typed version of the ρCal which allows the definition of iterators and, as a consequence, allows one to represent reductions in first order rewriting. The use of this method for representing *CRS* reductions will be the object of a later study.

Example 4.9 Given the *CRS*-reduction $f(App(Ab([x]f(x)), a)) \mapsto\!\!\!\!\to_{\mathcal{R}} f(f(a))$ presented in Example 3.4. In order to obtain a similar reduction in the $\rho Cal_{\P}$, we consider the ρ-term $f(y) \twoheadrightarrow f(\overline{BetaCRS}\ y)$ (with $BetaCRS$ defined in Example 3.2) and we apply it to the translation of the initial *CRS*-term:

$$
\begin{aligned}
&(f(y) \twoheadrightarrow f(\overline{BetaCRS}\ y))\ (\overline{f(App(Ab([x]f(x)), a))}) \\
&\mapsto_{\rho}\ [f(y) \ll f(App(Ab(x \twoheadrightarrow f(x)), a))].f(\overline{BetaCRS}\ y) \\
&\mapsto_{\sigma}\ f(\overline{BetaCRS}\ App(Ab(x \twoheadrightarrow f(x)), a))
\end{aligned}
$$

$$
\begin{aligned}
&\equiv f((App(Ab(x \rightarrowtriangle (Z\ x)), Z_1) \rightarrowtriangle Z\ Z_1)\ App(Ab(x \rightarrowtriangle f(x)), a)) \\
&\mapsto_\rho\ f([App(Ab(x \rightarrowtriangle (Z\ x)), Z_1) \ll App(Ab(x \rightarrowtriangle f(x)), a)].Z\ Z_1) \\
&\mapsto_\sigma\ f(Z\{Z/z \rightarrowtriangle f(z)\}\ Z_1\{Z_1/a\}) \\
&\qquad = f((z \rightarrowtriangle f(z))\ a) \\
&\mapsto_\rho f([z \ll a].f(z)) \\
&\mapsto_\sigma f(f(a))
\end{aligned}
$$

One can notice in Example 4.9 that the reductions in *CRS* and in rewriting calculus lead to the same final term, modulo the translation, but we do not have an one-to-one correspondence between the rewrite steps (the steps from $f((z \rightarrowtriangle f(z))\ a)$ to $f(f(a))$ are explicit only in the ρCal). The same behavior is obtained in the following (more complicated) example.

Example 4.10 [λ-calculus with surjective pairing]

Given a *CRS* with the following set of rewrite rules $\mathcal{R}$:

$$
\begin{aligned}
\mathcal{R} = \{&P_0 : \Pi_0(\Pi(X_1, X_2)) \to X_1, \\
&P_1 : \Pi_1(\Pi(X_1, X_2)) \to X_2, \\
&P : \Pi(\Pi_0 X_1, \Pi_1 X_1) \to X_1, \\
&BetaCRS\}
\end{aligned}
$$

where $X_1, X_2 \in \mathcal{Z}_0$, $\Pi \in \mathcal{F}_2$ (pair function), $\Pi_0, \Pi_1 \in \mathcal{F}_1$ (projections) and $BetaCRS$ as in Example 3.2.

We consider the *CRS*-term $t = App(Ab([z]\Pi(\Pi_1 z, \Pi_0 z)), \Pi(x_1, x_2))$ consisting in the application of the function $Ab([z]\Pi(\Pi_1 z, \Pi_0 z))$, that swaps the elements of a pair, to the pair $\Pi(x_1, x_2)$.

To reduce the *CRS*-term t we first apply the rule $BetaCRS$ with the assignment $\sigma = \{(Z, \underline{\lambda} z.\Pi(\Pi_1 z, \Pi_0 z), (Z_1, \Pi(x_1, x_2))\}$ and we obtain:

$$
\begin{aligned}
\sigma(Z(Z_1)) &= (\sigma(Z))(\sigma(Z_1)) \\
&= (\underline{\lambda} z.\Pi(\Pi_1 z, \Pi_0 z))(\Pi(x_1, x_2)) \downarrow_{\underline{\beta}} \\
&= \Pi(\Pi_1(\Pi(x_1, x_2)), \Pi_0(\Pi(x_1, x_2)))
\end{aligned}
$$

Next we apply the rules P_1 and P_0 to the first and second argument of Π respectively, using the assignment $\sigma' = \{(X_1, x_1), (X_2, x_2)\}$ and we obtain the final result:

$$\Pi(\Pi_1(\Pi(x_1, x_2)), \Pi_0(\Pi(x_1, x_2))) \mapsto_{P_1} \Pi(x_2, \Pi_0(\Pi(x_1, x_2))) \mapsto_{P_0} \Pi(x_2, x_1)$$

We translate now the example into the $\rho Cal_{\P}$. We translate the set of *CRS*

rewrite rules

$$\begin{aligned}
\overline{P_0} &\triangleq \Pi_0(\Pi(X_1, X_2)) \twoheadrightarrow X_1 \\
\overline{P_1} &\triangleq \Pi_1(\Pi(X_1, X_2)) \twoheadrightarrow X_2 \\
\overline{P} &\triangleq \Pi(\Pi_0 X_1, \Pi_1 X_1) \twoheadrightarrow X_1 \\
\overline{BetaCRS} &\triangleq App(Ab(x \twoheadrightarrow (Z\ x)))\ Z_1) \twoheadrightarrow Z\ Z_1
\end{aligned}$$

and the *CRS*-term t:

$$\overline{t} \triangleq App(Ab(z \twoheadrightarrow (\Pi(\Pi_1 z, \Pi_0 z))), \Pi(x_1, x_2))$$

Starting from the *CRS* reduction above, we build the following ρ-terms corresponding to the application of the $\overline{BetaCRS}$ and $\overline{P_0}, \overline{P_1}$ rules respectively

$$u_1 = \overline{BetaCRS}\ and\ u_2 = (\Pi(x, y) \twoheadrightarrow \Pi(\overline{P_1}\ x, \overline{P_0}\ y))$$

and we build the ρ-term: $u_2\ (u_1\ \overline{t})$. In a more automatic approach we should have built three terms corresponding to the three *CRS* reduction steps.

First of all, we perform the $\overline{BetaCRS}$ reduction step using the substitution $\overline{\sigma} = \{Z/\ z \twoheadrightarrow \Pi(\Pi_1 z, \Pi_0 z),\ Z_1/\ \Pi(x_1, x_2)\}$ and we obtain

$$\begin{aligned}
\overline{\sigma}(Z\ Z_1) &= \overline{\sigma}(Z)\ \overline{\sigma}(Z_1) \\
&= (z \twoheadrightarrow \Pi(\Pi_1 z, \Pi_0 z))\ \Pi(x_1, x_2) \\
&\mapsto_\rho [z \lessdot \Pi(x_1, x_2)].\Pi(\Pi_1 z, \Pi_0 z) \\
&\mapsto_\sigma \Pi(\Pi_1(\Pi(x_1, x_2)), \Pi_0(\Pi(x_1, x_2)))
\end{aligned}$$

Next, we continue reducing the ρ-term obtained as intermediary result:

$$\begin{aligned}
&(\Pi(x, y) \twoheadrightarrow \Pi(\overline{P_1}\ x, \overline{P_0}\ y))\ \Pi(\Pi_1(\Pi(x_1, x_2)), \Pi_0(\Pi(x_1, x_2))) \\
&\mapsto_\rho [\Pi(x, y) \ll \Pi(\Pi_1(\Pi(x_1, x_2)), \Pi_0(\Pi(x_1, x_2)))].\Pi(\overline{P_1}\ x, \overline{P_1}\ y) \\
&\mapsto_\sigma \Pi(\overline{P_1}\ \Pi(\Pi_1(\Pi(x_1, x_2)), \overline{P_0}\ \Pi_0(\Pi(x_1, x_2)))
\end{aligned}$$

The last reduction consists in applying the ρ-rules $\overline{P_0}$ and $\overline{P_1}$ using the substitution $\overline{\sigma'} = \{(X_1, x_1), (X_2, x_2)\}$.

$$\Pi(\overline{P_1}\ \Pi(\Pi_1(\Pi(x_1, x_2)), \overline{P_0}\ \Pi_0(\Pi(x_1, x_2))) \mapsto\!\!\!\!\!\!\rightarrow_{\rho\sigma\P} \Pi(x_2, x_1)$$

5 Conclusions

The applications of the rewriting calculus are various and numerous. The rewriting calculus is a sufficiently powerful framework allowing one to represent the usual computational formalisms. It contains the complementary properties of first-order rewriting and lambda calculus. Moreover, it permits the description of rewrite based languages and of the object oriented calculi in a natural and simple way. We have shown in this paper that also higher order

rewriting can be represented in the ρCal and in particular we have analyzed the relation with the *Combinatory Reduction systems*. Any reduction of a term *w.r.t.* a given *CRS* can be represented by a corresponding ρ-term.

This ρ-term can be built automatically starting from the *CRS*-reduction steps. We conjecture that the different approaches used for the representation of first-order rewriting in the ρCal can be applied here for the construction of an appropriate term only from the set of *CRS*-rules and without without any knowledge on the reduction steps. In fact, the use of a recent definition of iterators in ρCal for the representation of reduction strategies like innermost and in particular for the representation of *CRS*-reductions is a work in progress.

The results of the comparison indicates a certain gap between the two formalisms. "Walking through the context" is done implicitly in the *CRS*, while additional ρ-terms need to be inserted to direct the reduction in the ρCal. Rewrite rules are defined at the object level of the ρCal and they are applied explicitly. The reduction is then performed by the three evaluation rules of the ρCal. On the contrary, in the *CRS* we have a set of rewrite rules that is particular to the *CRS* considered and the strategy of application is left implicit. Moreover, the evaluation of an assignment is done at the meta-level of the *CRS* using meta λ-calculus, while in the ρCal the application of a substitution leads to additional explicit reduction steps. For this reason, we generally have a longer reduction scheme in the ρCal than in the *CRS*.

Since we are mainly interested in the expressive power of the ρCal we have proposed in this paper a translation from *CRS* to ρCal but the translation the other way round has not been explicitly defined here. We believe this translation is possible but maybe not as obvious as one may think since the explicit control of rewrite rules in the ρCal should be somehow simulated in the corresponding *CRS* rewrite system.

We have considered in this paper *CRS* satisfying the pattern condition. However, we believe that the results obtained can be applied also to general *CRS* and a similar correspondence between *CRS*-reductions and reductions in an appropriate version of ρCal can be defined similarly.

Other higher order rewrite systems have already been compared, for example the *CRS* and the *Higher-order Rewrite Systems* (*HRS*) [17,18]. The detailed comparison can be found in [21] and reveals that the two systems have the same expressive power and therefore they can be considered equivalent. Using this comparison, we can have an indirect representation of the *HRS* in the $\rho Cal_{\P}$ composing the translation from the *HRS* to the *CRS* as defined in [21] with the translation from *CRS* to ρCal we have defined in this paper. Some other higher order systems like the Expression Reduction Systems of Khasidashvili [12] (*ERS*) should be considered. Although *CRS* and *ERS* are conceptually very similar, their syntax differs in many aspects (for example the restriction in the *ERS* to *admissibles* assignments). Therefore, it may be interesting to analyze also the correspondence between the ρCal and

the *ERS* or other systems like the Explicit Reduction Systems of Pagano [20].

Acknowledgements.

The authors wish to thank Luigi Liquori for fruitful discussions and comments. We would also like to thank Femke van Raamsdonk for providing an important list of CRS examples.

References

[1] F. Barbanera, M. Fernández, and H. Geuvers. Modularity of Strong Normalisation and Confluence in the Algebraic λ-Cube. *Journal of Functional Programming*, 7(6):613–660, November 1997.

[2] H. Barendregt. *Lambda Calculus: its Syntax and Semantics*. North Holland, 1984.

[3] Gilles Barthe, Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. Pure Patterns Type Systems. In *Principles of Programming Languages - POPL2003, New Orleans, USA*. ACM, January 2003.

[4] F. Blanqui. *Type Theory and Rewriting*. PhD thesis, University Paris-Sud, 2001.

[5] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1941.

[6] H. Cirstea and C. Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, 2001.

[7] H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In *Proc. of RTA*, volume 2051 of *LNCS*, pages 77–92. Springer-Verlag, 2001.

[8] H. Cirstea, C. Kirchner, and L. Liquori. The Rho Cube. In *Proc. of FOSSACS*, volume 2030 of *LNCS*, pages 166–180, 2001.

[9] H. Cirstea, C. Kirchner, and L. Liquori. Rewriting calculus with(out) types. In Fabio Gadducci and Ugo Montanari, editors, *Proceedings of the fourth workshop on rewriting logic and applications*, Pisa (Italy), September 2002. Electronic Notes in Theoretical Computer Science.

[10] G. Faure and C. Kirchner. Exceptions in the Rewriting Calculus. In *Proc. of RTA*, volume 2378 of *LNCS*, pages 66–82. Springer-Verlag, 2002.

[11] J.P. Jouannaud and M. Okada. Abstract Data Type Systems. *Theoretical Computer Science*, 173(2):349–391, 1997.

[12] Z.O. Khasidashvili. Expression reduction systems. In *Proceedings of I. Vekua Institute of Applied Mathematics*, volume 36, pages 200–220, 1990.

[13] Z.O. Khasidashvili. Church-rosser theorem in orthogonal reduction systems. Technical report, INRIA Rocquencourt, 1992.

[14] J. W. Klop. *Combinatory Reduction Systems*. PhD thesis, CWI, 1980.

[15] J.W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory Reduction Systems: Introduction and Survey. *Theoretical Computer Science*, 121:279–308, 1993. Special issue in honour of Corrado Böhm.

[16] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In *Proc. of ELP*, volume 475 of *LNCS*, pages 253–281. Springer-Verlag, 1991.

[17] T. Nipkow. Higher-order critical pairs. In *Proceedings of Logic in Computer Science*, pages 342–349, 1991.

[18] T. Nipkow. Orthogonal higher-order rewrite systems are confluent. In *Proceedings of the International Conference on Typed La mbda Calculi and Applications*, pages 306–317, 1993.

[19] T. Nipkow and C. Prehofer. Higher-Order Rewriting and Equational Reasoning. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications. Volume I: Foundations*. Kluwer, 1998.

[20] B. Pagano. *Des calculs de substitution explicites et de leur application à la compilation des langages fonctionnels*. PhD thesis, U. Paris VI, 1997.

[21] V. van Oostrom and F. van Raamsdonk. Comparing combinatory reduction systems and higher-order rewrite systems. Report CS-R9361, CWI, 1993.

[22] D. A. Wolfram. *The Clausal Theory of Types*, volume 21 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.

Deductive Generation of Constraint Propagation Rules

Sebastian Brand

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Eric Monfroy

IRIN
Université de Nantes
2, rue de la Houssinière
BP 92208 Nantes Cedex 03
France

Abstract

Constraint propagation can often be conveniently expressed by rules. In recent years, a number of techniques for automatic generation of rule-based constraint solvers have been developed, most of them using a generate-and-test approach. We present a generation method that is based on deduction. A solver for a complex constraint is obtained from one or several weaker solvers for simple constraints. We describe deductive, incremental constructions of constraint solvers for several types of constraint modifications, among them conjunction and existential and universal quantification.

1 Introduction

Rule-based methodologies have had a firm place in constraint processing for a longer time. By now, a number of languages allow one to program constraint solvers in this declarative way. Notable examples are the languages based on the `cc` paradigm [14] such as [6] and especially `CHR` [10], but also the term-rewriting centred ELAN [11], applied for instance in [8], and CLAIRE [7] are suitable for rule-based constraint processing, even if not exclusively aimed at it. The last years have seen an increase in work on, and interesting results in, the automatic generation of rule-based constraint solvers [3,2]. Almost all of these approaches have in common that their paradigm is essentially

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

generate-and-test. Syntactically correct rule candidates are enumerated and subjected to a validity test against the constraint definition, which is usually extensive – the truth table –, or a constraint logic program. The exception is [13] who derive the conclusion from the given condition, which itself comes from a syntactic enumeration process, however. [1] creates new `CHR` solvers by merging given ones. The main focus there, however, is on termination and confluence, while we look only at solvers with propagation rules where these two properties are of no concern.

We explore here the idea of incremental, deductive rule generation. The key property of this way of generating rules is that the *source definition* of the constraint is irrelevant. Instead, the input to our method is another set of rules associated to some constraints, and the relation of these constraints to the desired constraint. The rules are processed according to declarative transformation steps, *meta rules*, leading to introduction of new or discarding of currently present rules. In that way, solvers for basic, primitive constraints, are transformed or combined into solvers for complex constraints. This means incremental construction of solvers, and reuse in rule generation. Deducing solvers can be substantially faster then generation from scratch. The resulting solvers are stronger than the simple union of their building blocks. The strength of constraint propagation increases, and the enforced local consistencies become stricter.

Although we introduce the deductive method in a general setting in Section 2, the main part of the paper deals with a specific type of rule: the inclusion/membership rules, that were first proposed in [3]. These rules are interesting and relevant for constraint programming as they can be used to enforce generalised arc-consistency (GAC), an important local consistency notion.

We discuss a number of transformations on inclusion rule sets, leading to solvers for modified constraints. The most important one is rule generation for the conjunctive constraint $C_1 \wedge C_2$, based on the rules for C_1 and C_2. An auxiliary modification is increasing the arity of a constraint: constraint padding. The inverse effect is achieved by existentially or universally quantifying a variable in a constraint. Furthermore, we discuss tightening a constraint by disallowing previous solutions, constraint restriction. An important special case of the latter is defining a constraint negatively by the set of its non-solutions. All these transformations yield a rule-based solver that enforces generalised arc-consistency on its associated constraint, when the source solver satisfies certain conditions.

We conclude in Section 6 with the example of incrementally constructing a solver for the `fulladder` constraint, based only on solvers for some primitive base constraints.

1.1 Preliminaries: Constraint Satisfaction Problems

Consider a sequence of variables $X = x_1, \ldots, x_n$ with respective domains $D_1, \ldots, D_n$ associated to them. By a constraint C on the variables X we mean a subset of $D_1 \times \ldots \times D_n$. Given an element $d = d_1, \ldots, d_n$ of $D_1 \times \ldots \times D_n$ and a subsequence $Y = x_{i_1}, \ldots, x_{i_\ell}$ of X we denote by $d[Y]$ the sequence $d_{i_1}, \ldots, d_{i_\ell}$. In particular, for a variable x_i from X, $d[x_i]$ denotes d_i. A constraint satisfaction problem, in short CSP, consists of a finite sequence of variables $X = x_1, \ldots, x_n$ with respective domains $\mathcal{D} = D_1, \ldots, D_n$, together with a finite set $\mathcal{C}$ of constraints, each on a subsequence of X. We write it as $\langle \mathcal{C}; x_1 \in D_1, \ldots, x_n \in D_n \rangle$, or short $\langle \mathcal{C}; X \in \mathcal{D} \rangle$. By a solution to $\langle \mathcal{C}; X \in \mathcal{D} \rangle$ we mean an element $d \in \mathcal{D}$ such that for each constraint $C \in \mathcal{C}$ on the variables Y we have $d[Y] \in C$. The latter is also be denoted by $\models_d C$. By extension we write $\models_d \mathcal{C}$ if $\models_d C$ for all $C \in \mathcal{C}$. Then d is a solution if $\models_d \mathcal{C}$.

1.2 Local Consistency and Propagation

A technique that is central in solving constraint satisfaction problems is constraint propagation. It is most common in its specific form of domain reduction: the domain of a variable participating in a constraint is reduced in the light of the domains of the other variables. As an example consider $\langle x < y \,;\, x \in \{1..3\}, y \in \{1..3\} \rangle$ in which the inequality constraint propagates to the smaller domains $x \in \{1..2\}, x \in \{2..3\}$. Constraint propagation is best understood as fixpoint computation of propagation operators on CSPs, and we are concerned with propagation operators that are rules. A CSP that is a fixpoint of a set of rules, for instance after exhaustive application of these rules, is said to be closed under this rule set.

A measure of the strength of propagation is given by the enforced local consistency (as opposed to global consistency that is typically computationally too costly to enforce). An important local consistency notion is generalised arc-consistency (GAC). It holds for a satisfiable constraint if every instantiation of any variable of the constraint can be extended to an instantiation of all variables that satisfies the constraint, and is permitted by the current domains.

Generally, propagation is incomplete in the sense that it does not identify a solution or inconsistency of a CSP. It is usually interleaved with search.

2 General Propagation Rules.

General constraint propagation rules have the form

$$\{c_1, \ldots, c_n\} \rightarrow \{c'_1, \ldots, c'_m\}$$

where both left-hand side (*lhs*) and right-hand side (*rhs*) are sets of constraints interpreted as conjunctions (allowing disjunctive *rhs*'s would amount to

search). Propagation rules operate on a constraint store that is itself a set of constraints. A rule is applicable if its *lhs* is contained in, or implied by, the store. In this case its *rhs* is added to the store.

2.1 Validity

A substantial property of a rule is that it is valid with respect to its constraints: the *lhs* must imply the *rhs*. We presume that the *rhs* does not contain new variables, i. e., it simply constrains more some of the *lhs* variables. In this case validity means that every solution of the *lhs* can be projected into one of the *rhs*:

$$C \to C' \qquad \text{is valid if} \qquad \forall d.\ \models_d C \text{ implies } \models_d C' \ .$$

2.2 Feasibility

It is useful to classify rules by the satisfiability of their *lhs*'s. We call a rule feasible, if its *lhs* is satisfiable, that is

$$C \to C' \qquad \text{is feasible if} \qquad \exists d.\ \models_d C \ .$$

An infeasible rule is thus trivially valid.

2.3 Redundancy

When considering a set of constraint propagation rules, it is useful to ask whether it is a smallest such set to enforce the associated local consistency notion. We define: a rule r is redundant with respect to a rule set $\mathcal{R}$ if every set of constraints closed under $\mathcal{R}$ is also closed under r. In this case r does not contain any extra information. It is helpful that, when testing a rule for being redundant, it suffices to test its *rhs* as the set of constraints [5].

If a rule set is strong enough to detect inconsistency of sets of constraints in the language of their *lhs*'s, then infeasible rules are redundant.

3 Transformations of (General) Propagation Rules

A transformation is a sequence of atomic steps introducing or removing single propagation rules. There are also two auxiliary structural transformation steps. These deal with the *rhs* of rules, while the *lhs*'s are focused on by the transformation steps that introduce or remove a rule. We assume a fixed, finite language of constraints. For brevity, we write $\mathcal{R}, r$ for the set $\mathcal{R} \cup \{r\}$ in the remainder of the paper.

3.1 Subsumption

A rule subsumes another with the same *rhs* if its *lhs* implies the other. As a meta rule we formulate more precisely

$$\text{[general-subsume]} \quad \frac{\mathcal{R},\ C_1 \to C',\ C_2 \to C'}{\mathcal{R},\ C_1 \to C'} \text{ if } \quad \forall d.\ \models_d C_2 \text{ implies } \models_d C_1$$

A typical corresponding situation is $C_1 \subseteq C_2$. We say that a rule is subsumed by a set of rules if it is subsumed by a rule contained in the set.

3.2 Derivation

Two ancestor rules with the same *rhs* give rise to a descendant if the constraint language can express the disjunction of the two *lhs*'s, or something stronger. Formally,

$$\text{[general-derive]} \quad \frac{\mathcal{R},\ C_1 \to C',\ C_2 \to C'}{\mathcal{R},\ C_1 \to C',\ C_2 \to C',\ C_3 \to C'}$$

$$\text{if} \qquad \forall d.\ \models_d C_3 \text{ implies } \models_d C_1 \text{ or } \models_d C_2$$

It is important to observe that validity is preserved – the descendant rule inherits it from its ancestors. The idea of this transformation step is to compose the first two rules, at best into one that subsumes both former rules.

To avoid trivial, subsumed descendants, C_3 should not simply imply one of C_1, C_2. Finding candidates for C_3 depends on the language. Ideally, it can be constructed from C_1, C_2, and we are able to do so in the following.

(De)composing rules
In the above meta rules, all mentioned propagation rules coincide in their *rhs*'s, while in general we deal with varying *rhs*'s. We assume suitable, implicit transformations between $C \to C'_1 \cup C'_2$ and the two rules $C \to C'_1,\ C \to C'_2$.

Having these two transformation steps, it is interesting to describe the result of a rule set derivation. For this, we proceed with a specific language.

4 Inclusion Rules

We consider rules of the form

$$C, x_1 \in S_1, \ldots, x_n \in S_n \ \to\ y \neq a$$

where S is a set of constants, and a a constant. The constraint C is on the $n+1$ distinct variables $\{x_1, \ldots, x_n, y\}$. The base domain of all the variables is D, which means $C \subseteq D^{n+1}$. We require $\varnothing \neq S_i \subseteq D$ for all $i \in \{1..n\}$.

We call C the constraint *associated* to the rule. In the following, all rules are regarded as being associated to the same constraint, and we omit it from

the notation. Finally, with $X = (x_1, \ldots, x_n)$ and $S = S_1 \times \ldots \times S_n$ we abbreviate the above inclusion rule as $X \in S \to y \neq a$.

The interest in inclusion rules arises from the possibility to express, by sets of such rules, generalised arc-consistency (GAC) for the associated constraint.

4.1 Transformations of Inclusion Rules

We provide now specialisations of the general subsumption and derivation transformations. Subsequently, we describe the rule set resulting from a stabilising derivation of such transformations. If certain conditions on the source rule set are met then the resulting rule set enforces generalised arc-consistency.

4.2 Subsumption

This meta rule discards a rule in presence of a stronger one:

$$\text{[subsume]} \quad \frac{\mathcal{R},\ X \in S \to y \neq a,\ X \in P \to y \neq a}{\mathcal{R},\ X \in S \to y \neq a} \qquad \text{if } S \supseteq P$$

A rule subsumes another one with tighter bounds. It is easy to see that this is an instance of [general-subsume].

Example 4.1 $x \in \{2\} \to y \neq 1$ is subsumed by $x \in \{2, 3\} \to y \neq 1$. □

4.3 Derivation

This meta rule creates a new rule from two present rules:

$$\text{[derive]} \quad \frac{\mathcal{R},\ X \in S \to y \neq a,\ X \in P \to y \neq a}{\mathcal{R},\ X \in S \to y \neq a,\ X \in P \to y \neq a,\ X \in Q \to y \neq a}$$

$$\text{if} \quad \begin{array}{ll} \text{(i)} & Q_i = S_i \cap P_i \neq \varnothing \text{ at all indices } i \in \{1..n\} \text{ except for some } k \\ \text{(ii)} & Q_k = S_k \cup P_k \\ \text{(iii)} & Q_k \supset S_k \text{ and } Q_k \supset P_k \end{array}$$

The side conditions guarantee that the derived rule is syntactically correct (1.), and that it is not subsumed by any parent (3.), although it may itself subsume one or both of them. Furthermore, it is valid (1. and 2.). It is useful to notice that $x_k \in Q_k$ is the collapsed disjunctive constraint "$x_k \in S_k$ or $x_k \in P_k$".

A [derive] step depends on k, and for two ancestor rules there may be several appropriate indices k, satisfying (1.,2.,3.). Note however, that no derived rule subsumes another with a different k.

Example 4.2 From

$$x_1 \in \{1,2\}, x_2 \in \{1,3\} \to y \neq 2$$
$$x_1 \in \{2,3\}, x_2 \in \{2,3\} \to y \neq 2$$

we derive the two rules

$$x_1 \in \{1,2,3\}, x_2 \in \{3\} \to y \neq 2 \qquad \text{with } k = 1$$
$$x_1 \in \{2\}, x_2 \in \{1,2,3\} \to y \neq 2 \qquad \text{with } k = 2 \ .$$

□

4.4 Properties of the Transformations

We proceed by collecting properties of these inference rules. We do so in two steps: first, linking the source and the result of an exhaustive application of the transformations, and second, characterising the propagation that a fully transformed rule set achieves.

Atomic Rule, Rule Set Closure

It is convenient to have the concept of an atomic rule: $X \in S \to y \neq a$ is atomic if each S_i is a singleton set. It is useful to be aware of the 1-1 correspondence between such an atomic rule and a non-solution d of the associated constraint, namely by $\{d[X]\} = S$ and $d[y] = a$.

We denote by *closure*($\mathcal{R}$) the rule set that results from an exhaustive application of [subsume, derive]. The following first finding links atomic rules and rule set closure.

Theorem 4.3 *Let $\mathcal{R}$ be a set of inclusion rules and C their associated constraint. If $\mathcal{R}$ subsumes every* atomic *rule valid for C then closure($\mathcal{R}$) subsumes every rule valid for C.*

Proof. We argue by contradiction: Let us say that $r = (X \in S \to y \neq a)$ is valid but not subsumed by *closure*($\mathcal{R}$). Without loss of generality we assume that all other rules $X \in S' \to y \neq a$ with $S' \subset S$ are subsumed.

Observe first that r is not atomic. Take then some S_k that is not a singleton, and partition it into $S_k = P_k \cup Q_k$ where neither P_k nor Q_k is empty. We construct complete bounds P, Q by defining $P_i = Q_i = S_i$ at the remaining indices $i \neq k$.

Both corresponding rules $X \in P \to y \neq a$ and $X \in Q \to y \neq a$ are valid since r is. For each of the two rules there must be a subsuming rule contained in *closure*($\mathcal{R}$), as we assumed initially. Enter now these two subsuming rules into [derive]. The resulting new rule must subsume r, contradicting our assumption.

As regards [subsume], we remark that subsumption is a transitive relation. Therefore, if a rule is subsumed by a rule set then this is still the case after an application of [subsume] to the set. □

We know now which "seed rules" are necessary so that after closure there are rules for all valid propagations. Next, we establish the local consistency notion achieved by these propagations.

Theorem 4.4 *Let $\mathcal{R}$ be a set of inclusion rules valid for their associated constraint C. Let $\mathcal{R}$ subsume every rule valid for C. Then the constraint C is closed under $\mathcal{R}$ iff C is generalised arc-consistent.*

Proof. Suppose that C is not closed under $r = (X \in S \to y \neq a)$ in $\mathcal{R}$, thus $C[X] \subseteq S$ and $a \in C[y]$. Since r is valid we know that for all d we have that $d[X] \in S$ implies $d[y] \neq a$. The counter position is that, for all d, $d[y] = a$ implies $d[X] \notin S$, and in turn $d[X] \notin C[X]$. But this means that the partial instantiation $\{y \mapsto a\}$ can not be extended to a solution of C.

For the reverse direction, suppose that $\{y \mapsto a\}$ can not be extended to a solution of the constraint C. So no d exist with $d[y] = a$ and $d[X] \in C[X]$. Then $x_1 \in C[x_1], \ldots, x_n \in C[x_n] \to y \neq a$ is a valid rule; and as such is subsumed by $\mathcal{R}$. The subsuming rule in $\mathcal{R}$, however, is applicable to C. □

Both preceding results together allow us to obtain a GAC-enforcing set of inclusion rules from an initial set of all atomic rules, or corresponding subsuming rules, that is then closed in particular under [derive].

4.5 Uniqueness

The closure of a set $\mathcal{R}$ under the meta rules [derive,subsume] is unique if the base domain is finite. Then there are only finitely many syntactically correct rules. Any closure algorithm that applies [derive] at most once to any two rules in $\mathcal{R}$ for a specific k must terminate. Furthermore, the meta rule system is confluent. Every cricital pair, obtained by applying two meta rules to $\mathcal{R}$, is joinable. This is easy to see for pairs stemming from [derive]+[derive] and [subsume]+[subsume]. The somewhat more involved case of [derive]+[subsume] requires a case distinction (further applicability of [derive]) that we omit here.

4.6 Relation to RGA

The method of choice for producing sets of inclusion rules is described in [3], which introduced the notion. We compare the outcome of their Rule Generation Algorithm (called RGA here), and our closure method. The main difference lies in infeasible rules. Notice that the inclusion rule $X \in S \to y \neq a$ is infeasible exactly if $S \cap C[X] = \varnothing$.

Lemma 4.5 *Let $\mathcal{R}_{\mathrm{RGA}}$ be the inclusion rule set that RGA generates for a constraint C. Let $\mathcal{R}_{\mathrm{CLS}}$ be a set of rules valid for C and also subsuming every*

atomic rule valid for C. Moreover, let $\mathcal{R}_{\mathrm{CLS}}$ be closed under [derive, subsume]. Then

- *every rule in $\mathcal{R}_{\mathrm{RGA}}$ is subsumed by a rule in $\mathcal{R}_{\mathrm{CLS}}$, and*
- *every feasible rule in $\mathcal{R}_{\mathrm{CLS}}$ subsumes a rule in $\mathcal{R}_{\mathrm{RGA}}$.*

Proof. RGA (see its presentation in [3]) enumerates all valid rules, discarding those that are subsumed. In turn, $\mathcal{R}_{\mathrm{CLS}}$ subsumes all valid rules, by Theorems 4.3,4.4.

Inversely, each feasible rule in $\mathcal{R}_{\mathrm{CLS}}$ is valid, and not strictly subsumed by another valid rule. This means that it is either in, or subsumes a rule in, $\mathcal{R}_{\mathrm{RGA}}$. (The latter case my arise due to presence of infeasible rules and [derive].) □

Example 4.6 The deductive approach may yield infeasible rules, and rules that are "partially infeasible". Consider the constraint C_{ex} on x, y with domain $\{1, 2, 3\}$. RGA generates the rules $\mathcal{R} = \{(1), (2), (3), (4)\}$.

C_{ex}	x	y
	1	1
	3	1
	3	3

$$y \in \{3\} \rightarrow x \neq 1 \quad (1)$$
$$y \in \{1, 2, 3\} \rightarrow x \neq 2 \quad (2)$$
$$x \in \{1, 2, 3\} \rightarrow y \neq 2 \quad (3)$$
$$x \in \{1\} \rightarrow y \neq 3 \quad (4)$$

Let us construct $\mathcal{R}'$ by replacing (1) in $\mathcal{R}$ by the valid, feasible rule

$$y \in \{2, 3\} \rightarrow x \neq 1 \ . \quad (5)$$

Observe that $\mathcal{R}'$ is closed under [derive, subsume]. The *rhs* of rule (5) contains the unsatisfiable part $y \in \{2\}$. Next, construct $\mathcal{R}''$ by adding to $\mathcal{R}$

$$y \in \{2\} \rightarrow x \neq 1 \ . \quad (6)$$

Rule (6) is valid but infeasible. Also $\mathcal{R}''$ is closed under [derive, subsume]. □

4.7 Significance of Infeasibility

For constraint propagation, rule (5) is preferable over rule (1), which is strictly subsumed. Rule (6), on the other hand, is redundant relative to $\mathcal{R}$, and so undesirable. Ideally, infeasible rules should not be derived. However, many feasible rules that RGA produces, are redundant [5]. Therefore, a finalising redundancy removal after rule generation is a good idea in either approach.

5 Applications

We demonstrate some applications of deductive rule generation.

5.1 Conjunction of Constraints

Given two constraints C_1 and C_2 on the same variables, and their associated rules $\mathcal{R}(C_1)$ and $\mathcal{R}(C_2)$, we are interested in rules for the conjunctive constraint $C_1 \wedge C_2$ (note that the solutions of this constraint, hence the constraint itself, is the intersection $C_1 \cap C_2$). In general the simple union $\mathcal{R}(C_1) \cup \mathcal{R}(C_2)$ does not propagate as strongly as possible. For example, consider the boolean variables $x, y \in \{0, 1\}$ constrained by $not(x, y) \wedge (x = y)$. This CSP is closed under any rules valid for the two constraints individually, yet it is inconsistent.

By Theorems 4.3,4.4 we can state, however, that if $\mathcal{R}(C_1), \mathcal{R}(C_2)$ subsume all atomic rules valid for C_1, C_2, resp., then

$$\mathcal{R}(C_1 \wedge C_2) = \mathit{closure}(\mathcal{R}(C_1) \cup \mathcal{R}(C_2))$$

enforces GAC on the conjunctive constraint $C_1 \wedge C_2$. To see this, observe that any atomic rule valid for $C_1 \wedge C_2$ must be valid for at least one of C_1, C_2 as well. An obvious generalisation is $\mathcal{R}(\bigwedge_{i=1}^{m} C_i) = \mathit{closure}(\bigcup_{i=1}^{m} \mathcal{R}(C_i))$. For an example, we refer to Subsection 6.

5.2 Local Consistency Notion

It is interesting to observe that from the view of the set of the constituent constraints C_i, all on the same set of variables, the consistency enforced is relational $(1, m)$-consistency [9]. Since we enforce GAC on the global conjunctive constraint, an instantiation of any one variable can be extended to a solution of the global constraint, which is also a solution of each of the constituent constraints. Enforcing GAC on the constituent constraints separately is equivalent to relational $(1, 1)$-consistency, a strictly weaker local consistency.

5.3 Constraint Padding

In order to construct the rules for a conjunctive constraint as in the preceding subsection, the participating constraints must be on the same set of variables. This can be achieved by suitably extending the individual constraints to new variables, without constraining the latter. We call this padding. Consider a constraint $C \subseteq D^n$ and a variable $v \in D$ not constrained by C. We define $C'(X, v) = \{d \mid d[X] \in C \wedge d[v] \in D\}$, or concisely $C' = C \times D$, and find its

rules by:

$$\begin{aligned}\mathcal{R}(C') = \mathit{closure}(\ & \{X \in S, v \in D \to y \neq a \mid (X \in S \to y \neq a) \in \mathcal{R}(C)\} \\ & \cup \\ & \{X \in S, y \in \{a\} \to v \neq b \mid \\ & \qquad (X \in S \to y \neq a) \in \mathcal{R}(C) \wedge b \in D\}\ )\end{aligned}$$

The first set in the union pads the input rules by simply adding a redundant test. In that way, all valid atomic rules with *rhs*'s on the variables of C are constructed. This set is closed under [derive, subsume] if $\mathcal{R}(C)$ is. The second set creates the rules for the new variable. Since v is not actually constrained, there can be no valid rule with a *rhs* on v. Hence all rules in the second set must be infeasible. Also observe that *exactly* all valid, atomic, infeasible rules on v are subsumed by this set. Although infeasible rules are redundant (with respect to a GAC-enforcing set of inclusion rules), it is essential to incorporate them here if the generated rule set is to serve as the input to another meta rule closure, such as for a conjunctive constraint.

In conclusion we can state that $\mathcal{R}(C')$ subsumes all atomic rules that are valid for C', if this is true for $\mathcal{R}(C)$ and C. Moreover, $\mathcal{R}(C')$ achieves then GAC on C'. The pre-closure processing is linear in the size of the set $\mathcal{R}(C)$.

Example 5.1 We pad the boolean constraint $not(x, y)$ to $not(x, y, z)$ with $z \in \{0, 1\}$.

$$\begin{array}{ll} not(x,y): & \\ & not(x,y,z): \\ y \in \{0\} \to x \neq 0 & \\ & y \in \{0\}, z \in \{0,1\} \to x \neq 0 \\ y \in \{1\} \to x \neq 1 & \\ & y \in \{1\}, z \in \{0,1\} \to x \neq 1 \\ x \in \{0\} \to x \neq 0 & \\ & x \in \{0\}, z \in \{0,1\} \to x \neq 0 \\ x \in \{1\} \to x \neq 1 & \\ & x \in \{1\}, z \in \{0,1\} \to x \neq 1 \\ & \\ & x \in \{0\}, y \in \{0\} \to z \neq 0, z \neq 1 \\ & x \in \{1\}, y \in \{1\} \to z \neq 0, z \neq 1 \end{array}$$

5.4 Defining a Constraint by its Non-Solutions

A constraint for which rules should be generated can be defined *positively* by stating its solutions. This is the input of the RGA algorithm [3]. Sometimes, however, it may be more natural to define a constraint *negatively* by stating the tuples that are *not* solutions. Suppose we are given a set N of such

non-solutions, or no-goods, defining the constraint C. We can write this as $C = D^{n+1} - N$ if C is on $n+1$ variables (recall $C \subseteq D^{n+1}$).

It is simple to obtain the corresponding GAC-enforcing rules as we can straightforwardly construct all valid atomic rules. Recall that every atomic rule corresponds to a non-solution of the constraint, and vice versa. Consequently, we find the seed rule set of all valid atomic rules by

$$\mathcal{R}_N = \{x_1 \in \{t_1\}, \ldots, x_n \in \{t_n\} \rightarrow x_0 \neq t_0\} \mid (t_0, \ldots, t_n) \text{ is a permutation of } t \in N\}$$

For instance, take a binary constraint $C(x, y)$ of which only $(1, 2)$ is not a solution. Then we obtain $\mathcal{R}_{\{(1,2)\}} = \{x \in \{1\} \rightarrow y \neq 2,\ y \in \{2\} \rightarrow x \neq 1\}$.

The closure $\mathcal{R}(C) = \mathit{closure}(\mathcal{R}_N)$ achieves GAC by Theorems 4.3,4.4. Constructing $\mathcal{R}_N$ is linear in the size of N: it produces $|N| \cdot (n+1)$ atomic rules.

Example 5.2 Define C over $x, y, z \in \{1..10\}$ as the constraint that at least one number in the ordered sequence x, y, z is not prime. The 4 non-solutions $(2,3,5), (2,3,7), (2,5,7), (3,5,7)$ are easily found. The rules are in Table 4.

12 atomic rules:	8 rules after closure:
$x \in \{2\}, y \in \{3\} \rightarrow z \neq 5$	$x \in \{2,3\}, y \in \{5\} \rightarrow z \neq 7$
$x \in \{2\}, z \in \{5\} \rightarrow y \neq 3$	$x \in \{2\}, y \in \{3\} \rightarrow z \neq 5$
$y \in \{3\}, z \in \{5\} \rightarrow x \neq 2$	$x \in \{2\}, y \in \{3,5\} \rightarrow z \neq 7$
$\vdots$	$x \in \{2,3\}, z \in \{7\} \rightarrow y \neq 5$
$y \in \{5\}, z \in \{7\} \rightarrow x \neq 3$	$x \in \{2\}, z \in \{5,7\} \rightarrow y \neq 3$
	$y \in \{3\}, z \in \{5,7\} \rightarrow x \neq 2$
	$y \in \{5\}, z \in \{7\} \rightarrow x \neq 3$
	$y \in \{3,5\}, z \in \{7\} \rightarrow x \neq 2$

Table 4
Rules of Example 5.2

Rule generation from the negative definition is particularly appropriate when the non-solutions are few, as in this example. Our implementation of the closure method took much less than a second to find the rules. In contrast, the RGA algorithm supplied with the 996 solution tuples did not return within 30 minutes. □

5.5 Restricting a Constraint

Suppose a constraint is defined by restricting another reference constraint by discarding solutions. That is, $C'(X) = C(X) \wedge X \notin N$ where N collects the solutions of C that cease to be solutions of C'. Such a situation may occur in a dynamic setting, where propagation with a relaxation should take place before the actual constraint is learned. Assume thus that we know N and the rules $\mathcal{R}(C)$. We construct the seed rule set $\mathcal{R}_N$ precisely as in the preceding Subsection 5.5. Then we combine to

$$\mathcal{R}(C') = \mathit{closure}(\mathcal{R}(C) \cup \mathcal{R}_N)$$

The properties of $\mathcal{R}(C')$ depend on those of $\mathcal{R}(C)$, cf. Theorems 4.3,4.4.

Example 5.3 Suppose the boolean constraint or on $x, y, z \in \{0, 1\}$ is the base constraint for or', defined by $or'(x, y, z) = or(x, y, z) \wedge (x, y, z) \neq (1, 1, 1)$, or alternatively, $or' = or - \{(1, 1, 1)\}$. We find $\mathcal{R}(or') = \mathit{closure}(\mathcal{R}(or) \cup \mathcal{R}_N)$ where $\mathcal{R}_N$ consists of the three rules:
$x \in \{1\}, y \in \{1\} \rightarrow z \in \{1\}$;
$x \in \{1\}, z \in \{1\} \rightarrow y \in \{1\}$;
$y \in \{1\}, z \in \{1\} \rightarrow x \in \{1\}$. □

5.6 Universal Quantification

Assume that C constrains the variables Y and another variable $x \in D$. Consider the constraint $C' = \forall x.C$ on the variables Y. It has the solutions $\{t \mid \forall a \in D.\ \exists t' \in C.\ t'[x] = a \wedge t'[Y] = t\}$. We can derive rules for C' based on those for C by simply discarding all references to x:

$$\mathcal{R}(C') = \mathit{closure}(\{Z \in S \rightarrow y \neq a \mid (Z \in S, x \in S_x \rightarrow y \neq a) \in \mathcal{R}(C)\})$$

If $\mathcal{R}(C)$ contains or subsumes all atomic rules valid for C then the equivalent holds true for $\mathcal{R}(C')$ and C'. So again, using Theorems 4.3,4.4, we obtain a GAC-enforcing rule set.

We argue for this as follows. Take a rule of C and a non-solution d excluded by it. d is a non-solution of C' as well, since the partial solution $d[Y]$ does not allow x to be all-quantified, $d[x]$ being the counter example. This means that we can correctly transform the rules of C into rules of C' as above.

It remains to consider completeness, that is, whether *all* atomic rules valid for C' are subsumed. Take such a rule, $Z \in S \rightarrow y \neq a$. But then some rule $Z \in S, x \in S_x \rightarrow y \neq a$ must be subsumed by $\mathcal{R}(C)$. For, otherwise all d with $\{d[Z]\} = S, d[y] = a$ were solutions, meaning that x could be all-quantified.

Again, constructing the pre-closure rule set is linear in the size of $\mathcal{R}(C)$.

Automatic rule generation for quantified constraints has been identified as desirable by [4], where a number of boolean constraints and associated rules for arc-consistency are discussed, for mixed sequences of universal and

existential quantification. In the subsection following the example we deal with existential quantification.

Example 5.4 Consider the constraint *rcc8comp*, the composition constraint from the Region Connection Calculus [12]. $rcc8comp(R_{\mathsf{AB}}, R_{\mathsf{BC}}, R_{\mathsf{AC}})$ describes the possible spatial relations between three regions $\mathsf{A}, \mathsf{B}, \mathsf{C}$. The 8 relations are disjoint, meet, overlap, coveredby, covers, contains, inside, equal. Two examples for allowed tuples are (contains, inside, equal), (contains, inside, overlap), meaning that if region A contains region B and region B is inside (not touching the border of) region C then region A can be exactly equal to, or overlap with, region C.

The meaning of the universally quantified constraint $\forall R_{\mathsf{AC}}.rcc8comp(R_{\mathsf{AB}}, R_{\mathsf{BC}}, R_{\mathsf{AC}})$ is then exactly those pairs of relations between A/B, and B/C, such that *any* relation is possible between A/C. Only three such pairs of region relations are legal: $(R_{\mathsf{AB}}, R_{\mathsf{BC}}) \in \{(\mathsf{disjoint}, \mathsf{disjoint}), (\mathsf{inside}, \mathsf{inside}), (\mathsf{overlap}, \mathsf{overlap})\}$.

rcc8comp is defined by 193 solutions tuples, from which, via set complement to get the non-solutions, and closure, 912 GAC-enforcing rules are generated. For the all-quantified constraint one obtains 7 rules by the above procedure.

5.7 Existential Quantification

Existential quantification, or projection, is the dual to introduction of variables, Subsection 5.3. Assume that C constrains the variables Y and another variable $x \in D$. Consider the constraint $C' = \exists x.C$ on the variables Y that has the solutions $C' = \{d[Y] \mid d \in C\} = C[Y]$. Rule construction in this situation is different in the sense that it requires closure prior to modification of the rules:

$$\mathcal{R}(C') = \{Z \in S \to y \neq a \mid (Z \in S, x \in D \to y \neq a) \in closure(\mathcal{R}(C))\}$$

If $\mathcal{R}(C)$ subsumes all atomic rules valid for C, then so does $\mathcal{R}(C')$ for C', and $\mathcal{R}(C')$ enforces GAC on C'.

Consider $Z \in S, x \in D \to y \neq a$ from the set $closure(\mathcal{R}(C))$. It means that it is correct to conclude $y \neq a$ from $Z \in S$, independent of the value of x. Then, clearly, the rule $Z \in S \to y \neq a$ is valid for C'. Inversely, consider some atomic rule $Z \in S \to y \neq a$ valid for C'. It means that there does *not* exist any solution d of C with $\{d[Z]\} = S$ and $d[y] = a$. So the rule $Z \in S, x \in D \to y \neq a$ is valid for C, and must be subsumed by $closure\mathcal{R}(C)$.

Finally, notice that $\mathcal{R}(C')$ is closed under [subsume, derive], since any transformation possible in $\mathcal{R}(C')$ would have been possible in $\mathcal{R}(C)$ using the corresponding ancestor rules.

6 An Example

We implemented the closure method as a naive meta-rule fixpoint computation algorithm in the CLP system ECL^{i}PS^e. The program accepts expressions that describe the construction of inclusion rule sets according to the transformations in the preceding section. The output is a list of `CHR` rules.

FullAdder as a Composite Constraint

The basic rule set constructions of Section 5 enable us to derive a rule set for a composite constraint based only on the rules of its subconstraints, and without any reference to the extensional definition of any of the constraints. We demonstrate this with the example of the *fulladder* constraint. The *fulladder* gate adds two bits and a carry bit and produces the sum bit and output carry bit. It is often defined with the help of the primitive gates and, or, xor and three auxiliary variables:

$$\begin{aligned} fulladder(x,y,z,s,c) \quad \equiv \quad & \exists\, c_1, c_2, s_1.\ xor(x,y,s_1) \ \wedge \\ & and(x,y,c_1) \ \wedge \\ & and(z,s_1,c_2) \ \wedge \\ & or(c_1,c_2,c) \ \wedge \\ & xor(z,s_1,s) \end{aligned}$$

To indicate the individual transformations, we make the padding visible:

$$\begin{aligned} fulladder(x,y,z,s,c) \quad \equiv \quad & \exists\, c_1, c_2, s_1. \quad \exists\, h_1, \ldots, h_{25}. \\ & xor(x,y,h_1,s_1,h_2,h_3,h_4,h_5) \ \wedge \\ & and(x,y,h_6,h_7,c_1,h_8,h_9,h_{10}) \ \wedge \\ & and(h_{11},h_{12},z,s_1,h_{13},c_2,h_{14},h_{15}) \ \wedge \\ & or(h_{16},h_{17},h_{18},h_{19},c_1,c_2,c,h_{20}) \ \wedge \\ & xor(h_{21},h_{22},z,s_1,h_{23},h_{24},h_{25},s) \end{aligned}$$

The input are the three base rule sets of *xor*, *and*, *or*. These rule sets are padded so as to associate them to the same (auxiliary) variables. Then the rules corresponding to the conjunctive constraint are computed, and finally existential quantification removes the auxiliary variables. This process generates 66 rules that enforce GAC on the *fulladder* constraint.

It is useful to note that this transformation order requires the computation of only one rule set closure. Padding and joining the rules for the conjunctive constraint requires a post-processing closure, while existential quantification needs a pre-processing closure and yields a rule set that is closed. Recall that all transformation steps other than closure are just linear in the size of their input set.

The rules of *fulladder* propagate strictly stronger than the rules of its individual subconstraints. The CSP $\langle \mathit{fulladder}(x, y, z, s, c)\,;\, x, z, c \in \{0, 1\}, y \in \{1\}, s \in \{0\}\rangle$ is closed under the subconstraint rules, but a *fulladder* deduced rule propagates $c \in \{1\}$.

To compare with RGA, it is first necessary to create the entire *fulladder* extensional definition – the truth table – by a separate algorithm. The definition can then be given to RGA. It generates 52 rules. Since some of our 66 rules are infeasible while RGA does not produce such rules, we subjected both to a removal of redundant rules. This reduced both rule sets to a unique set of 28 nonredundant rules.

7 Conclusions

We made a first step into exploring deductive and incremental generation of rule-based constraint solvers. A number of topics require more work. It would be interesting to obtain results for types of rules other than inclusion rules; then the problem of finding, and testing, a candidate condition in derive arises. More constraint modifications should be examined. Most of the ones currently given do not relax the constraint, and it appears that this is harder than restricting. The current implementation of the closure computation is not much refined and slow. We believe that a better internal representation and ordering of rule sets should lead to substantial improvements in efficiency. Currently we use an unordered list which for instance subsume needs to scan completely to find a subsuming rule. Also derive should prefer more general rules to construct descendants. The complexity of the closure computation is an open question. While it is clear that termination must occur, it is not obvious what the best strategy to achieve it is.

Acknowledgement

This work benefited from discussions with Lucas Bordeaux. We are grateful for the helpful suggestions of the reviewers.

References

[1] Slim Abdennadher and Thom Frühwirth. Using program analysis for integration and optimization of rule-based constraint solvers. In *Journées Francophones de Progr. Logique et Progr. par Contraintes (JFPLC'2002)*, 2002.

[2] Slim Abdennadher and Christophe Rigotti. Automatic generation of rule-based solvers for intentionally defined constraints. *International Journal on Artificial Intelligence Tools*, 11(2), 2002.

[3] Krzysztof R. Apt and Eric Monfroy. Constraint programming viewed as rule-based programming. *Theory and Practice of Logic Programming*, 2001.

[4] Lucas Bordeaux and Eric Monfroy. Beyond NP: Arc-consistency for quantified constraints. *Lecture Notes in Computer Science*, 2470, 2002.

[5] Sebastian Brand. A note on redundant rules in rule-based constraint programming. *Lecture Notes in Computer Science*, 2627, 2003.

[6] Björn Carlson, Mats Carlsson, and Sverker Janson. The implementation of AKL(FD). In *International Symposium on Logic Programming*, 1995.

[7] Yves Caseau, François-Xavier Josset, and François Laburthe. CLAIRE: Combining sets, search and rules to better express algorithms. *Theory and Practice of Logic Programming*, 2(6), 2002.

[8] Carlos Castro. Building constraint satisfaction problem solvers using rewrite rules and strategies. *Fundamenta Informaticae*, 34(3), 1998.

[9] Rina Dechter and Peter van Beek. Local and global relational consistency. *Theoretical Computer Science*, 173(1), 1997.

[10] Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 1998.

[11] Claude Kirchner and Christophe Ringeissen. Rule-based constraint programming. *Fundamenta Informaticae*, 34(3), 1998.

[12] David A. Randell, Zhan Cui, and Anthony Cohn. A spatial logic based on regions and connection. In *Principles of Knowledge Representation and Reasoning (KR'92)*. Morgan Kaufmann, 1992.

[13] C. Ringeissen and E. Monfroy. Generating propagation rules for finite domains via unification in finite algebras. In *New Trends in Constraints*, 2000.

[14] Vijay Anand Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

Typing rule-based transformations over topological collections

Julien Cohen [1,2]

LaMI, umr 8042 CNRS
Université d'Évry Val d'Essonne
523 place des Terrasses de l'Agora
91000 Évry, France

Abstract

Pattern-matching programming is an example of a rule-based programming style developed in functional languages. This programming style is intensively used in dialects of ML but is restricted to algebraic data-types.

This restriction limits the field of application. However, as shown by [9] at RULE'02, case-based function definitions can be extended to more general data structures called *topological collections*. We show in this paper that this extension retains the benefits of the typed discipline of the functional languages. More precisely, we show that topological collections and the rule-based definition of functions associated with them fit in a polytypic extension of mini-ML where type inference is still possible.

1 Introduction

Pattern-matching on algebraic data-types (ADT) allows the definition of functions by cases, a restricted form of rule based programming that is both relevant and powerful to specify function acting on ADTs. pattern matching, where only the top-level structure of an ADT is matched against the pattern [15]. Examples of more expressive patterns are given, e.g., by the Mathematica language. the handling of terms, that is, tree-shaped data structures (sets or multisets handled in Mathematica are represented by terms modulo associativity and commutativity).

[1] The author is grateful to Olivier Michel and Jean-Louis Giavitto of the MGS Project for their valuable support. The MGS home page is located at `http://mgs.lami.univ-evry.fr`.

[2] Email: `jcohen@lami.univ-evry.fr`

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

In [9] and [8] a framework where pattern matching can be expressed uniformly on many different data structures is exhibited. They rely on the notion of topological collection which embeds a neighborhood relation over its elements. The neighborhood relation enables the definition of a general notion of path (a sequential specification of a sub-structure); a pattern is used to specify a path that selects an arbitrary sub-collection to be substituted. This leads to a general functional language where the pattern matching is not limited to ADTs.

We show in this paper that the topological collections bring a smooth extension of the Hindley-Milner type system [10][14] with some polytypism [12] and we suggest an extension of the Damas-Milner type inference algorithm that allows to find a type to programs expressed in an extension of mini-ML with topological collections and rule based transformations over them.

Section 2 gives a brief description of the topological collections and their transformation; section 3 gives an overview of types in this framework; the types are investigated in section 4 where the typing rules and the inference algorithm are given; several direct extensions of the language are discussed in section 5 and section 6 concludes this paper.

2 Topological Collections and Transformations

Topological collections are data structures corresponding conceptually to a mapping from a set of positions into a set of values such that there is a neighborhood relation over the positions. Two values of a collection are said to be neighbors if their positions are neighbors. The *sequence* is an example of topological collection where the elements have at most a *left* neighbor and a *right* neighbor. The NEWS *grid* which is a generalization of arrays of dimension 2 is another example where each element has at most four neighbors, considering a Von Neumann neighborhood [13].

The notion of neighborhood is a means to embed in the programming language the spatial locality of computations of programs.

Many other data structures can be seen from the topological point of view. For example the *set* and the *multi-set* (or *bag*) are topological collections where each element is neighbor of each other element (the set of positions of a set, is the set of the elements itself). See [6] for other examples of topological collections.

These data structures come with a rule based style of programming: a rule defines a local transformation by specifying some elements to be matched and the corresponding action. The topological disposition of the matched elements is expressed directly within the pattern of the rule. Thus a collection can be transformed by the simultaneous application of local transformations to non-intersecting matching sub-sets of the collection.

The MGS programming language described in [6] and [8] supplies the

topological collections as first-class values and transformations as a means to describe rule based functions over collections. The language we work on in our paper is largely inspired by MGS although some features such as the possibility for a collection to contain elements of different types have been left out.

In the rest of this section we describe the handling of collections via rules in our restriction of MGS.

A *rule* is written p`=>`e where p is the pattern and e is the expression that will replace the instances of p. A *transformation* is a list of rules introduced by the keyword `trans`. The application of a transformation `trans[`p_1`=>`e_1`;` p_2`=>`e_2`]` to a collection c consists in selecting a number of non-intersecting occurrences of p_1 in c such that there is no further possible occurrence; then replacing the selected parts by the appropriate elements calculated from e_1; then selecting a number of non-intersecting occurrences of p_2 and replacing them with the appropriate values.

The pattern can be a single element x or a single element satisfying a condition x/e where e is a boolean expression; it can also be a two elements pattern x, y such that y is a neighbor of x. Here the comma expresses the neighborhood relation and is not intended to express a tuple. The pattern $x/(x = 0), y/(y = 1), z/(z = 2)$ matches three values such that the first is a 0, the second is a 1, the third is a 2, the second is in the neighborhood of the first and the third is in the neighborhood of the second.

The right hand side of the rule is composed of an expression denoting the elements replacing the selected elements. In order to allow the replacement of parts by parts of different size, the value expressed in the right hand side of a rule must be a sequence. The elements of this sequence will substitute the matched elements. Thus we can consider rules replacing sub-parts constituted of a single element with several element, or sub-parts constituted of several elements with one element or even with no element, and so on.

A way of building a sequence is using the empty sequence `empty_seq` and the constructor `::`. The syntactic shortcut `[`e`]` can be used to express e`::empty_seq`.

2.1 Two examples

The following two examples show two programs acting respectively on sequences and sets.

Sorting a Sequence.

A kind of bubble-sort is immediate:

```
trans[ x, y/(y<x) => y ::  x ::  empty_seq ; x => [x] ]
```

This two rules transformation has to be applied on the sequence until a fixpoint is reached. The fixpoint is a sorted sequence.

This is not really the bubble-sort because the swapping of elements can happen at arbitrary places; hence an out-of-order element does not necessarily bubble to the top in the characteristic way.

We will see in section 4 that the rule `x => [x]` is required.

Eratosthene's Sieve on a Set.

The idea is to apply the transformation on the set of the integers between 2 and n. The transformation replaces an x and an y such that x divides y by x. The iteration until a fixpoint of this transformation results in the set of the prime integers less than n.

```
trans [ x, y/(y mod x = 0) => [x] ; x => [x] ]
```

3 Typing the Collections and the Transformations

The type of a topological collection is described by two pieces of information: the type of the elements inside the collection and its organization. The former is called its *content type* and the latter its *topology* (see [11] for an example of separation between the shape and the data). For example, a set of integers and a set of strings do not have the same content type but have the same topology. Collection types will be denoted by $[\tau]\rho$ where τ is the content type and ρ is the topology. Thus a set of strings will have the type [`string`]`set`.

The usual notion of polymorphism of ML languages is provided on the content type. For example the *cardinal* function that returns the number of elements of a set would have the type $[\alpha]$`set` $\rightarrow$ `int` where α is a free type variable since it can be applied to a set irrespectively of the type of its elements. The nature of the content type does not affect the behavior of the cardinal function, therefore the polymorphism is said to be uniform on the content type.

Instead of providing different functions that count the number of elements for each topology, the language provides the function *size* with the type $[\alpha]\theta \rightarrow$ `int` where θ is a free topology variable. Functions that accept any kind of topology are said to be *polytypic* [12].

A way of handling collections is using polytypic operators and constant collections: the constructor operator `::` has the type $\alpha \rightarrow [\alpha]\theta \rightarrow [\alpha]\theta$; the destructors `oneof` and `rest` have the type $[\alpha]\theta \rightarrow \alpha$ and $[\alpha]\theta \rightarrow [\alpha]\theta$ and are such that for any collection c, `oneof(`c`)` and `rest(`c`)` make a partition of c (see [3]).

The constant collections are `empty_set`, `empty_seq` and so on.

Collections can also be handled with transformations. As seen in the previous section, transformations are functions on collections described by rewriting rules. This kind of function is introduced by the keyword `trans`. For example the function `trans [ x=>[x] ]` implements the identity over collections and has the type $[\alpha]\theta \rightarrow [\alpha]\theta$. It is the identity because it maps the identity to all the elements of the collection.

As we said, the right hand side of a rule must be a sequence because the pattern matched can be replaced by a different number of elements. On some topologies such as the *grid*, the pattern and the replacement sequence must have the same size. If the sizes are not compatible a *structural error* will be raised at execution time. These structural errors are not captured by our type system. See [6] for more details on the substitution process in the collections.

The *map* function can be expressed as follows:

```
fun f -> trans [ x => [f x] ]
```

and has the type $(\alpha \rightarrow \beta) \rightarrow [\alpha]\theta \rightarrow [\beta]\theta$.

Unlike in the original MGS language, a collection cannot contain elements of different types. We have chosen to set this restriction to allow to build an inference algorithm in the Damas-Milner style [5]. Allowing such heterogeneous collections would lead to a system with subsumption and union types that would need complex techniques to determine the types of a program.

4 The Language

In this section we first describe the syntax of the studied language. Then we describe the type verification rules and finally we give the type inference algorithm that computes the principal type of a program.

4.1 Syntax

Topological collections are values manipulated with constants, operators, functions and transformations, no new syntactic construction is needed.

For the transformation we have to enrich the syntax of mini-ML [4] as shown in figure 2.

The construction $p => e$ is called a *rule* and a transformation is a syntactic list of rules. In the construction id/e occurring in a pattern, e is called a *guard*.

The last rule of a transformation must be a variable for exhaustiveness purpose. Putting the rule `x => [x]` in last position of a transformation

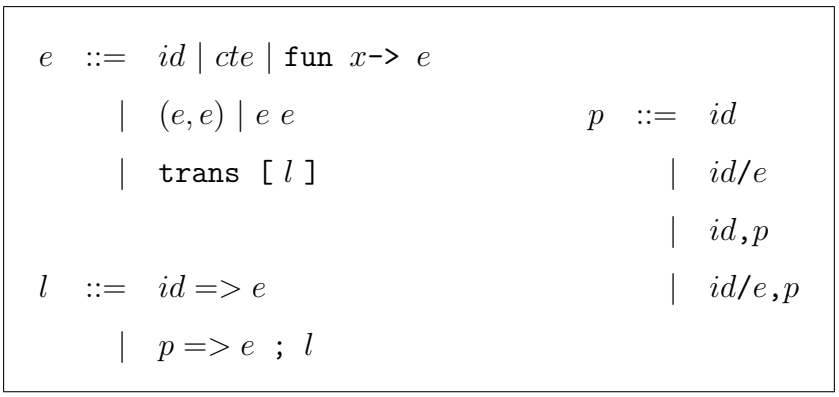


Fig. 2. Syntax of the language

expresses that all unmatched values are left unmodified. It is not possible to infer a relevant default case for a transformation. For example the rule `x => [x]` cannot be the default case for a transformation of the type $[string]\theta \rightarrow [int]\theta$. Therefore the default case must be specified explicitly by the programmer. This explains the grammar for the list of rules l which enforces the presence of a last rule of the form $id => e$ matching every remaining element. The expression e in the right hand side provides the appropriate default value.

We will use some operators such as `::` in an infix position but this syntax can be easily transformed into the one of figure 2. Operators are functional constants of the language.

4.2 The Type System

Types Algebra

We enrich the polymorphic type system of mini-ML with the topological collections. The collection type introduces a new kind of construction in types: the *topology*.

From a type point of view, transformations are just functions that act on topological collections without changing their topology, so no new construct is needed for them in the type algebra.

Types :	$\tau ::=$	T	base type (`int`, `float`, `bool`, `string`)
		α	type variables
		$\tau \rightarrow \tau$	functions
		$\tau \times \tau$	tuples
		$[\tau]\rho$	collections

Topologies :	$\rho ::=$	R	base topology (`bag`, `set`, `seq`, `grid`, ...)
		θ	topology variables

We give in appendix A the definitions of $\mathcal{L}_t$ and $\mathcal{L}_r$ which calculate the type variables and the topology variables occurring in a type.

Type Schemes
A type scheme is a type quantified over some type variables and some topology variables:

$$\sigma ::= \forall[\alpha_1, \ldots, \alpha_n][\theta_1, \ldots, \theta_m].\tau$$

A type τ is an instance of a type scheme $\sigma = \forall[\alpha_1, \ldots, \alpha_n][\theta_1, \ldots, \theta_m].\tau'$ and we write $\sigma \leqslant \tau$ if and only if there are some types $\tau_1, \ldots, \tau_n$ and some topologies $\rho_1, \ldots, \rho_m$ such that $\tau = \tau'[\alpha_1 \leftarrow \tau_1, \ldots, \alpha_n \leftarrow \tau_n, \theta_1 \leftarrow \rho_1, \ldots, \theta_m \leftarrow \rho_m]$.

In the following, an environment is a function from identifiers to type schemes.

TC is the function that gives the type scheme of the constants of the language. For example $TC(::)$ is $\forall[\alpha][\theta].\alpha \rightarrow [\alpha]\theta \rightarrow [\alpha]\theta$.

$\mathcal{L}_t$ and $\mathcal{L}_r$ are extended to type schemes and calculate the free variables of a type scheme, that is the variables occurring in the type scheme which are not bound by the quantifier. For example if σ is $\forall[\alpha_1][\theta_1].[\alpha_1]\theta_1 \rightarrow [\alpha_2]\theta_2$ then $\mathcal{L}_t(\sigma)$ is α_2 and $\mathcal{L}_r(\sigma)$ is θ_2.

Typing Rules
The typing rules are nearly the same as the Hindley-Milner rules [10][14]. The differences are that a rule has been added for the transformations and that the notions of instance and the *Gen* function have been adapted to the type algebra.

The *Gen* function transforms a type into a type scheme by quantifying the variables that are free in the type and that are not bound in the current environment. The definition of *Gen* is the following:

$Gen(\tau, \Gamma) = \forall[\alpha_1, \ldots, \alpha_n][\theta_1, \ldots, \theta_m].\tau$ with $\{\alpha_1, \ldots, \alpha_n\} = \mathcal{L}_t(\tau) \backslash \mathcal{L}_t(\Gamma)$ and $\{\theta_1, \ldots, \theta_m\} = \mathcal{L}_r(\tau) \backslash \mathcal{L}_r(\Gamma)$.

The typing rules are:

$$\frac{\Gamma(x) \leqslant \tau}{\Gamma \vdash x : \tau}\,(var - inst) \qquad \frac{TC(c) \leqslant \tau}{\Gamma \vdash c : \tau}\,(const - inst)$$

$$\frac{\Gamma \cup \{x : \tau_1\} \vdash e : \tau_2}{\Gamma \vdash (\texttt{fun}\ x \to e) : \tau_1 \to \tau_2}\,(fun) \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}\,(app)$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \cup \{x : Gen(\tau_1, \Gamma)\} \vdash e_2 : \tau_2}{\Gamma \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) : \tau_2}\,(let)$$

$$\frac{\begin{array}{c}\left\{\Gamma \cup \{x_i^j : \tau\}_{(j \leqslant m_i)} \cup \{\texttt{self} : [\tau]\rho\} \vdash e_i : \tau'\right\}_{(i \leqslant n)} \\ \left\{\Gamma \cup \{x_i^j : \tau\}_{(j \leqslant k)} \cup \{\texttt{self} : [\tau]\rho\} \vdash e_i^k : \texttt{bool}\right\}_{(i \leqslant n),(k \leqslant m_i)}\end{array}}{\Gamma \vdash \texttt{trans [}x_1^1/e_1^1\texttt{,...,}x_1^{m_1}/e_1^{m_1}\texttt{=>}e_1\texttt{;...;}x_n^1/e_n^1\texttt{,...,}x_n^{m_n}/e_n^{m_n}\texttt{=>}e_n\texttt{]} : [\tau]\rho \to [\tau']\rho}(trans)$$

In the (trans) rule, k_n is always equal to 1 and e_n^1 is always equal to `true`.

Inside a rule the `self` identifier refers to the collection the transformation is applied on.

The (trans) rule expresses that a transformation has the type $[\tau]\rho \to [\tau']\rho$ if when you suppose that all the x_i^j have the same type τ and that $self$ has the type $[\tau]\rho$ it can be proven that the e_i^j are boolean values and that the e_i have the type $[\tau']seq$.

We can see that if `self` is not used in a transformation, this one will be polytypic since ρ will not be bound to any topology.

The following examples show a type verification on a polytypic transformation and on a non-polytypic one.

Polytypic Example

The following transformation can be proven to be an $[int]\theta \to [int]\theta$ function for any topology θ.

```
trans [ x, y/x>y => x ::  y ::  (x-y) ::  empty_seq ; x => [x] ]
```

The proof is given in figure 3a where $\Gamma_0 = \{x : int; y : int; self : [int]\theta\}$, $\Gamma_1 = \{x : int; self : [int]\theta\}$ and with the following lemmas:

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : [int]seq}{\Gamma \vdash e_1 :: e_2 : [int]seq} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{[}e\texttt{]} : [\tau]seq}$$

Non-Polytypic Example

The operator `is_left` acts as a predicate that returns *true* if the element is at the left extremity of the sequence. Thus it returns *false* is the element

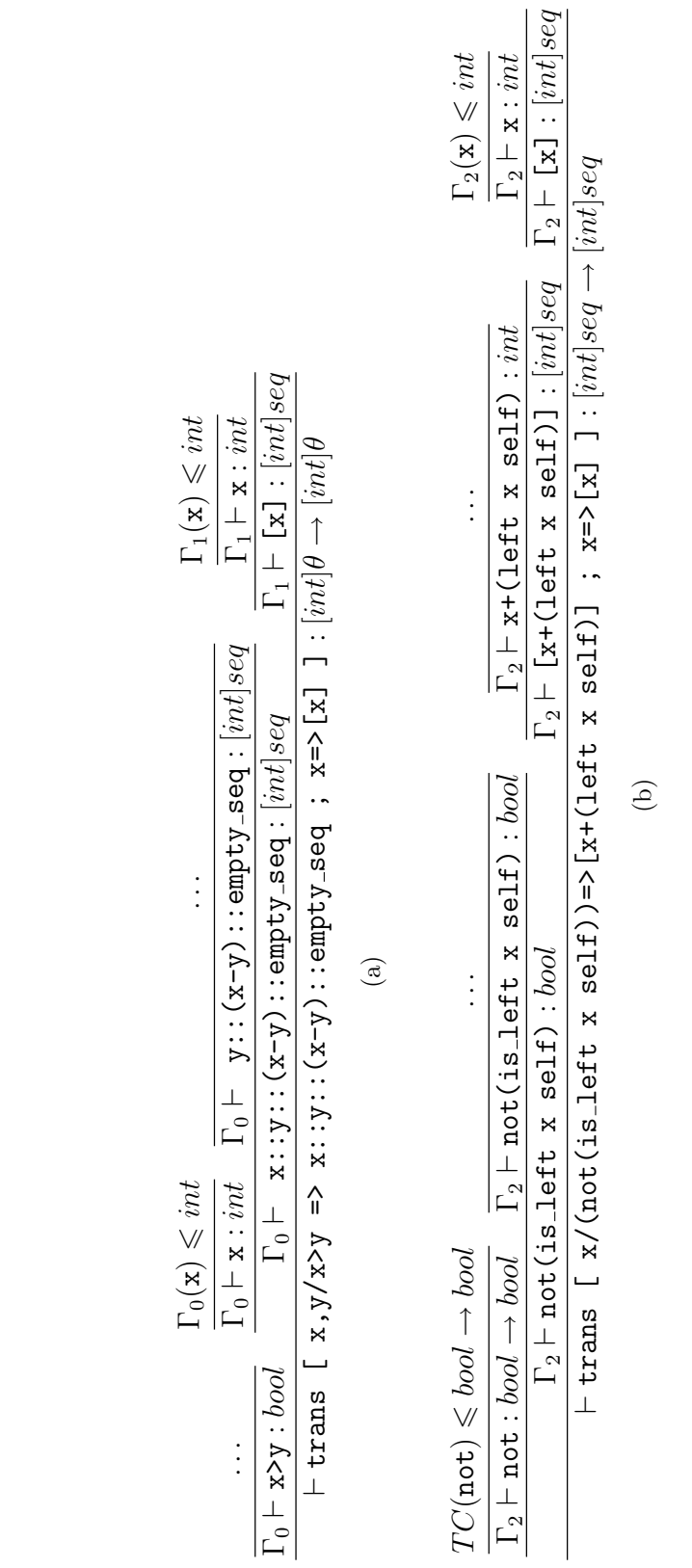


Fig. 3. Two examples of type verification

has a left neighbor. It can be used only within a transformation [3] and takes two arguments: the first is a pattern variable and the second is a collection. Similarly, the operator `left` takes a pattern variable x and a sequence s and returns the left neighbor of x in s.

Let us consider the following transformation:

```
trans [ x/(not (is_left x self)) => [x+(left x self)] ; x=>[x] ]
```

This transformation does not have the same effect as the following one:

```
trans [ l, x => (l ::  l+x ::  empty_seq) ; x=>[x] ]
```

because in the former, every element x of the sequence except the leftmost one will be replaced by the sum of itself and its left neighbor whereas in the latter, the l element will be replaced by itself and thus will not be increased. For example the former transformation applied to the sequence `(1::2::3::4::empty_seq)` results in `(1::3::5::7::empty_seq)` whereas the application of the latter transformation to the same sequence would result in `(1::4::3::7::emty_seq)`.

The figure 3b where $\Gamma_2 = \{x : int; self : [int]seq\}$ proves that the first transformation has the type $[int]seq \rightarrow [int]seq$.

This transformation cannot be proven to have the type $[int]\rho \rightarrow [int]\rho$ if $\rho \neq seq$ because `left` and `is_left` act exclusively on sequences.

4.3 Type Inference

The typing rules given in section 4.2 are a means to verify that a program has a given type but this type is a parameter of the verification procedure. We now give the equivalent of the Damas-Milner type inference that enables the full automated type verification since it computes the principal type of a program. The resulting type is said to be principal because every type that can fit the program is an instance of this type.

The type inference algorithm is given after the unification procedure.

Unification

Unifying two types τ_1 and τ_2 consists in finding a substitution φ over the free variables of τ_1 and τ_2 called the unifier such that $\varphi(\tau_1) = \varphi(\tau_2)$.

A substitution is a most general unifier (mgu) for two types τ_1 and τ_2 if for any unifier φ_1 of τ_1 and τ_2, there is a substitution φ_2 such that $\varphi = \varphi_2 \circ \varphi_1$.

We give the `mgu` function that computes the most general unifier of a set of pairs of types denoted by $\tau_1 = \tau_2$. This function is necessary to the type inference procedure. If `mgu` fails then there is no unifier for the given types.

[3] The `is_left` operator is only available in transformations, where the identifiers introduced by the pattern are bound to a position in the collection. Allowing only such identifiers to be arguments of `is_left` allows to remove any ambiguity on the position denoted in the sequence, even if the position contains a value occurring several times.

The difference between our mgu and Damas and Milner's original *mgu* is the addition of the case for the collection types. Two collection types are unified by unifying their content types and their topologies. The substitution doing this unification is found as $\varphi_1 \circ \varphi_2$ where φ_2 unifies the topologies and φ_1 unifies the content types. The computation of φ_2 is made by the dedicated $\mathtt{mgu}_r$ function. This function fails when the two topologies are different base topologies since they cannot be unified. The substitution φ_2 is applied to the content types before computing φ_1 with mgu.

The standard cases of the definition of mgu are:

$$\begin{aligned}
\mathtt{mgu}(\varnothing) &= [\,] \\
\mathtt{mgu}(\{\tau = \tau\} \cup C) &= \mathtt{mgu}(C) \\
\mathtt{mgu}(\{\alpha = \tau\} \cup C) \text{ (if } \alpha \text{ is not free in } \tau) &= \text{let } \varphi = [\alpha \leftarrow \tau] \text{ in } \mathtt{mgu}(\varphi(C)) \circ \varphi \\
\mathtt{mgu}(\{\tau = \alpha\} \cup C) \text{ (if } \alpha \text{ is not free in } \tau) &= \text{let } \varphi = [\alpha \leftarrow \tau] \text{ in } \mathtt{mgu}(\varphi(C)) \circ \varphi \\
\mathtt{mgu}(\{\tau_1 \to \tau_2 = \tau_1' \to \tau_2'\} \cup C) &= \mathtt{mgu}(\{\tau_1 = \tau' 1 \ ; \ \tau_2 = \tau' 2\} \cup C) \\
\mathtt{mgu}(\{\tau_1 \times \tau_2 = \tau_1' \times \tau_2'\} \cup C) &= \mathtt{mgu}(\{\tau_1 = \tau' 1 \ ; \ \tau_2 = \tau' 2\} \cup C)
\end{aligned}$$

The new case for the collections is:

$$\mathtt{mgu}(\{[\tau]\rho = [\tau']\rho'\} \cup C) = \text{let } \varphi = \mathtt{mgu}_r(\rho = \rho') \text{ in } \mathtt{mgu}(\varphi(\{\tau = \tau'\} \cup C)) \circ \varphi$$

The unification of topologies is defined by:

$$\begin{aligned}
\mathtt{mgu}_r(\rho = \rho) &= [\,] \\
\mathtt{mgu}_r(\theta = \rho) &= [\theta \leftarrow \rho] \\
\mathtt{mgu}_r(\rho = \theta) &= [\theta \leftarrow \rho]
\end{aligned}$$

Type Inference

The type reconstruction algorithm is nearly the same as the Damas-Milner one. The differences are that it uses specialized versions of *mgu* and *Gen* functions and that there is a new case for the transformations. It is described here in an imperative way: φ is the current substitution and V_t and V_r are sets of free type variables and topology variables.

The algorithm is given in figure 4.

The case for the transformations consists in unifying the types of all the pattern variables and unifying the types of the right hand side rules together and with a sequence collection type. These unifications have to be made with respect to the guards that are boolean values.

If W succeeds it computes the most general type of the program analyzed and this one can be run without type error. If it fails because of an mgu or an $\mathtt{mgu}_r$ failure then the program is ill-typed and might lead to a type error at execution time.

$$
\begin{array}{rcl}
\texttt{fresh\_t} & = & \text{let } \alpha \in V_t \\
 & & \text{do } V_t \leftarrow V_t \backslash \{\alpha\} \\
 & & \text{return } \alpha \\
\texttt{fresh\_r} & = & \text{let } \theta \in V_r \\
 & & \text{do } V_r \leftarrow V_r \backslash \{\theta\} \\
 & & \text{return } \theta
\end{array}
$$

$W(\Gamma \vdash e) =$
`(* original cases *)`
If $e = x$
  let $\forall[\alpha_1, \ldots, \alpha_n][\theta_1, ..., \theta_m].\tau = \Gamma(x)$
  let $\alpha'_1, \ldots, \alpha'_n =$ `fresh_t`$, \ldots,$ `fresh_t`
  let $\theta'_1, \ldots, \theta'_m =$ `fresh_r`$, \ldots,$ `fresh_r`
  return $\tau[\alpha_1 \leftarrow \alpha'_1, \ldots, \alpha_n \leftarrow \alpha'_n, \theta_1 \leftarrow \theta'_1, \ldots, \theta_m \leftarrow \theta'_m]$
If $e =$ `fun` $x \rightarrow e$
  let $\alpha =$ `fresh_t`
  let $\tau = W(\Gamma \cup x : \forall[\,][\,].\alpha \vdash e)$
  return $\alpha \rightarrow \tau$
If $e = e_1\ e_2$
  let $\tau_1 = W(\Gamma \vdash e_1)$
  let $\tau_2 = W(\Gamma \vdash e_2)$
  let $\alpha =$ `fresh_t`
  do $\varphi \leftarrow$ `mgu`$(\varphi(\tau_1) = \varphi(\tau_2 \rightarrow \alpha)) \circ \varphi$
If $e =$ `let` $x = e_1$ `in` e_2
  let $\tau_1 = W(\Gamma \vdash e_1)$
  let $\sigma = Gen(\varphi(\tau_1), \varphi(\Gamma))$
  return $W(\Gamma \cup \{x : \sigma\} \vdash e_2)$

`(* new case for the transformations *)`
If $e =$ `trans [`p_1`=>`e_1`; ...;` p_n`=>`e_n`]`
  let $\alpha, \beta =$ `fresh_t, fresh_t`
  let $\theta =$ `fresh_r`
  `for` $i = 1..n$
    let $id_i^1/e_i^1, \ldots, id_i^{m_i}/e_i^{m_i} = p_i$
    `for` $j = 1..m_i$
      let $\tau_i^j = W\big(\Gamma \cup \{\texttt{self} : [\alpha]\theta\} \cup \{id_i^k : \alpha\}_{k \leqslant j} \vdash e_i^j\big)$
      do $\varphi \leftarrow$ `mgu`$\big(\{\varphi(\tau_i^j) = bool\}\big) \circ \varphi$
    let $\tau_i = W\big(\Gamma \cup \{\texttt{self} : [\alpha]\theta\} \cup \{id_i^k : \alpha\}_{k \leqslant m_i} \vdash e_i\big)$
    do $\varphi \leftarrow$ `mgu`$\big(\{\varphi(\tau_i) = \varphi([\beta]\texttt{seq})\}\big) \circ \varphi$
  return $[\alpha]\theta \rightarrow [\beta]\theta$

Fig. 4. Type inference algorithm

5 Extensions

5.1 Repetition in a Pattern

The *star* `*` expressing an arbitrary repetition of a sub-pattern during the matching process has been introduced in [9]. The pattern `x/(x=0), * as y, z/(z=0)` for example can match an arbitrary subcollection such that it contains two 0 and that there is a *path* between these 0. This means that one can reach the second 0 from the first one only by going from an element to one of its neighbors repetitively.

To take the star into account we modify the syntax of the patterns as follows:

$$\begin{aligned} p &::= q \mid q, p \\ q &::= id \mid * \ as \ id \end{aligned}$$

where q stands for elementary patterns.

We have not kept the guards in the elementary patterns in order to keep the formulas readable but their addition does not lead to new problems.

The elements matched by the star are named and can be referred to as a sequence.

The star could have been considered as a repetition of a subpattern as in `(x,y/x=y)*` but we have chosen to restrict the star to the repetition of single elements for the sake of simplicity.

Before giving the new typing rule, we introduce a function which gives the type binding corresponding to an elementary pattern: $b(q, \tau)$ is such that $b(x, \tau) = (x : \tau)$ and $b(\texttt{* as } x, \tau) = (x : [\tau]seq)$. This function is used in the *trans* typing rule which is modified as follows:

$$\frac{\left\{\Gamma \cup \{b(q_i^j, \tau)\}_{j \leqslant m_i} \cup \{\texttt{self} : [\tau]\rho\} \vdash e_i : [\tau']\texttt{seq}\right\}_{i \leqslant n}}{\Gamma \vdash \texttt{trans [}q_1^1\texttt{,...,}q_1^{m_1}\texttt{=>}e_1\texttt{;...;}q_n^1\texttt{,...,}q_n^{m_n}\texttt{=>}e_n\texttt{]} : [\tau]\rho \rightarrow [\tau']\rho} \ (trans')$$

5.2 Directions in Patterns

In section 4.2 we saw the operator `left` that returns the left neighbor of an element in a sequence. In the framework of topological collections, a topology can supply several neighborhood operators. For example `left` and `right` are the neighborhood operators of the sequence and `north` and `east` are neighborhood operators of the grid. Neighborhood operators are also called *directions*.

A direction can be used to refine the patterns: the commas of the pattern can be substituted by a direction to restrict the accepted neighbors for the rest of the pattern. The substituting direction is surrounded with the symbols

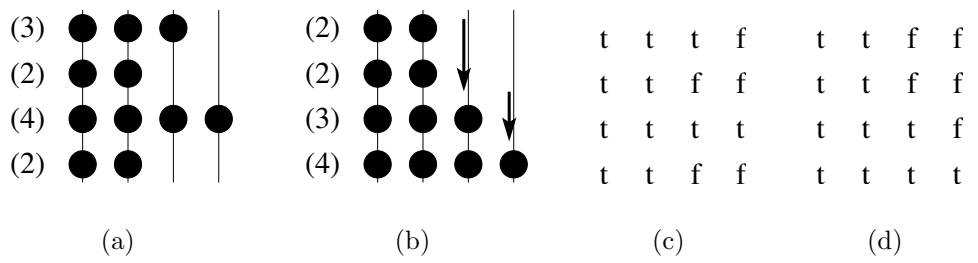


Fig. 5. The Bead-Sort

`|` and `>` to sketch a kind of arrow.

For example if d is a direction we can use the pattern `x |d> y` which is a shortcut [4] for `x,y/y=(d x self)`. However, the pattern `x |d> y` allows faster research of the instances of the pattern in the collection than `x,y/y=(d x self)`.

The pattern `x |d> y` can be typed as `x,y/y=(d x self)`.

The Bead-Sort Example

The bead-sort is an original way of sorting positive integers presented by [2]. The sorting algorithm considers a column of numbers written in unary basis. Figure 5a shows the numbers 3, 2, 4 and 2 where the beads stand for the digits. The sorting is done by letting the beads fall down as shown on figure 5b.

The problem can be represented on a grid of booleans where *true* stands for a digit and *false* for the absence of digit as shown on figure 5c. The bead-sort is achieved by iterating the application of the following transformation until a fixpoint is reached:

```
trans [ x/x=false |north> y/y=true => y::x::empty_seq ; x=>[x] ]
```

The first rule of this transformation is expressed as

```
x/x=false , y/(y=true && y=north x self) => y::x::empty_seq
```

in order to fit the type system. The result of W on this transformation is $[bool]grid \rightarrow [bool]grid$.

5.3 Strategies

As far as the rules application strategy guarantees that every element of the collection is matched (this is always possible since the last rule always matches) the type system is not affected.

For instance, the MGS language provides several strategies such as higher priority given to the first rules or random application of the rules.

[4] The expression `y=(d x self)` in a guard where `y` is a pattern variable and `d` is a direction tests that the values denoted are the same and that their positions in the collection are the same. See the MGS manual [7] for more details.

6 Conclusion

Including the topological collections and pattern matching programming on these structures in the ML framework allows to bring together a powerful programming language with a rule programming framework common to several other languages.

Our algorithm has been tested on MGS programs and has been included in a prototype MGS compiler in order to achieve type-oriented optimizations on the produced code. We believe that the best pattern matching algorithms would be wasted on a dynamically typed language and thus a type inference algorithm is an important step in the development of an efficient compiler for rule based transformations.

However some restrictions on the MGS language had to be done in order to keep the simplicity of the Damas-Milner algorithm. We are currently working on a type inference system with union types [1] to account for heterogeneous collections supplied by the MGS language.

Finally, we said that an error could occur when a transformation tries to replace a subpart by a part of different shape on topologies as the grid which cannot get out of shape. Such errors are not type errors but some of them could be detected statically with a specific type based analysis. Some research such as [11] manage with this kind of error but the concerned languages do not provide the flexibility of the rule based transformations proposed here.

A Free Variables

The free variables of a type are the variables occurring in that type. $\mathcal{L}_t$ computes the free type variables whereas $\mathcal{L}_r$ computes the free topology variables.

$$\begin{aligned}
\mathcal{L}_t(T) &= \varnothing \\
\mathcal{L}_t(\alpha) &= \{\alpha\} \\
\mathcal{L}_t(\tau_1 \to \tau_2) &= \mathcal{L}_t(\tau_1) \cup \mathcal{L}_t(\tau_2) \\
\mathcal{L}_t(\tau_1 \times \tau_2) &= \mathcal{L}_t(\tau_1) \cup \mathcal{L}_t(\tau_2) \\
\mathcal{L}_t([\tau]\rho) &= \mathcal{L}_t(\tau)
\end{aligned}$$

$$\begin{aligned}
\mathcal{L}_r(T) &= \varnothing \\
\mathcal{L}_r(\alpha) &= \varnothing \\
\mathcal{L}_r(\tau_1 \to \tau_2) &= \mathcal{L}_r(\tau_1) \cup \mathcal{L}_r(\tau_2) \\
\mathcal{L}_r(\tau_1 \times \tau_2) &= \mathcal{L}_r(\tau_1) \cup \mathcal{L}_r(\tau_2) \\
\mathcal{L}_r([\tau]\theta) &= \{\theta\} \cup \mathcal{L}_r(\tau) \\
\mathcal{L}_r([\tau]R) &= \mathcal{L}_r(\tau)
\end{aligned}$$

The free variables of a type scheme are the non-quantified variables occurring in it:

$$\mathcal{L}_t(\forall[\alpha_1,\ldots,\alpha_n],[\theta_1,\ldots,\theta_m].\tau) = \mathcal{L}_t(\tau)\backslash\{\alpha_1,\ldots,\alpha_n\}$$

$$\mathcal{L}_r(\forall[\alpha_1,\ldots,\alpha_n],[\theta_1,\ldots,\theta_m].\tau) = \mathcal{L}_r(\tau)\backslash\{\theta_1,\ldots,\theta_m\}$$

References

[1] Aiken, A. and E. Wimmers, *Type inclusion constraints and type inference*, in: *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture*, 1993, pp. 31–41.

[2] Arulanandham, J. J., C. S. Calude and M. J. Dinneen, *Bead-Sort: A natural sorting algorithm*, EATCS Bull **76** (2002), pp. 153–162.

[3] Buneman, P., S. Naqvi, V. Tannen and L. Wong, *Principles of programming with complex objects and collection types*, Theoretical Computer Science **149** (1995), pp. 3–48.

[4] Clement, D., J. Despeyroux, T. Despeyroux and G. Kahn, *A simple applicative language: Mini-ML*, in: *Proceedings of the ACM conference on LISP and Functional Programming*, 1986, pp. 13–27.

[5] Damas, L. and R. Milner, *Principal type-schemes for functionnal programs*, in: *Proceedings of the 15'th Annual Symposium on Principles of Programming Languages* (1982), pp. 207–212.

[6] Giavitto, J. and O. Michel, *MGS: a rule-based programming language for complex objects and collections*, in: M. van den Brand and R. Verma, editors, *Electronic Notes in Theoretical Computer Science*, **59** (2001).

[7] Giavitto, J.-L. and O. Michel, *MGS: a programming language for the transformations of topological collections*, Technical Report lami-61-2001, LaMI Université d'Évry Val d'Essonne (2001).

[8] Giavitto, J.-L. and O. Michel, *Data structure as topological spaces*, in: *Proceedings of the 3nd International Conference on Unconventional Models of Computation UMC02*, **2509**, Himeji, Japan, 2002, pp. 137–150, Lecture Notes in Computer Science.

[9] Giavitto, J.-L. and O. Michel, *Pattern-matching and rewriting rules for group indexed data structures*, in: *RULE'02* (2002), pp. 55–66.

[10] Hindley, J., *The principal type scheme of an object in combinatory logic*, Transactions of the American Mathematical Society **146** (1969), pp. 29–60.

[11] Jay, C. B., *A semantics for shape*, Science of Computer Programming **25** (1995), pp. 251–283.

[12] Jeuring, J. and P. Jansson, *Polytypic programming*, in: J. Launchbury, E. Meijer and T. Sheard, editors, *Advanced Functional Programming, Second International School* (1996), pp. 68–114, LNCS 1129.

[13] Lisper, B. and P. Hammarlund, *On the relation between functional and data-parallel programming languages*, in: *Proc. of the 6th. Int. Conf. on Functional Languages and Computer Architectures*, ACM, 1993, pp. 210–222.

[14] Milner, R., *A theory of type polymorphism in programming*, Journal of Computer and System Sciences **17** (1978), pp. 348–375.

[15] Wadler, P., "Efficient compilation of pattern matching," Prentice-Hall, 1987 Ch. 6 of "The Implementation of Functionnal Programming Language", S. L. Peyton Jones.

A Tool Support for Reusing ELAN Rule-Based Components [1]

Anamaria Martins Moreira [a,2], Christophe Ringeissen [b,3], Anderson Santana [a,4]

[a] *Universidade Federal do Rio Grande do Norte; Departamento de Informática e Matemática Aplicada; Campus Universitário; Lagoa Nova; 59072-970 Natal, RN; Brazil*

[b] *LORIA; Technopôle de Nancy-Brabois; Campus scientifique; 615, rue du Jardin Botanique; BP101; 54602 Villers-lès-Nancy Cedex; France*

Abstract

The adaptation of software components developed for a specific application in order to generate reusable components often includes some kind of generalization. This generalization may be carried out, for instance, by the renaming of some identifiers or by its parameterization. In our work, we are specially interested in the generalization by parameterization of algebraic specification components. Generalization and some other transformations on algebraic specifications are being integrated in the FERUS tool. This tool was initially developed for the Common Algebraic Specification Language, called CASL, and we show in the paper its adaptation to the new version of the rule-based programming language ELAN.

1 Introduction

Formal specifications can provide significant support for software component reuse, as they allow tools to "understand" the semantics of the components they are manipulating. Tools that manipulate programming code can easily deal with syntactic features of components (e.g., renaming operations) but usually have a hard time when it comes to semantics. Formal specifications, with their "simpler" and precisely defined semantics and associated verification tools, contribute to the verification of the validity

[1] This work is supported by a joint research project funded by CNPq and INRIA. It has been partly performed while the first and third authors were visiting LORIA.

[2] Email: anamaria@consiste.dimap.ufrn.br

[3] Email: Christophe.Ringeissen@loria.fr

[4] Email: anderson@consiste.dimap.ufrn.br

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

of semantic properties of components in the different steps of the reuse process. For instance, they can be of great help on the generation of reusable components through the parameterization of more specific ones, supporting the process of creation of reusable components. So, we are interested in the area of programming *for* reuse. However, instead of creating components explicitly for a future reuse, we propose to create reusable components from an already developed application. Both choices have their pros and cons, and the second one presents as main advantage that the generated component has already been used at least once (in its original version), as already advocated e.g. by M. Wirsing in [13]. In this case, the main difficulty is to transform components that were developed for a particular application into effectively reusable components. This must be done through their generalization, and this is where our work fits.

In this paper, we propose a tool called FERUS, to support the reuse of components, via some operations, including one that aims at generalizing algebraic specification components by their parameterization. The main difficulty when trying to generalize by parameterization is to identify the "good" level of generalization for each component. Highly specific ones have small chances of being reused, but on the other hand, if a component is too general, its reuse will often be useless. It is necessary to state the semantic properties of a component that are considered "important" somehow (the component would loose its *raison d'être* if these properties were not satisfied). So, we need to be able to identify the requirements that a formal parameter should satisfy in order to preserve these stated properties in the generalization. When these stated properties are derived from equational axioms, a subset can be extracted from them and added to the formal parameter so that they are preserved in the generalized component. This simple technique provides sufficient conditions for the validity of the considered properties in the models of the more general specification, with the advantage of being easily computed by a simple algorithm.

The FERUS tool, which was first developed to deal with CASL [5] specifications, has the goal of supporting formal specification components development. It provides an environment for specification design and prototyping, that allows to edit, compile and execute specifications (given that the specification is executable). Its main feature is the possibility to derive new components through reuse driven transformation operations, for example, to create an instance of a parameterized module using the instantiate operation, and conversely to create a parameterized module by abstracting some sorts and related declarations, with the generalization operation.

In order to demonstrate that FERUS is suitable to different contexts and/or languages inside the domain of algebraic specifications, we proposed to adapt the tool to the language ELAN [11]. The idea was to keep the tool architecture unchanged, thus only language specific features were subject of adaptation. An interesting feature of applying FERUS to the ELAN language is that although

ELAN has many characteristics of an algebraic specification language, it was not conceived as such and there are some major differences in semantics and structuring constructs. The flexibility of the tool could then be better tested, with promising results.

The paper is organized as follows: Section 2 aims at positioning the ELAN rule-based programming language in the algebraic specification setting. The generalization (meta) operator is presented in Section 3 in a informal way. In the case we want to preserve a set of axioms, we present a simple algorithm to compute the signature morphism and the formal parameter required by the generalization operation. Generalization and other operations on algebraic specifications are implemented in the FERUS tool. In Section 4, we present a detailed view of the FERUS-ELAN instance and we discuss the adaptation issues when considering the ELAN rule-based programming language.

2 Algebraic Specifications and Genericity

Algebraic specifications [14] are a classical paradigm for specifying functional properties of systems. It is a simple and well founded technique which presents some nice characteristics such as *executability* (of some particular specifications), usually carried out by rewriting.

An algebraic specification usually consists of *sort* and *operator* declarations, defining a *signature*; and *axioms* built over this signature (and some set of variables). Axioms describe properties which are expected to be true in all models of the specification. Signature and axioms together are called the *presentation* of the specification. To this presentation corresponds a semantics, which is traditionally presented as the class of algebras that are models of the presentation. Recently, starting with Maude [4] and its underlying rewriting logic [6], on one hand, and ELAN and the ρ-calculus [3] on the other hand, some different interpretations of algebraic specifications have been proposed; but it is the classical approach that we consider here: algebraic structures as models, with a set of values for each declared sort, a function for each declared operator, and rewriting as the computational executability tool.

An algebraic specification language is characterized by many different factors, which can in general be identified as the institution over which specification components are written and its modularity constructs. Each different language has its particularities in both aspects. In this work we consider some characteristics that are available in most of the existing languages: conditional equational logic, many-sorted total operators, and the built-in equality predicate as the unique predicate. With this basic building block, we can then have initial and loose semantics (only the initial algebra or all algebras that satisfy the specification axioms). We can also have structured specifications that import previously defined ones with some constraints that define how imported specification models are to be used in the definition of the models of the importing specification.

In this paper, we consider the new version of ELAN called ELAN 4. This new version borrows the syntax of the ASF+SDF specification language [11], as well as its semantics if we do not consider some features of ELAN: we are dealing with most of the language, except for rewriting strategies[5]. Its underlying institution and model semantics are the same as the for the ASF+SDF language, i.e., $Cond^{=}$ with initial semantics, as classified by Mossakowski in [10]. Structuring is mainly obtained through the importation and renaming of (parts of) imported specifications. Parameterization in these languages is very restricted, however, as only sorts are considered. Also, the semantics of a generic specification and of its instantiation are quite special, so we will discuss this in some more detail here.

In most algebraic specification languages, the parameter of a generic specification is supposed or required to present a loose semantics, accepting a large class of possible models. Then, instantiation is the substitution of this generic parameter by one of its "models" (a specification whose models are a sub-class of the models of the parameter, possibly isomorphic, modulo signature translation). This *specialization condition* is needed for an instantiation to be defined. Because there is no loose interpretation of specifications in ELAN and ASF+SDF (only initial semantics), genericity cannot be treated in the same way. Parameterization and instantiation in these languages are just special cases of a more general construct, called *renaming* [12], and there is no requirement in the sense of specialization in the instantiation of a generic parameter. To be able to apply the theory of generalization to ELAN, we will interpret some ELAN specifications differently.

Example 2.1 Consider the specification of lists with an accumulation operation below where variable declarations have been omitted:

```
module List
  imports Psynt                        %% formal parameter
  exports
    sorts List
    context-free syntax
      nil                -> List
      cons (Elem, List)  -> List
      sum (List)         -> Elem
  rules
      [] sum(nil)        => k                          %% sum.nil
      [] sum(cons(x,l)) => bin(x, sum(l))              %% sum.cons
```

[5] To be precise, the current version of the FERUS tool does not cope with many other features of the language, as e.g., mixfix operators and priorities. This is due to the use of an incremental software development methodology, but these will be included in the following and have no influence in the presented work. This is not the case for strategies, which are much more than syntactic sugar and need special study.

```
module Psynt
  exports
    sorts Elem
    context-free syntax
      k                  -> Elem
      bin (Elem, Elem)   -> Elem
```

In a classical model semantics approach, we would expect `Psynt` to have a loose interpretation, accepting as models every algebra with a set of values, a constant and a binary operator, and the module `List` would specify lists of `Elem`, for each of these possible models. On the other hand, with an initial semantics, we only have isomorphic models to the term algebra, with values `k`, `bin(k,k)`, `bin(k,bin(k,k))`, etc. Of course, with this kind of semantics, many usual instantiations of `List` (e.g., `List[Nat]`) would not be valid with the usual instantiation definition (the natural numbers are not isomorphic to this term algebra, and the specialization condition above does not hold). However, it is possible to *rename* `Elem`, `k` and `bin` into `Nat`, `0` and `+`, without any requirement concerning the models of these specifications. It is then possible to simulate classical parameterization and instantiation in the context of ELAN, and this is what we do in FERUS-ELAN. Throughout the rest of the paper, we will then consider that an ELAN imported specification annotated with the `formal parameter` comment is a formal parameter in the classical sense, and that instantiation requires the above specialization condition. Conversely, generalization will aim at substituting regular imports of a specification by an annotated formal parameter.

3 Generalization

The generalization operation is the key for preparing a component to be kept in a library in order to be reused. Then, the instantiation may applied to reuse the generalized (generic) component. The main effect of the generalization operation is to safely substitute input specifications of a specification component by a formal parameter from which the substituted specification is a specialization. Roughly speaking, generalization corresponds to "enlarging" [6] the class of models over which we construct our specification, consequently "enlarging" the class of models of the new component. Conversely, instantiation reduces the class of models of the specification, and so reduces the class of models of the related component.

Let us now briefly present the generalization operation in a rather informal way. A detailed definition can be found in [7,8]. The generalization operator

Co' = `generalize` Co `via` m `with` Param

[6] We use here an intuitive notion of enlarging (generalization) or reducing (instantiation) a class of models, although the differences in the underlying signatures make the comparison not straightforward.

admits three arguments:

(i) A given specification Co which is built over some other imported specifications and may already be generic. In ELAN, it would have the form:

`module` Co `imports` *params-and-imports* `exports SPEC`

where `SPEC` consists of (local) sorts $Sort$(`SPEC`), operators Op(`SPEC`), variables Var(`SPEC`), ad axioms Ax(`SPEC`) which are said to be defined inside the body of Co. Note that local ingredients occurring in `SPEC` may define neither a signature nor a specification due to the existence of *params* and *imports*, where may for instance be defined a sort used in the profile of an operator in Op(`SPEC`), or an operator used in Ax(`SPEC`).

(ii) A specification Param, which is the potential parameter.

(iii) A signature morphism m from objects in Param to imported objects (including parameters, if any) of Co.

In order to be able to define the generalized specification Co', some conditions on Co, m and Param must be satisfied. Hence, the signature morphism m must be injective, so that it can be inverted, and it must correspond to a *specification morphism* from Param to the resulting imported specification *sp* of Co (i.e., the reduct or forgetful functor determined by m applied to models of *sp* must result in models of Param). In addition, we need some technical conditions (*dangling operators condition* and *generalization condition*, see [8] for details) to ensure that we get a syntactically consistent specification component without any references to removed objects. If all of these conditions hold, then we can substitute the previous imports and parameters of Co by the sole Param specification, renaming imported sorts and operators that appear in the local text of Co according to the inverse of m, thus leading to the following specification, using the `[...]` notation as a shortcut for the `%% formal parameter` annotation:

`module` Co' `[` Param `] exports` m_g(`SPEC`)

where the "translation" mapping m_g is a simple renaming of symbols derived from m^{-1}. It is applied throughout the body of the specification being generalized, renaming every occurrence of the generalized sorts and operators.

In the case where the generalization condition does not hold, we can still generalize, but we cannot automatically remove the original imports, obtaining

`module` Co' `[` Param `] imports` *params-and-imports* `exports` m_g(`SPEC`)

Example 3.1 Consider the ELAN specification below of lists of natural numbers with an operation `sum` that gives the total sum of all elements of a given list. It could be generalized into the previous generic list specification `List` through:

```
List = generalize List_Nat via [Elem |-> Nat, k |-> 0, bin |-> +]
                            with Psynt
```

```
module List_Nat
 imports Nat
 exports
   sorts List
   context-free syntax
      nil                        -> List
      cons (Nat, List)           -> List
      sum (List)                 -> Nat
  rules
      [] sum(nil)                => 0            %% sum.nil
      [] sum(cons(x,l))          => x + sum(l)   %% sum.cons

module Nat
 exports
   sorts Nat
   context-free syntax
      0                         -> Nat
      suc(Nat)                  -> Nat
      Nat ``+'' Nat             -> Nat
  rules
      [] x + 0                   => x            %% add.0
      [] x + suc(y)              => suc(x+y)     %% add.suc
```

Note that it may be important to rename locally declared sorts and operators after generalization. For instance, in this example, the sort for lists of naturals could have been named `ListNat`, and in this case, one would like the corresponding sort of the generalized component to be named `List`. This could be done through an extra renaming operation in ELAN.

The generalization operator presented above is basically a tool for, once the desired generalization is defined (by the morphism m and the specification Param), safely executing the corresponding transformation of the component. The most delicate part in the process is however the definition of the desired generalization, i.e., to define m and Param.

To provide some support on this step, we present an algorithm that computes a specification Param for a given set of sorts we want to generalize. This algorithm finds which sorts and operators have to occur in the formal parameter Param, depending on which set of axioms is to included in it. When the set of axioms is empty (resp. non-empty), we are talking about syntactic (resp. semantic) generalization. The general form of generalization is particularly interesting when we want for instance to preserve the proof of a theorem in the generalized component: in that case, the set of axioms to be included corresponds to all non-local axioms involved in the proof.

Obtaining the Generalization Morphism

We present the algorithm computing the arguments needed by the generalization operator. To present it, we use a little more notation:

- Given an axiom e, $operators\_of(e)$ returns the set of operators occurring in e, and by extension to sets of axioms, we define $operators\_of(\{e\}_{e\in E}) = \{operators\_of(e)\}_{e\in E}$.
- Given a operator o, $sorts\_of(o)$ returns the set of sorts occurring in the profile of o, and by extension to sets of operators, we define $sorts\_of(\{o\}_{o\in O}) = \{sorts\_of(o)\}_{o\in O}$. Moreover, given a set of axioms E, we define $sorts\_of(E) = sorts\_of(operators\_of(E))$.

Input: Let GS be a set of imported sorts to generalize in a component Co, and E be a set of axioms in Co we want to preserve by generalization (e.g., the set of axioms used in a rewrite proof of some important property of the original specification). Include in E all trivial axioms $f(\boldsymbol{x}) = f(\boldsymbol{x})$ for each imported operator f used in a local axiom of Co ($Ax(\texttt{SPEC})$).

1 `repeat`
2 $\quad E^g := \{e \in E - Ax(\texttt{SPEC}) | sorts\_of(e) \cap GS \neq \varnothing\}$
3 $\quad Pend := sorts\_of(E^g) - GS$
4 $\quad GS := GS \cup Pend$
5 `until` $Pend = \varnothing$

Output: The generalization is defined for the morphism

$$m_{\texttt{SEM}} : \texttt{PSEM}[\texttt{sorts } s'_1, .., s'_m \texttt{ ops } op'_1, .., op'_j] \rightarrow sp[\texttt{sorts } s_1, .., s_m \texttt{ ops } op_1, .., op_j]$$

where

- sp is the whole imported and/or parameter specification of Co,
- the objects of the specification `PSEM` are the sorts $GS = \{s_1, \ldots, s_m\}$ and operators in $operators\_of(E^g) = \{op_1, \ldots, op_j\}$ with respective new names $\{s'_1, \ldots, s'_m\}$ and $\{op'_1, \ldots, op'_j\}$, and the non-trivial axioms of E^g with the corresponding renamings.

Example 3.2 Consider again the ELAN specification of lists of natural numbers given in Example 3.1. The equation `th1` below is one of its theorems as shows the rewrite proof $p1$.

```
sum(cons(X, nil)) = X                    %% th1
```

$$p1 = \begin{cases} sum(cons(X, nil)) & \rightarrow_{sum.cons} \\ X + sum(nil) & \rightarrow_{sum.nil} \\ X + 0 & \rightarrow_{add.0} \\ X & \end{cases}$$

The generalization of the sort `Nat` with preservation of (axioms in) $p1$ gives:

```
List = generalize List_Nat via [Elem |-> Nat, k |-> 0, bin |-> +]
                            with Psem

module List
 imports Psem
 exports
  ... (same as before, but now with th1 as consequence)

module Psem
  imports Psynt
  rules
      [] bin(X,k) => X                %% bin.k
```

4 FERUS-ELAN

In its current development stage, FERUS-ELAN is mainly intended to support transformation operations over specification components written in the ELAN language. Furthermore, common specification development functionalities (edition, compilation/decompilation, and execution) are provided. The specification transformation operations currently available in FERUS are described below. It should be pointed out that these are meta-operations. Most of the FERUS operations are strongly related to some of CASL structuring constructs, mainly because they are quite standard in the algebraic specification domain. However, in FERUS the idea is to verify applicability conditions for a given transformation operation and carry it out, whenever possible, generating a new specification component. One interesting consequence of this approach is the potential adaptability of the FERUS tool to specification languages different from CASL, as shown by the presented FERUS-ELAN instance, even if they do not include the same sophisticated structuring constructs as CASL.

The available transformation operations are:

rename — allows the substitution of sort and operator identifiers by new ones, provided that these renamings preserve the semantics of the specification component being renamed (isomorphic models).

extend — allows to extend a specification component by the addition of sorts, operations and/or axioms.

reduce — allows to eliminate or hide parts of a specification component.

instantiate — allows the substitution of formal parameters of a generic specification component by actual parameter specifications (specializations of the formal parameter).

generalize — allows the substitution of imported (or extended) specifica-

tions of a specification component by a formal parameter from which the substituted specification is a specialization (there exists a morphism from the added formal parameter to the removed imported or extended specification).

For efficiency reasons, these transformation operations are implemented in a graph representation of the specifications, detailed in Section 4.2.

In addition to the above transformation operations, FERUS-ELAN provides some functionalities to interact with ELAN:

compile — parses and checks a given specification component and generates its representation in a graph-like internal format.

decompile — generates ELAN text from the internal representation of a specification. It is particularly useful to visualize the results of a transformation operation.

execute — calls the ELAN rewrite engine to execute specifications.

These functionalities are implemented using the ATerm exchange format to interact with external tools of ELAN (version 4).

A general picture of the FERUS-ELAN tool is given in the next section.

4.1 General Architecture of FERUS-ELAN

The FERUS-ELAN design is identical to the version dedicated to CASL. This reinforces that the tool modeling is appropriate for the domain of algebraic specification languages. The architectural components of FERUS-ELAN illustrated in figure 4.1 are described as follows :

library implementing the internal format — libelan provides a set of useful functions to create and to handle the data structure that represents the abstract syntax of elan as graphs. In fact this library was first implemented for a model-checker and was also been successfully applied for the FERUS-CASL instance (libcasl). This means that the graph-like data structure is kept the same, only the organization of nodes had to change in order to represent the abstract syntax of each different language.

compiler — its task is to translate specification components written in a sublanguage of ELAN to the FERUS internal format. The compiler uses a set of libraries provided by ELAN to catch the parse trees of each specification component, which is an important change with respect to the FERUS-CASL version that works with abstract syntax trees. Once these parse trees are identified and analyzed, the compiler interacts with libelan in order to create and store the graph nodes that represent the specification in a repository of compiled modules.

library implementing the transformation operations — libferuselan implements the operations like renaming, generalization, among others. Some

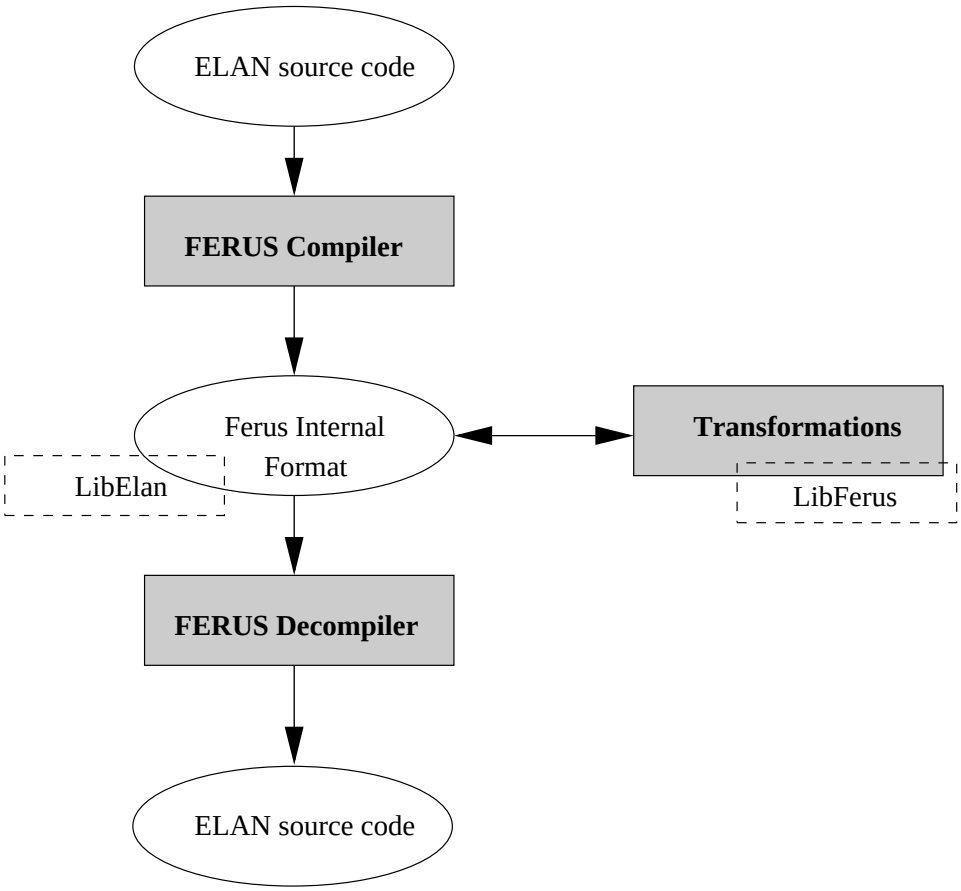


Fig. 4.1. FERUS-ELAN architecture

of the operations could be implemented with the same semantic for both CASL and ELAN, but since parameterization mechanisms differ it was necessary to do a study of how to achieve generalization in ELAN. The goal of libferuselan is to keep implementation details hidden and let the tool's architecture open to be reused by other tool developers.

decompiler — retrieves the textual representation of a component contained in the FERUS-ELAN repository and generates specification text in the ELAN sublanguage dealt by the tool. It is basically the inverse of compilation, also making use of libelan.

graphical user interface — intended to support the user in the process of component reuse. Indeed, the verification of some of the correctness preconditions of transformation operations is not trivial and the user may need some support in the definition of their parameters. Therefore, we developed special-purpose user-interfaces, based on the wizard metaphor [2].

```
/* elanKind - Enumeration structure defining the kind of nodes
 * needed for a complete definition of the specifications.
 */
typedef enum {
  elanModuleDef, //Module definition
  elanImport, //Imported modules
  elanDefSection, //Exports section definition
  elanSortDef, //Sort declaration
  elanOperatorDef, //Operator declaration
  elanVariableDef, //Variable Declaration
  elanAxiomDef,  //Axiom
  elanVariableInst,  //Occurrence of variable
  elanOperation,  //
  elanList //General list
} elanKind;
```

Fig. 4.2. Definition of the node kinds in libelan

4.2 libelan: FERUS-ELAN Internal Format

Both versions of FERUS use graph format to represent the abstract syntax of specifications. This format is more suitable for the transformations than using a maximal subterm sharing format as ATerms [1]. A discussion concerning the pros and cons of each approach can be found in [9].

libelan implements the functionalities needed to handle this format in memory as well as its storage in a repository of compiled specifications. It is important to stress that although it is not the same instance used to represent CASL specifications, the format itself is very similar to the one for libcasl, presented in [9]. Only the set of nodes has been changed, so that it could represent the abstract syntax for ELAN modules.

4.2.1 Implementation details for libelan

This section contains excerpts of the source code (in C) from libelan.

Figure 4.2 contains the definition of the data type `elanKind` that enumerates the different kinds of nodes employed to represent ELAN specifications. For each value of `elanKind`, the library contains a record type definition, as that given in Figure 4.3 for specification definitions.

Note that all the structure definitions are grouped in a single data type `elanNodeRec`. Direct access to these structures is not provided to the user, and only the data type `elanNode`, defined as a pointer to `elanNodeRec`, shall be manipulated by client code, using a set of routines defined in the interface of the library.

The attributes `aKind` and `aToolInfo` and the edge labeled `nRoot` are common to all the different node signatures. `aKind`, obviously, is the kind of the node, `aToolInfo` is a slot where client applications may store specific

```
typedef union elanNodeRec_ {
  ...
  struct {
    elanRef position;           //elanNode position
    void * aToolInfo;           // free slot (tool-specific)
    elanKind aKind;             //elanModuleDef - kind of the node
    elanRef nRoot;              //Root specification
    unsigned aLine;             //Line in the source code
    char * aIdentifier;         //module's Identifier
    elanRef nImports;           /* list of elanImport */
    elanRef nExports;           /* Module's Exports section */
    elanRef nHiddens;           /* Module's Hiddens section */
    elanRef nAxiomDefs;         /* list of AxiomDef */
  } elanModuleDef;  //Module definition
 ...
} elanNodeRec;
```

Fig. 4.3. Structure of a specification definition node (kind `elanModuleDef`).

```
extern elanNode elanMakeModuleDef
(int paLine,
 char * paIdentifier,
 elanNode pnImports,
 elanNode pnExports,
 elanNode pnHiddens,
 elanNode pnAxiomDefs
 );
```

Fig. 4.4. Example of declaration for a constructor.

data as a pointer. The `nRoot` edge links the node to the root of the subgraph of the specification it belongs to.

In the internal state of the library, each node is uniquely referenced by a pair composed of the identifier of the specification it belongs to, and its position in the node table. libelan has internal routines that, given a node reference, returns the actual node and vice-versa.

libelan interface provides initialization routine, constructors, and accessors routines. For instance, Figure 4.4 presents the constructor for specification definition nodes. Additionally, libelan provides file input and output routines, and thus may be used to maintain specification libraries as well as a mean to communicate between tools.

4.3 libferuselan: Transformations

This library was designed to make transformations available to the tool and to allow separation of concerns between the user interface support and the

```
typedef enum {
  elanRenaming,
  elanExtension,
  elanReduction,
  elanInstantiation,
  elanGeneralization
} elanTransformationKind;
```

Fig. 4.5. Enumeration of kinds of transformations that can be applied to ELAN modules.

```
extern int applyMapping (
  elanNode pModule,
  const char * pNewModuleIdentifier,
  elanMapping pMapping
);
```

Fig. 4.6. Transformation application function.

transformation functionalities.

4.3.1 Implementation details for libferuselan

Essentially a transformation generates a new component given some parameters. This can be seen as an abstraction in the following form:

`NewComponent = Operation OldComponent with Parameters`

Hence, for the generalization transformation, `Parameters` consists of a `Param` module and a mapping to describe the signature morphism, as shown in Section 3.

The libferuselan library was intended to implement this kind of abstraction, and to make it transparent to the programmer, whose concerns should be how to build the correct mappings accordingly to each transformation listed in Figure 4.5. Once the mapping is done, all he has to do is to apply the mapping by calling a single function (see Figure 4.6), that makes the interface with the library similar to the form presented above.

5 Conclusion

This paper reports the adaptation of the FERUS tool for the new ELAN language (version 4). This has led to a new version called FERUS-ELAN, which is already able to deal with a significant subset of the ELAN language. Our experiments demonstrate the efficiency of its internal format over which we apply operations for reusability of algebraic specification components. In the future, we will extend the current implementation in order to support the complete ELAN language.

For a specification language like ELAN, it is important to have a good tool

support, possibly by reusing and adapting existing tools developed for related languages. In this direction, the main advance has been obtained with the adaptation of the metaEnvironment initially developed for ASF+SDF [11]. This metaEnvironment is component-based, and the interoperability of components is supported by a Toolbus - a software coordination architecture. As a tool for reuse-driven transformations of ELAN specifications, we envision to connect FERUS-ELAN to the Toolbus and so to integrate our tool to the new ELAN metaEnvironment. For a better integration, we must ease the execution (compilation/interpretation) of ELAN specifications generated by FERUS-ELAN. Obviously, the decompiler can be used to obtain the textual representation of an ELAN specification, but this well-formed specification has to be parsed again. Provided that an abstract representation is used as an exchange format, we could imagine to decompile the safe results generated by FERUS-ELAN and to give them directly to ELAN execution tools, without introducing a non-necessary parsing phase. Thus, the problem of executing specifications generated by FERUS-ELAN illustrates the interest of using a common exchange format for the connection of ELAN-related tools.

References

[1] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software-Practice and Experience*, 30:259–291, 2000.

[2] J. Carroll, R. Mack, and W. Kellogg. Interface metaphors and user interface design. In *Handbook of Human-Computer Interaction*. Elsevier, 1988.

[3] Horatiu Cirstea and Claude Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.

[4] F. Durán. *Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, University of Málaga, 1999.

[5] CoFI group. *The CoFI Algebraic Specification Language*. Available at the CoFI home page: http://www.briks.dk/Projects/CoFI.

[6] N. Martí Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, August 2002.

[7] A. Martins. *La Généralisation : un Outil pour la Réutilisation*. PhD thesis, INPG, March 1995.

[8] A. Martins and C. Ringeissen. Generalizing CASL specification components and preserving rewrite proofs. Technical report, INRIA, 2003. to be published.

[9] Anamaria Martins Moreira, Christophe Ringeissen, David Déharbe, and Gleydson Lima. Manipulating Algebraic Specifications with Term-based and Graph-based Representations. submitted to Journal of Algebraic and Logic Programming, special issue on ATerms, 2003.

[10] T. Mossakowski. Relating CASL with other specification languages: the institution level. *Theoretical Computer Science*, 286(2):367–475, September 2002. Guest editor: J.L. Fiadeiro.

[11] Mark. G. J. van den Brand, Pierre-Etienne Moreau, and Christophe Ringeissen. The ELAN environment: a rewriting logic environment based on ASF+SDF technology. In *Proceedings of the 2st International Workshop on Language Descriptions, Tools and Applications*, volume 65, 2002.

[12] E. Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, Sep. 1997.

[13] M. Wirsing. Algebraic description of reusable software components. Technical Report MIP-8816, Fakultät für Mathematik und Informatik - Universität Passau, 1988.

[14] M. Wirsing. Algebraic specification. In *Handbook of Theoretical Computer Science*, chapter 13. Elsevier Science Publishers B.V., 1990.

On-demand Evaluation by Program Transformation [1]

María Alpuente[† 2] Santiago Escobar[† 3] Salvador Lucas[† 4]

[†] *DSIC, UPV, Camino de Vera s/n, 46022 Valencia, Spain.*

Abstract

Strategy annotations are used in eager programming languages (e.g., OBJ2, OBJ3, CafeOBJ, and Maude) for improving efficiency and/or reducing the risk of nontermination. Syntactically, they are given either as lists of natural numbers or as lists of integers associated to function symbols whose (absolute) values refer to the arguments of the corresponding symbol. A positive index enables the evaluation of an argument whereas a negative index means "evaluation on-demand". Recently, we have introduced a formal description of the operational meaning of such on-demand strategy annotations which improves previous formalizations that were lacking satisfactory computational properties. In this paper, we introduce an automatic, semantics–preserving program transformation which produces a program (without negative annotations) which can be then correctly executed by typical OBJ interpreters. Moreover, to demonstrate the practicality of our ideas, the program transformation has been implemented (in Haskell) and we compare the evaluation of transformed programs with the original ones on a set of representative benchmarks.

1 Introduction

Eager rewriting-based programming languages such as Lisp, OBJ*, CafeOBJ, ELAN, or Maude evaluate expressions by innermost rewriting. Since nontermination is a known problem of innermost reduction, *syntactic annotations* (generally specified as sequences of integers associated to function arguments, called *local strategies*) have been used in OBJ2 [9], OBJ3 [11], CafeOBJ [10], and Maude [6] to improve efficiency and (hopefully) avoid nontermination. Local strategies are used in OBJ programs [5] for guiding

[1] Work partially supported by CICYT TIC2001-2705-C03-01 and MCYT grants HA2001-0059 and HU2001-0019.

[2] Email:alpuente@dsic.upv.es, URL:http://www.dsic.upv.es/users/elp/alpuente.html

[3] Email:sescobar@dsic.upv.es, URL:http://www.dsic.upv.es/users/elp/sescobar.html

[4] Email:slucas@dsic.upv.es, URL:http://www.dsic.upv.es/users/elp/slucas.html

[5] As in [11], by OBJ we mean OBJ2, OBJ3, CafeOBJ, or Maude.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

the *evaluation strategy* (abbr. E-strategy): when considering a function call $f(t_1, \ldots, t_k)$, only the arguments whose indices are present as *positive* integers in the local strategy for f are evaluated (following the specified ordering). If 0 is found, then the evaluation of f is attempted. Unfortunately, this restriction of rewriting can have a negative impact in the ability to compute normal forms. Whenever the user provides no local strategy for a given symbol, the (Maude, OBJ*, CafeOBJ) interpreter automatically assigns a *default* E-strategy. For instance, the default local strategy of Maude associates $(1\ 2 \cdots k\ 0)$ to each k-ary symbol f having no explicit strategy, i.e. all arguments are marked as evaluable.

Example 1.1 Consider the following OBJ program (borrowed from [18]):

```
obj Ex1 is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat [strat (1)] .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat [strat (1)] .
  op from : Nat -> LNat [strat (1 0)] .
  op 2nd  : LNat -> Nat [strat (1 0)] .
  vars X Y : Nat . var Ys : LNat .
  eq 2nd(cons(X,cons(Y,Ys))) = Y .
  eq from(X) = cons(X,from(s(X))) .
endo
```

The evaluation of the term `2nd(from(0))` as performed by a typical OBJ interpreter [6] is:

```
OBJ> red 2nd(from(0)) .
reduce in Ex1 : 2nd(from(0))
rewrites: 1
result Nat: 2nd(cons(0,from(s(0))))
```

This corresponds to the following reduction sequence:

`2nd(from(0))` $\rightarrow$ `2nd(cons(0,from(s(0))))`

The evaluation stops at this point since reductions on the second argument of `cons` are disallowed due to the syntactic annotation.

The handicaps of using only positive annotations regarding correctness and completeness of computations are discussed in [1,2,14,15,19,18]: the problem is that the absence of some indices in the local strategies can have a negative impact in the ability of such strategies to compute normal forms. For instance, a further step is required (*demanded* by the rule of `2nd`) in order to obtain the desired outcome in Example 1.1:

`2nd(cons(0,from(s(0))))` $\rightarrow$ `2nd(cons(0,cons(s(0),from(s(s(0))))))`

At this stage, it is not necessary to perform any reduction on symbol `from`,

[6] We use the SRI's OBJ3 interpreter (version 2.0) available at: `http://www.kindsoftware.com/products/opensource/obj3/OBJ3/`.

since reducing at the root position yields the final value:

`2nd(cons(0,cons(s(0),from(s(s(0)))))` → `s(0)`

In [19,18], *negative* indices are proposed to indicate those arguments that should be evaluated only 'on-demand', where the 'demand' is an attempt to match an argument term with the left-hand side of a rewrite rule [7,11,19]. For instance, in [18] the authors suggest that `(1 -2)` is the "apt" local strategy for `cons` in Example 1.1; i.e. first argument can be evaluated but second argument can be evaluated only "on-demand". The evaluation with strategy `(1 -2)` for `cons` is able to reduce `2nd(from(0))` to `s(0)` without entering in a non-terminating evaluation, whereas evaluation only with positive annotations enters in an infinite derivation or does not provide the associated normal form. It is worthy to note that the calculus is simpler than typical functional lazy rewriting and the appropriate strategy annotations for achieving suitable normal forms can be inferred from the program (see [1,**?**,14,15,19,18]).

On-demand strategy annotations have not been properly implemented to date: even if negative annotations are (syntactically) accepted in current OBJ implementations, namely OBJ3 and Maude, unfortunately they do not have the expected (on-demand) effect over the computations.

Example 1.2 Consider the following OBJ program (similar to Example 1.1 except the strategy annotation for symbol `cons`):

```
obj Ex2 is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat [strat (1)] .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat [strat (1 -2)] .
  op from : Nat -> LNat [strat (1 0)] .
  op 2nd  : LNat -> Nat [strat (1 0)] .
  vars X Y : Nat . var Ys : LNat .
  eq 2nd(cons(X,cons(Y,Ys))) = Y .
  eq from(X) = cons(X,from(s(X))) .
endo
```

The OBJ3 interpreter does not implement negative (on-demand) annotations though does *accept* this program and the evaluation of `2nd(from(0))` surprisingly delivers the very same result as in Example 1.1. That is, the negative annotation is just disregarded by the OBJ3 interpreter (which, in this case, causes loss of completeness). On the other hand, the Maude interpreter does not implement negative (on-demand) annotations though does also accept this program and the evaluation of the same expression diverges. This is because the negative annotation is interpreted by Maude as a positive one thus resulting in non-termination.

On the other hand, CafeOBJ is able to deal with negative annotations using the on-demand evaluation model of [18]. The CafeOBJ interpreter is able to compute the intended value `s(0)` of Example 1.1. However, in [1] we discussed

a number of problems of the on-demand evaluation model of [18,19], as shown in the following example.

Example 1.3 [1] Consider the following OBJ program:

```
obj LENGTH is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat  [strat (1)] .
  op from : Nat -> LNat .
  op length : LNat -> Nat     [strat (0)] .
  op length' : LNat -> Nat    [strat (-1 0)] .
  vars X Y : Nat . var Z : LNat .
  eq from(X) = cons(X,from(s(X))) .
  eq length(nil) = 0 .
  eq length(cons(X,Z)) = s(length'(Z)) .
  eq length'(Z) = length(Z) .
endo
```

The expression `length'(from(0))` is rewritten (in one step) to `length(from(0))`. No evaluation is demanded on the argument of `length'` for enabling this step (the negative annotation `-1` is included for `length'` but the corresponding rule includes a variable at the first argument of `length'`) and no further evaluation on `length(from(0))` should be performed (due to the local strategy `(0)` of `length` which forbids evaluation on any argument of `length`). However, the annotation `-1` of function `length'` is treated in such a way by the operational model of [19,18] that the on-demand evaluation of the expression `length'(from(0))` yields an infinite sequence (see [1] for a more detailed explanation). For instance, CafeOBJ[7] ends with a stack overflow:

```
LENGTH> red length'(from(0)) .
-- reduce in LENGTH : length'(from(0))
Error: Stack overflow (signal 1000)
```

We proposed in [1] a solution to these problems which is based on a suitable extension of the *E*-evaluation strategy of OBJ-like languages (that only considers annotations given as natural numbers) to cope with on-demand strategy annotations. Our strategy incorporates a better treatment of demandness and also enjoys good computational properties; in particular, we show how to use it for computing (head-)normal forms and we prove it is conservative w.r.t. other on-demand strategies: *lazy rewriting* [8] and *on-demand rewriting* [14]. A program transformation for proving termination of the on-demand evaluation strategy was also formalized, which relies on standard techniques. Furthermore, a *direct* implementation of the on-demand

[7] We use the CafeOBJ interpreter (version 1.3.1) available at `http://www.ldl.jaist.ac.jp/Research/CafeOBJ/system.html`.

evaluation strategy of [1] (called OnDemandOBJ) has been developed [8].

In this paper, we show how OBJ programs that use local strategies containing negative annotations (and hence could be correctly executed by using the strategy proposed in [1]) can be (also) executed in the existing OBJ implementations which only admit positive annotations (e.g. Maude). This is done by means of an automatic program transformation which encodes the 'on-demand' strategy instrumented by the negative annotations within new function symbols (and corresponding program rules) that only use positive strategy annotations. Before entering in too technical details, we give an example which illustrates the power of our transformation.

Example 1.4 The program of Example 1.2 is transformed by using our method into the following OBJ program *without* negative annotations.

```
obj Ex3 is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat           [strat (1)] .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat     [strat (0)] .
  op cons_root : Nat LNat -> LNat  [strat (1 0)] .
  op cons_+2 : Nat LNat -> LNat   [strat (2)] .
  op from : Nat -> LNat          [strat (1 0)] .
  op 2nd  : LNat -> Nat          [strat (1 0)] .
  op 2nd_+1  : LNat -> Nat       [strat (1 0)] .
  op quoteNat : Nat -> Nat       [strat (0)] .
  op quoteLNat : LNat -> LNat    [strat (0)] .
  vars X Y : Nat . vars Xs : LNat .
  eq 2nd(cons(X,Xs)) = 2nd_+1(cons_+2(X,Xs)) .
  eq 2nd_+1(cons_+2(X,cons(Y,Xs))) = quoteNat(Y) .
  eq from(X) = quoteLNat(cons(X,from(s(X)))) .
  eq quoteNat(2nd(Xs)) = 2nd(quoteLNat(Xs)) .
  eq quoteNat(2nd_+1(Xs)) = 2nd(Xs) .
  eq quoteNat(s(X)) = s(quoteNat(X)) .
  eq quoteNat(0) = 0 .
  eq quoteLNat(from(X)) = from(quoteNat(X)) .
  eq quoteLNat(cons(X,Xs)) = cons_root(quoteNat(X),Xs) .
  eq quoteLNat(cons_+2(X,Xs)) = cons(X,Xs) .
  eq quoteLNat(nil) = nil .
  eq cons_root(X,Xs) = cons(X,Xs) .
endo
```

where $\mathtt{cons}_{root}$ and $\mathtt{cons}_{+2}$ are new symbols introduced by the transformation which perform the pattern matching in a stepwise manner, and `quoteNat` and `quoteLNat` are auxiliary symbols which help to perform a correct evaluation to head normal forms. Roughly speaking, for each constructor symbol `c` with a negative annotation $-i$, we remove all strategy annotations for `c` and introduce an auxiliary constructor symbol $\mathtt{c}_{+i}$ with positive annotation i. Also, we

[8] Available at `http://www.dsic.upv.es/users/elp/soft.html`.

introduce a defined symbol $\mathtt{c}_{root}$ without the negative annotations but with the positive ones plus 0 and we introduce a new rule which is used to translate the new symbol $\mathtt{c}_{root}$ back to `c`. Then, we add new rules which re–define `f` in terms of `c`, $\mathtt{c}_{root}$, and $\mathtt{c}_{+i}$. Finally, for each program rule $l \to r$, we introduce a symbol `quote` in r (specialized for each sort) and add a number of new rules for symbols `quote` which transform `c` into $\mathtt{c}_{root}$ and help to appropriately (head)-normalize terms.

Now, the evaluation of `2nd(from(0))` using OBJ3 yields:

```
OBJ> red quoteNat(2nd(from(0))) .
reduce in Ex3 : quoteNat(2nd(from(0)))
rewrites: 16
result Nat: s(0)
```

which is the desired result. Note that for evaluating expression e, we only need to call $\mathtt{quote}_\tau(e)$ for the appropriate sort τ of e.

After some preliminaries in Section 2, we recall the on-demand evaluation of [1] in Section 3. Then, Section 4 introduces the program transformation together with completeness and correctness results. In Section 5, we experimentally demonstrate that the program transformation pays off in practice. Section 6 concludes.

2 Preliminaries

Given a set A, $\mathcal{P}(A)$ denotes the set of all subsets of A. Let $R \subseteq A \times A$ be a binary relation on a set A. We denote the reflexive closure of R by $R^=$, its transitive closure by R^+, and its reflexive and transitive closure by R^* [5]. An element $a \in A$ is an R-normal form, if there exists no b such that $a\ R\ b$. We say that b is an R-normal form of a (written $a\ R^!\ b$), if b is an R-normal form and $a\ R^* b$. We say that R is *terminating* iff there is no infinite sequence $a_1\ R\ a_2\ R\ a_3 \cdots$. Throughout the paper, $\mathcal{X}$ denotes a countable set of variables and $\mathcal{F}$ denotes a signature, i.e. a set of function symbols $\{\mathtt{f}, \mathtt{g}, \ldots\}$, each having a fixed arity given by a function $ar : \mathcal{F} \to \mathbb{N}$. We denote the set of terms built from $\mathcal{F}$ and $\mathcal{X}$ by $\mathcal{T}(\mathcal{F}, \mathcal{X})$. A term is said to be linear if it has no multiple occurrences of a single variable. Terms are viewed as labelled trees in the usual way. Let $Subst(\mathcal{T}(\mathcal{F}, \mathcal{X}))$ denote the set of substitutions. We denote by id the "identity" substitution: $id(x) = x$ for all $x \in \mathcal{X}$. Positions $p, q, \ldots$ are represented by chains of positive natural numbers used to address subterms of t. We denote the empty chain by Λ. By $\mathcal{P}os(t)$ we denote the set of positions of a term t. Given a set $S \subseteq \mathcal{F} \cup \mathcal{X}$, $\mathcal{P}os_S(t)$ denotes positions in t where symbols in S occur. When no confusion arises, we denote $\mathcal{P}os_{\{f\}}(t)$ as $\mathcal{P}os_f(t)$ for a symbol $f \in \mathcal{F} \cup \mathcal{X}$. Given positions p, q, we denote its concatenation as $p.q$. Positions are ordered by the standard prefix ordering $\leqslant$. Positions can also be ordered by the lexicographical ordering: $p \leqslant_{lex} q$ iff $p \leqslant q$ or $p = w.i.p'$, $q = w.j.q'$, $i, j \in \mathbb{N}$, and $i < j$. Given a set of positions P, $minimal_{\leqslant}(P)$ is the set of minimal positions of P w.r.t. $\leqslant$.

If p is a position, and Q is a set of positions, $p.Q$ is the set $\{p.q \mid q \in Q\}$. The subterm at position p of t is denoted as $t|_p$, and $t[s]_p$ is the term t with the subterm at position p replaced by s. The symbol labelling the root of t is denoted as $root(t)$.

A rewrite rule is an ordered pair (l, r), written $l \to r$, with $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$ and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. The left-hand side ($lhs$) of the rule is l and r is the right-hand side (rhs). A TRS is a pair $\mathcal{R} = (\mathcal{F}, R)$ where R is a set of rewrite rules. $L(\mathcal{R})$ denotes the set of lhs's of $\mathcal{R}$. A TRS $\mathcal{R}$ is left-linear if for all $l \in L(\mathcal{R})$, l is a linear term. Given $\mathcal{R} = (\mathcal{F}, R)$, we take $\mathcal{F}$ as the disjoint union $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ of symbols $c \in \mathcal{C}$, called *constructors* and symbols $f \in \mathcal{D}$, called *defined functions*, where $\mathcal{D} = \{root(l) \mid l \to r \in R\}$ and $\mathcal{C} = \mathcal{F} - \mathcal{D}$. We say that a rule $l \to r$ defined $f \in \mathcal{D}$ if $root(t) = f$. A TRS $\mathcal{R} = (\mathcal{C} \uplus \mathcal{D}, R)$ is a constructor system (CS) if for all $f(l_1, \ldots, l_k) \in L(\mathcal{R})$, $l_i \in \mathcal{T}(\mathcal{C}, \mathcal{X})$, for $1 \leqslant i \leqslant k$. A term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ rewrites to s (at position p), written $t \xrightarrow{p}_{\mathcal{R}} s$ (or just $t \to s$), if $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$, for some rule $l \to r \in R$, $p \in \mathcal{P}os(t)$ and substitution σ.

A mapping $\mu : \mathcal{F} \to \mathcal{P}(\mathbb{N})$ is a *replacement map* (or $\mathcal{F}$-map) if $\forall f \in \mathcal{F}$, $\mu(f) \subseteq \{1, \ldots, ar(f)\}$ [13]. Let $M_{\mathcal{F}}$ be the set of all $\mathcal{F}$-maps. The ordering $\sqsubseteq$ on $M_{\mathcal{F}}$, the set of all $\mathcal{F}$-maps, is: $\mu \sqsubseteq \mu'$ if for all $f \in \mathcal{F}$, $\mu(f) \subseteq \mu'(f)$. Let $\mu^{can}_{\mathcal{R}}$ be the canonical replacement map, i.e. *the most restrictive replacement map which ensures that the non-variable subterms of the left-hand sides of the rules of $\mathcal{R}$ are replacing*, which is easily obtained from $\mathcal{R}$: $\forall f \in \mathcal{F}$, $i \in \{1, \ldots, ar(f)\}$, $i \in \mu^{can}_{\mathcal{R}}(f)$ iff $\exists l \in L(\mathcal{R}), p \in \mathcal{P}os_{\mathcal{F}}(l), (root(l|_p) = f \wedge p.i \in \mathcal{P}os_{\mathcal{F}}(l))$. Let $CM_{\mathcal{R}} = \{\mu \in M_{\mathcal{F}} \mid \mu^{can}_{\mathcal{R}} \sqsubseteq \mu\}$ be the set of replacement maps which are less or equally restrictive than $\mu^{can}_{\mathcal{R}}$.

3 On-demand evaluation strategy

A local strategy for a k-ary symbol $f \in \mathcal{F}$ is a sequence $\varphi(f)$ of integers taken from $\{-k, \ldots, -1, 0, 1, \ldots, k\}$ which are given in parentheses. Symbols without an explicit local strategy are given a *default* annotation which depends on the considered language. A mapping φ that associates a local strategy $\varphi(f)$ to every $f \in \mathcal{F}$ is called an E-strategy map [17,18]. The E-strategy maps are used to correctly guide the evaluation strategy of OBJ-like languages in order to evaluate expressions. In the following, we recall the on-demand E-evaluation strategy of [1]

Let $\mathbb{L}$ be the set of all lists consisting of integers. We define an (embedding) ordering $\sqsubseteq$ between sequences of integers as: $nil \sqsubseteq L$, $\forall L \in \mathbb{L}$; $(i_1\ i_2\ \cdots\ i_m) \sqsubseteq (j_1\ j_2\ \cdots\ j_n)$ if $i_1 = j_1$ and $(i_2\ \cdots\ i_m) \sqsubseteq (j_2\ \cdots\ j_n)$; or $(i_1\ i_2\ \cdots\ i_m) \sqsubseteq (j_1\ j_2\ \cdots\ j_n)$ if $i_1 \neq j_1$ and $(i_1\ i_2\ \cdots\ i_m) \sqsubseteq (j_2\ \cdots\ j_n)$. An ordering $\sqsubseteq$ between strategy maps is defined: $\varphi \sqsubseteq \varphi'$ if, for all $f \in \mathcal{F}$, $\varphi(f) \sqsubseteq \varphi'(f)$. Roughly speaking, $\varphi \sqsubseteq \varphi'$ if for all $f \in \mathcal{F}$, $\varphi'(f)$ is $\varphi(f)$ except by the introduction of some additional indices. Sometimes, it is interesting to get rid of the ordering on (non-nullary) indices in a given local strategy; for this

reason, given an E-strategy map φ, we introduce the following replacement map $\mu^{\varphi}(f) = \{|i| \mid i \in \varphi(f) \wedge i \neq 0\}$.

Let $\mathbb{L}_n$ be the set of all lists of integers whose absolute value does not exceed $n \in \mathbb{N}$. Given an E-strategy map φ, we use the signature $\mathcal{F}^{\sharp}_{\varphi} = \cup\{f_{L_1|L_2}, \overline{f}_{L_1|L_2} \mid f \in \mathcal{F} \wedge L_1, L_2 \in \mathbb{L}_{ar(f)}.(L_1{+}{+}L_2 \sqsubseteq \varphi(f))\}$ (where the function $+{+}$ defines the concatenation of two sequences of integers) and labelled variables $\mathcal{X}^{\sharp}_{\varphi} = \{x_{nil|nil} \mid x \in \mathcal{X}\}$ for marking ordinary terms $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ as terms in $\mathcal{T}(\mathcal{F}^{\sharp}_{\varphi}, \mathcal{X}^{\sharp}_{\varphi})$. Overlining the root symbol of a subterm means that no evaluation is required for this subterm, and the control goes back to the parent. The list L_2 of $L_1 \mid L_2$ denotes the (partially consumed) local strategy being considered for a given symbol, whereas L_1 is interpreted as a kind of store which records previously considered annotations. The main idea is that annotations move from L_2 to L_1 once they have been processed.

We use $f^{\sharp}$ to denote f or $\overline{f}$, for a symbol $f \in \mathcal{F}$. We define the list of *activable* indices of a labelled symbol $f^{\sharp}_{L_1|L_2}$ as $\mathit{activable}(f^{\sharp}_{L_1|L_2}) = \begin{cases} L_1 \text{ if } L_1 \neq \mathit{nil} \\ L_2 \text{ if } L_1 = \mathit{nil} \end{cases}$. The operator φ is extended to a mapping from $\mathcal{T}(\mathcal{F}, \mathcal{X})$ to $\mathcal{T}(\mathcal{F}^{\sharp}_{\varphi}, \mathcal{X}^{\sharp}_{\varphi})$ as follows:

$$\varphi(t) = \begin{cases} x_{nil|nil} & \text{if } t = x \in \mathcal{X} \\ f_{nil|\varphi(f)}(\varphi(t_1), \ldots, \varphi(t_k)) & \text{if } t = f(t_1, \ldots, t_k) \end{cases}$$

Also, the operator $\mathit{erase} : \mathcal{T}(\mathcal{F}^{\sharp}_{\varphi}, \mathcal{X}^{\sharp}_{\varphi}) \to \mathcal{T}(\mathcal{F}, \mathcal{X})$ removes all extra indices.

Given terms $t, l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, we let $\mathcal{P}os_{\neq}(t, l) = \{p \in \mathcal{P}os_{\mathcal{F}}(t) \cap \mathcal{P}os_{\mathcal{F}}(l) \mid \mathit{root}(l|_p) \neq \mathit{root}(t|_p)\}$. We define the set of demanded positions of t w.r.t. l (a lhs of a rule defining $\mathit{root}(t)$), i.e. the set of (positions of) maximal disagreeing subterms as:

$$DP_l(t) = \begin{cases} \mathit{minimal}_{\leqslant}(\mathcal{P}os_{\neq}(t, l)) & \text{if } \mathit{minimal}_{\leqslant}(\mathcal{P}os_{\neq}(t, l)) \subseteq \mathcal{P}os_{\mathcal{D}}(t) \\ \varnothing & \text{otherwise} \end{cases}$$

Note that the restriction of disagreeing positions to positions with defined symbols (in other words, non-constructor symbols) disables the evaluation of subterms which could never produce a redex (see [1,3,12,16]).

Example 3.1 Let us consider $l_1 =$ `2nd(cons(0,cons(Y,Ys)))` and $l_2 =$ `2nd(cons(s(X),cons(Y,Ys)))`, where $\mathcal{C} = \{\mathtt{cons}, \mathtt{nil}, \mathtt{s}, \mathtt{0}\}$ and $\mathcal{D} = \{\mathtt{2nd}, \mathtt{from}\}$. Let $t_1 =$ `2nd(cons(0,nil))`, we have $DP_{l_1}(t_1) = DP_{l_2}(t_1) = \varnothing$, i.e. no position is demanded by l_1 or l_2 because of a constructor conflict with subterm `nil` at position 1.2. Let $t_2 =$ `2nd(cons(2nd(from(0)),from(0)))`, we have $DP_{l_1}(t_2) = DP_{l_2}(t_2) = \{1.1, 1.2\}$, i.e. positions 1 and 2 are demanded by l_1 and l_2 because both positions are function-rooted. And, let $t_3 =$ `2nd(cons(0,from(0)))`, we have $DP_{l_1}(t_3) = \{1.2\}$ but $DP_{l_2}(t_3) = \varnothing$,

i.e. position 1.2 is demanded by l_1 but not by l_2 because of a constructor conflict with l_2.

We define the set of *positive* positions of a term $s \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ as $\mathcal{P}os_P(s) = \{\Lambda\} \cup \{i.\mathcal{P}os_P(s|_i) \mid i > 0$ and $activable(root(s))$ contains $i\}$ and the set of *activable* positions as $\mathcal{P}os_A(s) = \{\Lambda\} \cup \{i.\mathcal{P}os_A(s|_i) \mid i > 0$ and $activable(root(s))$ contains i or $-i\}$. We also define the set of positions with *empty* annotation list as $\mathcal{P}os_{nil}(s) = \{p \in \mathcal{P}os(s) \mid root(s|_p) = f_{L|nil}\}$. Then, the set of *activable demanded* positions of a term $t \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ w.r.t. l (a lhs of a rule defining $root(erase(t))$) is defined as follows:

$$ADP_l(t) = \begin{cases} DP_l(erase(t)) \cap \mathcal{P}os_A(t) \text{ if } DP_l(erase(t)) \not\subseteq \mathcal{P}os_P(t) \cup \mathcal{P}os_{nil}(t) \\ \varnothing \qquad\qquad\qquad\qquad\qquad \text{otherwise} \end{cases}$$

and the set of activable demanded positions of $t \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$ w.r.t. TRS $\mathcal{R}$ as $ADP_\mathcal{R}(t) = \cup\{ADP_l(t) \mid l \to r \in \mathcal{R} \wedge root(erase(t)) = root(l)\}$. Note that the restriction of activable demanded positions to non-positive and non-empty positions is consistent w.r.t the intended meaning of strategy annotations since positive or empty positions should not be evaluated on-demand (see [1]).

Example 3.2 Let us consider the following term $l = \texttt{2nd(cons(0,cons(Y,Yz)))}$. Let

$$t_1 = \mathtt{2nd}_{(1)|(0)}(\mathtt{cons}_{(1\ 2)|nil}(\mathtt{0}_{nil|nil}, \mathtt{from}_{nil|(1\ 0)}(\mathtt{s}_{nil|(1)}(\mathtt{0}_{nil|nil}))))$$

we have $DP_l(erase(t_1)) = \{1.2\}$ but $ADP_l(t_1) = \varnothing$, i.e. position 1.2 is demanded by l but it is a positive position. Let

$$t_2 = \mathtt{2nd}_{(1)|(0)}(\mathtt{cons}_{(1\ -2)|nil}(\mathtt{0}_{nil|nil}, \mathtt{from}_{nil|nil}(\mathtt{s}_{nil|(1)}(\mathtt{0}_{nil|nil}))))$$

we have $ADP_l(t_2) = \varnothing$, i.e. position 1.2 is still demanded by l but it is rooted by a symbol with an empty annotation list. Let

$$t_3 = \mathtt{2nd}_{(1)|(0)}(\mathtt{cons}_{(1)|nil}(\mathtt{0}_{nil|nil}, \mathtt{from}_{nil|(1\ 0)}(\mathtt{s}_{nil|(1)}(\mathtt{0}_{nil|nil}))))$$

we have $ADP_l(t_3) = \varnothing$, i.e. position 1.2 is again demanded by l but it is not an activable position. Finally, let

$$t_4 = \mathtt{2nd}_{(1)|(0)}(\mathtt{cons}_{(1\ -2)|nil}(\mathtt{0}_{nil|nil}, \mathtt{from}_{nil|(1\ 0)}(\mathtt{s}_{nil|(1)}(\mathtt{0}_{nil|nil}))))$$

we have $ADP_l(t_4) = \{1.2\}$.

Given a term $s \in \mathcal{T}(\mathcal{F}^\sharp_\varphi, \mathcal{X}^\sharp_\varphi)$, the total ordering $\leqslant_s$ between activable positions of s is defined as (1) $\Lambda \leqslant_s p$ for all $p \in \mathcal{P}os_A(s)$; (2) if $i.p, i.q \in \mathcal{P}os_A(s)$ and $p \leqslant_{s|_i} q$, then $i.p \leqslant_s i.q$; and (3) if $i.p, j.q \in \mathcal{P}os_A(s)$, $i \neq j$, and i (or $-i$) appears before j (or $-j$) in $activable(root(s))$, then $i.p \leqslant_s j.q$. The ordering

$\leqslant_s$ allows us to choose a position from the set of all activable demanded positions in s, which is consistent with user's annotations (see $min_{\leqslant_s}$ below). We define the set $OD_{\mathcal{R}}(s)$ of on-demand positions of a term $s \in \mathcal{T}(\mathcal{F}^{\sharp}_{\varphi}, \mathcal{X}^{\sharp}_{\varphi})$ w.r.t. TRS $\mathcal{R}$ as follows:

$$\text{if } ADP_{\mathcal{R}}(s) = \varnothing \text{ then } OD_{\mathcal{R}}(s) = \varnothing \text{ else } OD_{\mathcal{R}}(s) = \{min_{\leqslant_s}(ADP_{\mathcal{R}}(s))\}$$

Example 3.3 Continuing Example 3.2. Let us consider the term

$$t_5 = \mathtt{2nd}_{(1)|(0)}(\mathtt{cons}_{(-1\ -2)|nil}(\mathtt{2nd}_{nil|(10)}(\mathtt{nil}_{nil|nil}), \mathtt{from}_{nil|(1\ 0)}(\mathtt{0}_{nil|nil})))$$

we have $ADP_l(t_5) = \{1.1, 1.2\}$ but $OD_{\{l\}}(t_5) = \{1.1\}$ since annotation `-1` appears before `-2` in the memoizing list for symbol `cons`.

Given a term $t \in \mathcal{T}(\mathcal{F}^{\sharp}_{\varphi}, \mathcal{X}^{\sharp}_{\varphi})$ and position $p \in \mathcal{P}os(t)$, $mark(t, p)$ is the term s with all symbols above p (except the root) marked as non-evaluable, in symbols $\mathcal{P}os(s) = \mathcal{P}os(t)$ and $\forall q \in \mathcal{P}os(t)$, if $\Lambda < q < p$ and $root(t|_q) = f_{L_1|L_2}$, then $root(s|_q) = \overline{f}_{L_1|L_2}$, otherwise $root(s|_q) = root(t|_q)$. This is useful for avoiding recursive definitions of the evaluation strategy (see [1]).

Example 3.4 Continuing Example 3.2. We have that $mark(t_4, 1.2) =$

$$\mathtt{2nd}_{(1)|(0)}(\overline{\mathtt{cons}}_{(1\ -2)|nil}(\mathtt{0}_{nil|nil}, \mathtt{from}_{nil|(1\ 0)}(\mathtt{s}_{nil|(1)}(\mathtt{0}_{nil|nil}))))$$

Given a TRS $\mathcal{R}$ and an E-strategy map φ, we formulate the on-demand strategy via the $eval^{\varphi}_{\mathcal{R}}$ function, which returns the set of terms achievable from a given term thorough the strategy map φ. In the following definition, the symbol @ denotes appending an element at the end of a list.

Definition 3.5 Given a TRS $\mathcal{R} = (\mathcal{F}, R)$ and an arbitrary E-strategy map φ for $\mathcal{F}$, we define: $eval^{\varphi}_{\mathcal{R}}(t) = \{erase(s) \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \mid \langle \varphi(t), \Lambda\rangle \xrightarrow{\sharp}{}^{!}_{\varphi} \langle s, \Lambda\rangle\}$. The binary relation $\xrightarrow{\sharp}_{\varphi}$ on $\mathcal{T}(\mathcal{F}^{\sharp}_{\varphi}, \mathcal{X}^{\sharp}_{\varphi}) \times \mathbb{N}^*_+$ is defined as follows: $\langle t, p\rangle \xrightarrow{\sharp}_{\varphi} \langle s, q\rangle$ if and only if $p \in \mathcal{P}os(t)$ and either

(i) $t|_p = f_{L|nil}(t_1, \ldots, t_k)$, $s = t$ and $p = q.i$ for some i; or

(ii) $t|_p = f_{L_1|i:L_2}(t_1, \ldots, t_k)$, $i > 0$, $q = p.i$, and $s = t[f_{L_1@i|L_2}(t_1, \ldots, t_k)]_p$; or

(iii) $t|_p = f_{L_1|-i:L_2}(t_1, \ldots, t_k)$, $i > 0$, $q = p$, and $s = t[f_{L_1@-i|L_2}(t_1, \ldots, t_k)]_p$; or

(iv) $t|_p = f_{L_1|0:L_2}(t_1, \ldots, t_k) = \sigma(l')$, $erase(l') = l$, $s = t[\sigma(\varphi(r))]_p$ for some $l \to r \in R$ and substitution σ, $q = p$; or

(v) $t|_p = f_{L_1|0:L_2}(t_1, \ldots, t_k)$, $erase(t|_p)$ is not a redex, $OD_{\mathcal{R}}(t|_p) = \varnothing$, $s = t[f_{L_1|L_2}(t_1, \ldots, t_k)]_p$, and $q = p$; or

(vi) $t|_p = f_{L_1|0:L_2}(t_1, \ldots, t_k)$, $erase(t|_p)$ is not a redex, $OD_{\mathcal{R}}(t|_p) = \{p'\}$, $s = t[mark(t|_p, p')]_p$, $q = p.p'$; or

(vii) $t|_p = \overline{f}_{L_1|L_2}(t_1, \ldots, t_k)$, $s = t[f_{L_1|L_2}(t_1, \ldots, t_k)]_p$ and $p = q.i$ for some i.

Case 1 means that no more annotations are provided and the evaluation is completed. In case 2, a positive argument index is provided and the evaluation jumps to the subterm at such argument (note that the index is stored). Case 3 only stores the negative index for further use. Cases 4, 5, and 6 consider the attempt to match the term against the left-hand sides of the rules of the program. Case 4 applies if the considered (unlabelled) subterm is a redex (which is, then, contracted). If the subterm is not a redex, cases 5 and 6 are considered (possibly involving some on-demand evaluation). We use the lists of indices labelling the symbols for recording the concrete positions on which we are able to allow on-demand evaluations; in particular, the first (memoizing) list is crucial for achieving this (see definition of set $OD_{\mathcal{R}}$, which uses function *activable* and the ordering $\leqslant_s$ for indicating the positions demanded by some rule in order to become a redex). Case 5 applies if no demanded evaluation is allowed (or required). Case 6 applies if the on-demanded evaluation of the subterm $t|_{p.p'}$ is required. In this case, the symbols lying on the path from $t|_p$ to $t|_{p.p'}$ (excluding the border ones) are overlined. Then, the evaluation process continues onto $t|_{p.p'}$. Once the evaluation of $t|_{p.p'}$ has finished, the only possibility is the repeated (but possibly empty) application of steps corresponding to the last case 7, which implements the return of the evaluation process back to position p (which originated the on-demand evaluation).

Example 3.6 Following the Example 1.2, the appropriate evaluation sequence of the term `2nd(from(0))` via $eval^{\varphi}_{\mathcal{R}}$ is depicted in Figure 3.1, where the index considered in each step is surrounded by a box. Roughly speaking, the following rewriting sequence

```
2nd(from(0))
  → 2nd(cons(0,from(s(0))))
  → 2nd(cons(0,cons(s(0),from(s(s(0))))))
  → s(0)
```

is performed while managing strategy annotations.

4 The Program Transformation

In the following, we formalize a program transformation which translates OBJ programs with arbitrary indices into OBJ programs with positive indices alone. We first explain the awkward points associated to the evaluation with negative indices in order to discern how to transform these indices into positive ones.

Example 4.1 Consider the following OBJ program which is mainly borrowed from a CafeOBJ program in [19] where we consider negative indices for `cons`:

```
obj Ex3rd is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat       [strat (1)] .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat [strat (1 -2)] .
```

$$
\begin{aligned}
&\langle \mathtt{2nd}_{nil|(\boxed{1}\ 0)}(\mathtt{from}_{nil|(1\ 0)}(\mathtt{0}_{nil|nil})), \Lambda\rangle\\
&\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{2nd}_{(1)|(0)}(\mathtt{from}_{nil|(\boxed{1}\ 0)}(\mathtt{0}_{nil|nil})), 1\rangle\\
&\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{2nd}_{(1)|(0)}(\mathtt{from}_{(1)|(0)}(\mathtt{0}_{\boxed{nil}|nil})), 1.1\rangle\\
&\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{2nd}_{(1)|(0)}(\underline{\mathtt{from}_{(1)|(\boxed{0})}(\mathtt{0}_{nil|nil})}), 1\rangle\\
&\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{2nd}_{(1)|(0)}(\mathtt{cons}_{nil|(\boxed{1}\ -2)}(\mathtt{0}_{nil|nil}, \mathtt{from}_{nil|(1\ 0)}(\mathtt{s}_{nil|(1)}(\mathtt{0}_{nil|nil})))), 1\rangle\\
&\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{2nd}_{(1)|(0)}(\mathtt{cons}_{(1)|(\boxed{-2})}(\mathtt{0}_{nil|nil}, \mathtt{from}_{nil|(1\ 0)}(\mathtt{s}_{nil|(1)}(\mathtt{0}_{nil|nil})))), 1\rangle\\
&\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{2nd}_{(1)|(0)}(\mathtt{cons}_{(1\ -2)|\boxed{nil}}(\mathtt{0}_{nil|nil}, \mathtt{from}_{nil|(1\ 0)}(\mathtt{s}_{nil|(1)}(\mathtt{0}_{nil|nil})))), 1\rangle\\
&\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{2nd}_{(1)|(0)}(\mathtt{cons}_{(1\ \boxed{-2})|nil}(\mathtt{0}_{nil|nil}, \mathtt{from}_{nil|(1\ 0)}(\mathtt{s}_{nil|(1)}(\mathtt{0}_{nil|nil})))), \Lambda\rangle\\
&\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{2nd}_{(1)|(0)}(\overline{\mathtt{cons}}_{(1\ -2)|nil}(\mathtt{0}_{nil|nil}, \mathtt{from}_{nil|(\boxed{1}\ 0)}(\mathtt{s}_{nil|(1)}(\mathtt{0}_{nil|nil})))), 1.2\rangle\\
&\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{2nd}_{(1)|(0)}(\overline{\mathtt{cons}}_{(1\ -2)|nil}(\mathtt{0}_{nil|nil}, \mathtt{from}_{nil|(0)}(\mathtt{s}_{nil|(\boxed{1})}(\mathtt{0}_{nil|nil})))), 1.2.1\rangle\\
&\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{2nd}_{(1)|(0)}(\overline{\mathtt{cons}}_{(1\ -2)|nil}(\mathtt{0}_{nil|nil}, \mathtt{from}_{nil|(0)}(\mathtt{s}_{(1)|nil}(\mathtt{0}_{nil|\boxed{nil}})))), 1.2.1.1\rangle\\
&\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{2nd}_{(1)|(0)}(\overline{\mathtt{cons}}_{(1\ -2)|nil}(\mathtt{0}_{nil|nil}, \mathtt{from}_{nil|(0)}(\mathtt{s}_{\boxed{nil}|nil}(\mathtt{0}_{nil|nil})))), 1.2.1\rangle\\
&\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{2nd}_{(1)|(0)}(\overline{\mathtt{cons}}_{(1\ -2)|nil}(\mathtt{0}_{nil|nil}, \underline{\mathtt{from}_{(1)|(\boxed{0})}(\mathtt{s}_{(1)|nil}(\mathtt{0}_{nil|nil}))})), 1.2\rangle\\
&\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{2nd}_{(1)|(0)}(\overline{\mathtt{cons}}_{(1\ -2)|nil}(\mathtt{0}_{nil|nil}, \mathtt{cons}_{nil|(\boxed{1}\ -2)}(\mathtt{s}_{(1)|nil}(\mathtt{0}_{nil|nil}),\\
&\qquad\quad \mathtt{from}_{nil|(1\ 0)}(\mathtt{s}_{nil|(1)}(\mathtt{s}_{(1)|nil}(\mathtt{0}_{nil|nil})))))), 1.2\rangle\\
&\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{2nd}_{(1)|(0)}(\overline{\mathtt{cons}}_{(1\ -2)|nil}(\mathtt{0}_{nil|nil}, \mathtt{cons}_{(1)|(-2)}(\mathtt{s}_{(1)|\boxed{nil}}(\mathtt{0}_{nil|nil}), \cdots))), 1.2.1\rangle\\
&\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{2nd}_{(1)|(0)}(\overline{\mathtt{cons}}_{(1\ -2)|nil}(\mathtt{0}_{nil|nil}, \mathtt{cons}_{(1)|(\boxed{-2})}(\mathtt{s}_{(1)|nil}(\mathtt{0}_{nil|nil}), \cdots))), 1.2\rangle\\
&\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{2nd}_{(1)|(0)}(\overline{\mathtt{cons}}_{(1\ -2)|nil}(\mathtt{0}_{nil|nil}, \mathtt{cons}_{(1\ -2)|\boxed{nil}}(\mathtt{s}_{(1)|nil}(\mathtt{0}_{nil|nil}), \cdots))), 1.2\rangle\\
&\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{2nd}_{(1)|(0)}(\overline{\mathtt{cons}}_{(1\ -2)|\boxed{nil}}(\mathtt{0}_{nil|nil}, \mathtt{cons}_{(1\ -2)|nil}(\mathtt{s}_{(1)|nil}(\mathtt{0}_{nil|nil}), \cdots))), 1\rangle\\
&\quad \xrightarrow{\sharp}_{\varphi} \langle \underline{\mathtt{2nd}_{(1)|(\boxed{0})}(\mathtt{cons}_{(1\ -2)|nil}(\mathtt{0}_{nil|nil}, \mathtt{cons}_{(1\ -2)|nil}(\mathtt{s}_{(1)|nil}(\mathtt{0}_{nil|nil}), \cdots)))}, \Lambda\rangle\\
&\quad \xrightarrow{\sharp}_{\varphi} \langle \mathtt{s}_{(1)|nil}(\mathtt{0}_{nil|nil}), \Lambda\rangle
\end{aligned}
$$

Fig. 3.1. On-demand evaluation of term `2nd(from(0))`

```
  op from : Nat -> LNat      [strat (1 0)] .
  op 3rd  : LNat -> Nat      [strat (1 0)] .
  vars X Y Z : Nat . var Zs : LNat .
  eq 3rd(cons(X,cons(Y,cons(Z,Zs)))) = Z .
  eq from(X) = cons(X,from(s(X))) .
endo
```

Let us introduce an auxiliary function symbol `inf`, which returns an infinite sequence of symbols `s` (note that the evaluation of the call `inf` is non-terminating since the strategy annotation for the constructor `s` above is `1`):

```
op inf : -> Nat .
eq inf = s(inf) .
```

The on-demand E-evaluation of $t =$ `3rd(cons(0,cons(inf,cons(s(0),nil))))` terminates and returns `s(0)`, since the term `inf` is not under a positive (reducible) position nor is demanded by the rule defining `3rd`.

Let us consider a naïve approach for transforming negative indices into positive indices which duplicates the symbols containing negative indices into new symbols containing the positive counterparts (as in the program transformation showed in [1] for approximating termination). The raw application of such program transformation to the previous example delivers the following rules:

```
eq 3rd(cons(X,Zs)) = 3rd(cons+2(X,Zs)) .
eq 3rd(cons+2(X,cons(Y,Zs))) = 3rd(cons+2(X,cons+2(Y,Zs))) .
eq 3rd(cons+2(X,cons+2(Y,cons(Z,Zs)))) = Z .
```

together with the following definitions for symbols `cons` and $\texttt{cons}_{+2}$:

```
op cons : Nat LNat -> LNat [strat (1)] .
op cons+2 : Nat LNat -> LNat [strat (2)] .
```

However, the evaluation of the previous call t w.r.t. this new program enters in an infinite reduction sequence, since the term `inf` will be under a positive (reducible) position after transforming the leftmost symbol `cons` of t into $\texttt{cons}_{+2}$.

In order to avoid this problem, the solution is to remove all positive annotations of symbol `cons`, i.e. we obtain:

```
op cons : Nat LNat -> LNat [strat (0)] .
op cons+2 : Nat LNat -> LNat [strat (2)] .
```

However, occurrences of symbol `cons` appearing at positive positions of the program rules do not behave correctly now, e.g. `3rd(cons(inf,from(s(0)))` does not have an infinite reduction sequence as in the original program because the subterm `inf` does not appear under a positive position.

Nevertheless, we can define a new symbol $\texttt{cons}_{root}$ which inherits the behavior of the original symbol `cons` (i.e. the positive indices of `cons`), and consistently rename the symbols in the TRS through a special symbol `quote` before evaluating each term (note that `quote` is specialized to sorts); i.e. we finally obtain the program:

```
obj Ex3rdII is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat          [strat (1)] .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat    [strat (0)] .
  op consroot : Nat LNat -> LNat [strat (1 0)] .
  op cons+2 : Nat LNat -> LNat  [strat (2)] .
  op from : Nat -> LNat         [strat (1 0)] .
  op 3rd  : LNat -> Nat         [strat (1 0)] .
  op quoteNat : Nat -> Nat      [strat (0)].
  op quoteLNat : LNat -> LNat   [strat (0)].
  vars X Y Z : Nat . var Zs : LNat .
  eq 3rd(cons(X,Zs)) = 3rd(cons+2(X,Zs)) .
  eq 3rd(cons+2(X,cons(Y,Zs))) = 3rd(cons+2(X,cons+2(Y,Zs))) .
  eq 3rd(cons+2(X,cons+2(Y,cons(Z,Zs)))) = quoteNat(Z) .
```

```
  eq from(X) = quoteLNat(cons(X,from(s(X)))) .
  eq quoteNat(3rd(Zs)) = 3rd(quoteLNat(Zs)) .
  eq quoteNat(s(X)) = s(quoteNat(X)) .
  eq quoteNat(0) = 0 .
  eq quoteLNat(from(X)) = from(quoteNat(X)) .
  eq quoteLNat(cons(X,Zs)) = cons_root(quoteNat(X),Zs) .
  eq quoteLNat(nil) = nil .
  eq cons_root(X,Zs) = cons(X,Zs) .
endo
```

Finally, it is worthy to mention that further problems arise when different parallel on-demand paths exist in a term. Thus, it is necessary to correctly guide the evaluation process using more f_{+i} symbols from the root of the left hand side traversing each on-demand path (it is concretely defined in Section 4.2).

We define the complete transformation for a TRS by two different transformers which tackle the two difficulties described above. That is, the transformation starts by applying the first transformer which introduces symbols f_{root} and specialized symbols `quote` in order to set the stage for the second transformer. Then, the second transformer is applied iteratively, which turns the negative indices into positive ones by introducing symbols f_{+i}. This transformer finally removes all negative annotations together with positive annotations of conflictive symbols (such as symbol `cons` in the previous example).

4.1 Transformation for fixing rules

Let $\mathcal{R} = (\mathcal{F}, R)$ be a TRS, and φ be an E-strategy map. We define the set of positions which are activable (but not reducible) of a term $t \in \mathcal{T}(\mathcal{F}^{\sharp}_{\varphi}, \mathcal{X}^{\sharp}_{\varphi})$ as $\mathcal{P}os_{A-P}(t) = \mathcal{P}os_A(t) - \mathcal{P}os_P(t)$. Given a strategy map φ, φ_+ denotes the result of removing all negative indices from φ.

Let us define the set of symbols of the original TRS at positions activable but not reducible as $\mathcal{F}^{\varphi}_{\mathcal{R}} = \{f \in \mathcal{F} \mid ar(f) > 0 \wedge \varphi_+(f) \neq nil \wedge \exists l \in L(\mathcal{R}) : \mathcal{P}os_f(l) \cap \mathcal{P}os_{A-P}(\varphi(l)) \neq \varnothing\}$. Note that if $\mathcal{R}$ is a CS, then $\mathcal{F}^{\varphi}_{\mathcal{R}} \subseteq \mathcal{C}$. In the following, we define the set of auxiliary rules $Quote_{\mathcal{R}^{\mathbb{I}}, \mathcal{F}^{\varphi}_{\mathcal{R}}}$ to appropriately (head)-normalize terms:

$$Quote_{\mathcal{R}^{\mathbb{I}}, \mathcal{F}^{\varphi}_{\mathcal{R}}} = \bigcup_{f \in \mathcal{F}} \begin{cases} \mathtt{quote}(f(\overline{x})) \to f_{root}(\rho_f(\overline{x})) & \text{if } f \in \mathcal{F}^{\varphi}_{\mathcal{R}} \\ \mathtt{quote}(f(\overline{x})) \to f(\rho_f(\overline{x})) & \text{if } f \notin \mathcal{F}^{\varphi}_{\mathcal{R}} \end{cases}$$

$$\text{where } \rho_f(x_i) = \begin{cases} \mathtt{quote}(x_i) & \text{if } \mathtt{(i)} \sqsubseteq \varphi^{\mathbb{I}}(f) \\ x_i & \text{if } \mathtt{(i)} \not\sqsubseteq \varphi^{\mathbb{I}}(f) \end{cases}$$

We define the *first transformer* for fixing strategy annotations $\mathcal{R}^{\mathbb{I}} =$

$(\mathcal{F}^{\mathbb{I}}, R^{\mathbb{I}})$ and $\varphi^{\mathbb{I}}$ as follows: $\mathcal{F}^{\mathbb{I}} = \mathcal{F} \cup \{f_{root} \mid f \in \mathcal{F}^{\varphi}_{\mathcal{R}}\}$, and

$$R^{\mathbb{I}} = \{f(\bar{t}) \to \mathtt{quote}(r) \mid f(\bar{t}) \to r \in R\} \cup \{f_{root}(\overline{x}) \to f(\overline{x}) \mid f \in \mathcal{F}^{\varphi}_{\mathcal{R}}\} \cup Quote_{\mathcal{R}^{\mathbb{I}}, \mathcal{F}^{\varphi}_{\mathcal{R}}}$$

Also, $\varphi^{\mathbb{I}}(f) = \varphi(f)$ for all $f \in \mathcal{F}$, $\varphi^{\mathbb{I}}(f_{root}) = \varphi(f)$++`(0)` for all $f \in \mathcal{F}^{\varphi}_{\mathcal{R}}$, and $\varphi^{\mathbb{I}}(\mathtt{quote}) = \mathtt{(0)}$.

Example 4.2 Consider the TRS $\mathcal{R}$ and the E-strategy map φ of Example 4.1. The TRS $\mathcal{R}^{\mathbb{I}}$ together with $\varphi^{\mathbb{I}}$ is:

```
obj Ex3rdI is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat           [strat (1)] .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat     [strat (1 -2)] .
  op cons_root : Nat LNat -> LNat  [strat (1 -2 0)] .
  op from : Nat -> LNat          [strat (1 0)] .
  op 3rd  : LNat -> Nat          [strat (1 0)] .
  op quoteNat : Nat -> Nat       [strat (0)] .
  op quoteLNat : LNat -> LNat    [strat (0)] .
  vars X Y Z : Nat . var Zs : LNat .
  eq 3rd(cons(X,cons(Y,cons(Z,Zs)))) = quoteNat(Z) .
  eq from(X) = quoteLNat(cons(X,from(s(X)))) .
  eq quoteNat(3rd(Zs)) = 3rd(quoteLNat(Zs)) .
  eq quoteNat(s(X)) = s(quoteNat(X)) .
  eq quoteNat(0) = 0 .
  eq quoteLNat(from(X)) = from(quoteNat(X)) .
  eq quoteLNat(cons(X,Zs)) = cons_root(quoteNat(X),Zs) .
  eq quoteLNat(nil) = nil .
  eq cons_root(X,Zs) = cons(X,Zs) .
endo
```

4.2 Transformation for eliminating negative indices

We formulate the *transformer* for switching negative indices into positive ones. Let $\mathcal{R} = (\mathcal{F}, R)$ be a TRS, and φ be an E-strategy map. Given $l \in L(\mathcal{R})$, we define the set $\mathcal{I}^{\varphi}(l) = \mathcal{P}os_{A-P}(\varphi(l)) \cap \mathcal{P}os_{\mathcal{F}}(l)$. Assume that $\mathcal{I}^{\varphi}(l) \neq \varnothing$ for a rule $l \to r \in \mathcal{R}$, $p.i = max_{\leqslant_{\varphi(l)}}(\mathcal{I}^{\varphi}(l))$ for $p.i \in \mathcal{P}os(l)$ and $i \in \mathbb{N}$ (note that $\Lambda \notin \mathcal{I}^{\varphi}(l)$ by definition), and $f^{\Lambda}, \ldots, f^{p}$ such that $root(l|_q) = f^q \in \mathcal{F}$ for $q \leqslant p$. Then, the *transformer for eliminating negative indices* $\mathcal{R}^{neg} = (\mathcal{F}^{neg}, R^{neg})$ and φ^{neg} is as follows: $\mathcal{F}^{neg} = \mathcal{F} \cup \{f^{\Lambda}_{+j_{\Lambda}}, \ldots, f^{p}_{+j_p}\}$ where $j_{\Lambda}, \ldots, j_p \in \mathbb{N}$ and $q.j_q \leqslant p.i$ for $q \leqslant p$; and

$$R^{neg} = R - \{l \to r\} \cup \{l[y]_{p.i} \to l'[y]_{p.i}, l' \to r\} \cup \{\mathtt{quote}(f^{q}_{+j_q}(\overline{x})) \to f^{q}(\overline{x})\}$$

where y is a fresh variable and l' is obtained from l such that $\forall q \in \mathcal{P}os(l')$:

$$root(l'|_q) = \begin{cases} f^q_{+j_q} & \text{if } q \leqslant p \\ root(l|_q) & \text{otherwise} \end{cases}$$

We let $\varphi^{neg}(f) = \varphi(f)$ for $f \in \mathcal{F}$ and

$$\varphi^{neg}(f_{+j}) = \begin{cases} \texttt{(j 0)} & \text{if } \texttt{(-j 0)} \sqsubseteq \varphi(f) \vee \texttt{(j 0)} \sqsubseteq \varphi(f) \\ \texttt{(j)} & \text{if } \texttt{(-j 0)} \not\sqsubseteq \varphi(f) \wedge \texttt{(j 0)} \not\sqsubseteq \varphi(f) \end{cases}$$

Example 4.3 Consider the TRS $\mathcal{R} = \mathcal{R}^{\mathbb{I}}$ and the E-strategy map $\varphi = \varphi^{\mathbb{I}}$ in Example 4.2. The application of the transformer, the TRS $\mathcal{R}^{neg}$ together with φ^{neg}, is:

```
obj Ex3rdINeg is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat           [strat (1)] .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat     [strat (1 -2)] .
  op cons_root : Nat LNat -> LNat  [strat (1 -2 0)] .
  op cons_+2 : Nat LNat -> LNat   [strat (2)] .
  op from : Nat -> LNat          [strat (1 0)] .
  op 3rd  : LNat -> Nat          [strat (1 0)] .
  op 3rd_+1  : LNat -> Nat       [strat (1 0)] .
  op quoteNat : Nat -> Nat       [strat (0)] .
  op quoteLNat : LNat -> LNat    [strat (0)] .
  vars X Y Z : Nat . var Zs : LNat .
  eq 3rd(cons(X,cons(Y,Zs))) = 3rd_+1(cons_+2(X,cons_+2(Y,Zs))) .
  eq 3rd_+1(cons_+2(X,cons_+2(Y,cons(Z,Zs)))) = quoteNat(Z) .
  eq from(X) = quoteLNat(cons(X,from(s(X)))) .
  eq quoteNat(3rd(Zs)) = 3rd(quoteLNat(Zs)) .
  eq quoteNat(3rd_+1(Zs)) = 3rd(Zs) .
  eq quoteNat(s(X)) = s(quoteNat(X)) .
  eq quoteNat(0) = 0 .
  eq quoteLNat(from(X)) = from(quoteNat(X)) .
  eq quoteLNat(cons(X,Zs)) = cons_root(quoteNat(X),Zs) .
  eq quoteLNat(cons_+2(X,Zs)) = cons(X,Zs) .
  eq quoteLNat(nil) = nil .
  eq cons_root(X,Zs) = cons(X,Zs) .
endo
```

The second transformation process starts from $\mathcal{R}^{\mathbb{I}}$ and $\varphi^{\mathbb{I}}$ and applies as many transformation steps $\mathcal{R}^{neg}$ and φ^{neg} for removing negative indices as necessary to obtain $\mathcal{R}' = (\mathcal{F}', R')$ and φ' such that no negative index is necessary, i.e. $\mathcal{I}^{\varphi'}(l) = \varnothing$ for all $l \in L(\mathcal{R}')$.

The final TRS $\mathcal{R}^{\mathbb{III}} = (\mathcal{F}^{\mathbb{III}}, R^{\mathbb{III}})$ and $\varphi^{\mathbb{III}}$ is obtained as $\mathcal{F}^{\mathbb{III}} = \mathcal{F}'$, $R^{\mathbb{I}} = R'$, and $\varphi^{\mathbb{III}}(f) = \varphi'_+(f)$ for $f \in \mathcal{F}' - \mathcal{F}^{\varphi}_{\mathcal{R}}$ and $\varphi^{\mathbb{III}}(f) = nil$ for $f \in \mathcal{F}^{\varphi}_{\mathcal{R}}$.

Example 4.4 Continuing with Example 4.3. The final TRS $\mathcal{R}^{\mathbb{III}}$ together with $\varphi^{\mathbb{III}}$ is:

```
obj Ex3rdII is
  sorts Nat LNat .
  op 0    : -> Nat .
  op s    : Nat -> Nat           [strat (1)] .
  op nil  : -> LNat .
  op cons : Nat LNat -> LNat     [strat (0)] .
  op cons_root : Nat LNat -> LNat  [strat (1 0)] .
  op cons_+2 : Nat LNat -> LNat   [strat (2)] .
  op from : Nat -> LNat          [strat (1 0)] .
  op 3rd  : LNat -> Nat          [strat (1 0)] .
  op 3rd_+1  : LNat -> Nat       [strat (1 0)] .
  op quoteNat : Nat -> Nat       [strat (0)] .
  op quoteLNat : LNat -> LNat    [strat (0)] .
  vars X Y Z : Nat . var Zs : LNat .
  eq 3rd(cons(X,Zs)) = 3rd_+1(cons_+2(X,Zs))) .
  eq 3rd_+1(cons_+2(X,cons(Y,Zs))) = 3rd_+1(cons_+2(X,cons_+2(Y,Zs))) .
  eq 3rd_+1(cons_+2(X,cons_+2(Y,cons(Z,Zs)))) = quoteNat(Z) .
  eq from(X) = quoteLNat(cons(X,from(s(X)))) .
  eq quoteNat(3rd(Zs)) = 3rd(quoteLNat(Zs)) .
  eq quoteNat(3rd_+1(Zs)) = 3rd(Zs) .
  eq quoteNat(s(X)) = s(quoteNat(X)) .
  eq quoteNat(0) = 0 .
  eq quoteLNat(from(X)) = from(quoteNat(X)) .
  eq quoteLNat(cons(X,Zs)) = cons_root(quoteNat(X),Zs) .
  eq quoteLNat(cons_+2(X,Zs)) = cons(X,Zs) .
  eq quoteLNat(nil) = nil .
  eq cons_root(X,Zs) = cons(X,Zs) .
endo
```

We would like to emphasize that the transformation defined in this paper is able to deal with the general case where more than one negative annotation exist for the same function symbol and different demandness paths are possible; which are just ordered using order between negative annotations and managed by using symbols f_{+i} wich traverse the path from the root. This is not illustrated in our main example due to space restrictions, though we give some hints in the following example.

Example 4.5 Consider the following program rule:

```
min(s(X),0,s(0),s(s(Y))) = 0
```

with the strategy map $\varphi(\texttt{min}) = \texttt{(-3 -2 -1 -4 0)}$, $\varphi(\texttt{s}) = \texttt{(-1)}$, and $\varphi(\texttt{0}) = nil$. The transformation of this rule produces (without considering symbol `quote`):

```
min(X,Z,W,Y) = min_+3(X,Z,W,Y)
min_+3(X,Z,s(W),Y) = min_+3(X,Z,s_+1(W),Y)
min_+3(X,Z,s_+1(0),Y) = min_+2(X,Z,s(0),Y)
min_+2(X,0,s(0),Y) = min_+1(X,0,s(0),Y)
```

$\mathtt{min}_{+1}(\mathtt{s(X)},\mathtt{0},\mathtt{s(0)},\mathtt{Y}) = \mathtt{min}_{+4}(\mathtt{s(X)},\mathtt{0},\mathtt{s(0)},\mathtt{Y})$
$\mathtt{min}_{+4}(\mathtt{s(X)},\mathtt{0},\mathtt{s(0)},\mathtt{s(Y)}) = \mathtt{min}_{+4}(\mathtt{s(X)},\mathtt{0},\mathtt{s(0)},\mathtt{s}_{+1}(\mathtt{Y}))$
$\mathtt{min}_{+4}(\mathtt{s(X)},\mathtt{0},\mathtt{s(0)},\mathtt{s}_{+1}(\mathtt{s(Y)})) = \mathtt{0}$

and the strategy map $\varphi^{\mathrm{III}}(\mathtt{min}) = \mathtt{(0)}$, $\varphi^{\mathrm{III}}(\mathtt{min}_{+1}) = \mathtt{(1\ 0)}$, $\varphi^{\mathrm{III}}(\mathtt{min}_{+2}) = \mathtt{(2\ 0)}$, $\varphi^{\mathrm{III}}(\mathtt{min}_{+3}) = \mathtt{(3\ 0)}$, $\varphi^{\mathrm{III}}(\mathtt{min}_{+4}) = \mathtt{(4\ 0)}$, $\varphi^{\mathrm{III}}(\mathtt{s}) = nil$, $\varphi^{\mathrm{III}}(\mathtt{s}_{+1}) = \mathtt{(1)}$, and $\varphi^{\mathrm{III}}(\mathtt{0}) = nil$. This transformed program reproduces the ordering between the different on-demand paths of the left-hand side.

4.3 Properties

In the following, we establish the main results of the program transformation. We define a *standard* E-strategy map φ as an E-strategy map where no index different to 0 appears at the right of any index 0 in the annotations sequence. Given a strategy map φ for $\mathcal{F}$, we say that a TRS $\mathcal{R} = (\mathcal{F}, R)$ is φ-terminating if, for all $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, there is no infinite $\xrightarrow{\sharp}_{\varphi}$-rewrite sequence starting from $\langle \varphi(t), \Lambda \rangle$. Note that Examples 1.2, 1.3, and 4.1 can be proved φ-terminating using the technique developed in [1]. A defined function $f \in \mathcal{D}$ is *completely defined* if it does not occur in any ground term in normal form, i.e. functions are reducible on all ground terms (of appropriate sort). A TRS $\mathcal{R}$ is *completely defined* (or CD) if each defined function of the signature is completely defined. In the following theorem we prove completeness of the transformation, i.e. that the transformation preserves normal forms.

Theorem 4.6 (Completeness) *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a CS and φ be a standard E-strategy map such that $\mathcal{R}$ is either (a) φ-terminating or (b) CD. Let $\mathcal{R}^{\mathrm{III}} = (\mathcal{F}^{\mathrm{III}}, R^{\mathrm{III}})$ and φ^{III}. For all $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $s \in \mathcal{T}(\mathcal{C}, \mathcal{X})$, if $s \in eval^{\varphi}_{\mathcal{R}}(t)$, then $s \in eval^{\varphi^{\mathrm{III}}}_{\mathcal{R}^{\mathrm{III}}}(t)$.*

Correctness of the transformation is also proved without any condition on termination of the TRS.

Theorem 4.7 (Correctness) *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a CS and φ be a standard E-strategy map. Let $\mathcal{R}^{\mathrm{III}} = (\mathcal{F}^{\mathrm{III}}, R^{\mathrm{III}})$ and φ^{III}. For all $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $s \in \mathcal{T}(\mathcal{C}, \mathcal{X})$, if $s \in eval^{\varphi^{\mathrm{III}}}_{\mathcal{R}^{\mathrm{III}}}(t)$, then $s \in eval^{\varphi}_{\mathcal{R}}(t)$.*

Finally, termination is preserved by the transformation.

Theorem 4.8 (Termination) *Let $\mathcal{R} = (\mathcal{F}, R) = (\mathcal{C} \uplus \mathcal{D}, R)$ be a CS and φ be a standard E-strategy map. $\mathcal{R}$ is φ-terminating iff $\mathcal{R}^{\mathrm{III}}$ is φ^{III}-terminating.*

5 Experiments

A prototype implementation of the transformation proposed in this paper has been developed in Haskell (using ghc 5.04.1). The system is publicly available at

`http://www.dsic.upv.es/users/elp/soft.html`

Tables 5.1, and 5.2, show the runtimes [9] in milliseconds and the number of evaluation steps of the benchmarks for the different OBJ-family systems. The OnDemandOBJ interpreter is the on-demand prototype interpreter of the on-demand evaluation of [1]. CafeOBJ [10] is developed in Lisp at the Japan Advanced Inst. of Science and Technology (JAIST); OBJ3 [11], also written in LISP, is maintained by the University of California at San Diego; Maude [12] is developed in C++ and maintained by the Computer Science Lab at SRI International. OBJ3 and Maude provide only computations with positive annotations whereas CafeOBJ provides computations with negative annotations as well, using the on-demand evaluation of [19,18]. OnDemandOBJ computes with negative annotations using the on-demand evaluation of [1]. Note that CafeOBJ and OBJ3 implement sharing of variables whereas Maude and OnDemandOBJ do not; thus, the number of evaluation steps in Table 5.2 is pairwise equivalent: CafeOBJ and OBJ3 in one hand and Maude and OnDemandOBJ in the other hand. Also, since Maude is implemented in C++, typical execution times are nearly `0` milliseconds. Finally, mark *overflow* indicates that execution raised a memory overflow and normal form was not achieved; whereas mark *unavailable* indicates that the program can not be executed in such OBJ implementation.

The benchmark `pi` codifies the well-known infinite serie expansion to approximate number π: $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$ and uses negative annotations to obtain a terminating and complete example, which can not be obtained using only positive annotations. Termination of the program can be formally proved using the technique of [1]. Also, by using the results in [2], we can guarantee that every expression such as `pi(`n`)` for some n of sort `Nat` produces (as expected) a completely evaluated expression of sort `LRecip`. The benchmark `pi_noneg` consists of the application of the program transformation described in this paper to `pi`. Table 5.1 compares the evaluation of expression `pi(square(square(3)))` using `pi` and `pi_noneg`. It witnesses that negative annotations are extremely useful in practice and that the program transformation enables the execution of negatively annotated programs in all OBJ implementations. On the other hand, Table 5.1 also evidences that the implementation of the on-demand evaluation strategy in other systems is really interesting.

On the other hand, Table 5.2 illustrates the interest of using negative annotations to improve the behavior of programs: The benchmark `msquare_eager` codifies the functions `square`, `minus`, `times`, and `plus` over natural numbers using only positive annotations. Every k-ary symbol f is given a strategy

[9] The average of 10 executions measured in a Pentium III machine running redhat 7.2.
[10] Available at `http://www.ldl.jaist.ac.jp/Research/CafeOBJ/system.html`.
[11] Available at `http://www.kindsoftware.com/products/opensource/obj3/OBJ3/`.
[12] Available at `http://maude.cs.uiuc.edu/current/system/`.

ms./rewrites	`pi`	`pi_noneg`
OnDemandOBJ	25/364	45/689
CafeOBJ	30/364	55/689
OBJ3	*unavailable*	75/689
Maude	*unavailable*	0/689

Table 5.1
Execution of call `pi(square(square(3)))`

$(1\ 2\ \cdots k\ 0)$ (this corresponds to *default* strategies in Maude). Note that the program is terminating as a TRS (i.e., without any annotation). The benchmark `msquare_apt` is similar to `msquare_eager`, but *canonical* positive strategies are provided: the i-th argument of a symbol f is annotated if there is an occurrence of f in the left-hand side of a rule having a non-variable i-th argument; otherwise, the argument is not annotated (see [4]). The benchmark `msquare_neg` is similar to `msquare_eager`, though *canonical* arbitrary strategies are provided: now (from left-to-right), the i-th argument of a defined symbol f is annotated if all occurrences of f in the left-hand side of the rules contain a non-variable i-th argument; if all occurrences of f in the left-hand side of the rules have a variable i-th argument, then the argument is not annotated; in any other case, annotation $-i$ is given to f (see [4]). The benchmark `msquare_noneg` represents the application of the program transformation to `msquare_neg`. Then, for instance, program `msquare_neg` runs in less time and requires a smaller number of rewrite steps than `msquare_eager` or `msquare_apt`, which do not include negative annotations. Moreover, note that the program transformation is also very useful since execution of the expression `minus(0,square(square(5)))` is improved. These experimental results, together with the OBJ source programs, are available at

`http://www.dsic.upv.es/users/elp/ondemandOBJ/experiments`

6 Conclusions

The paper presents a contribution to the extension of evaluation strategies for functional languages of the OBJ family with evaluation on demand, thereby introducing a flavour of laziness into such languages. Our proposal is based on a program transformation for OBJ programs which achieves correctness and works well in current OBJ interpreters. The main technical results of this work are as follows:

- The proposed transformation preserves the termination of the original program which uses (positive and) negative annotations. That is, if the original program terminates under the evaluation strategy of [1], then the transformed program terminates also.

ms./rewrites	msquare_eager	msquare_apt	msquare_neg	msquare_noneg
OnDemandOBJ	33/ 715 40/ 914	62/ 1640 78/ 1992	0/ 1 80/ 1992	0/ 1 250/ 5243
CafeOBJ	40/ 715 50/ 914	50/ 715 60/ 914	0/ 1 60/ 914	0/ 1 180/ 2601
OBJ3	20/ 715 30/ 914	*overflow* *overflow*	*unavailable* *unavailable*	0/ 1 *overflow*
Maude	0/ 715 0/ 914	0/ 1640 3/ 1992	*unavailable* *unavailable*	0/ 1 5/ 5243

Table 5.2
Execution of terms `minus(0,square(square(5)))` and `minus(square(square(5)),square(square(3)))`

- Correct and completeness of the transformation holds w.r.t. the semantics of strategy annotations given in [1]. That is, the semantics of input expressions in the original program (under the on-demand E-strategy of [1]) and in the transformed program (under the E-strategy) do coincide.

Moreover, our transformation is useful both for

(i) making possible the use of arbitrary strategy annotations in languages that (syntactically) allow them but that still do not provide the necessary operational support (e.g., OBJ3).

(ii) providing a notion of negative strategy annotation (somewhat laziness) for languages that does not allow them (e.g., Maude).

and hence we think that our work contributes to foster the use of OBJ in programming. As future work, we plan to formally determine the overhead associated to the evaluation in the transformed program.

References

[1] M. Alpuente, S. Escobar, B. Gramlich, and S. Lucas. Improving on-demand strategy annotations. In M. Baaz and A. Voronkov, editors, *Proc. 9th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'02)*, volume 2514 of *Lecture Notes in Computer Science*, pages 1–18, Tbilisi, Georgia, 2002. Springer-Verlag, Berlin.

[2] M. Alpuente, S. Escobar, and S. Lucas. Correct and complete (positive) strategy annotations for OBJ. In *Electronic Notes in Theoretical Computer Science*, volume 71. Elsevier Sciences Publisher, 2002.

[3] M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of the ACM SIGPLAN Conf. on Partial*

Evaluation and Semantics-Based Program Manipulation, PEPM'97, volume 32, 12 of *ACM Sigplan Notices*, pages 151–162, New York, 1997. ACM Press.

[4] S. Antoy and S. Lucas. Demandness in rewriting and narrowing. In *Proc. of 11th International Workshop on Functional and (Constraint) Logic Programming WFLP'02*, volume 76 of *Electronic Notes in Theoretical Computer Science*. Elsevier Sciences, 2002.

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

[6] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In *Electronic Notes in Theoretical Computer Science*, volume 4, pages 65–89. Elsevier Sciences Publisher, 1996.

[7] S. Eker. Term rewriting with operator evaluation strategies. In *Electronic Notes in Theoretical Computer Science*, volume 15. Elsevier Sciences Publisher, 2000.

[8] W. Fokkink, J. Kamperman, and P. Walters. Lazy rewriting on eager machinery. *ACM Transactions on Programming Languages and Systems*, 22(1):45–86, 2000.

[9] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proc. of 12th Annual ACM Symp. on Principles of Programming Languages*, pages 52–66. ACM Press, New York, 1985.

[10] K. Futatsugi and A. Nakagawa. An overview of CAFE specification environment – an algebraic approach for creating, verifying, and maintaining formal specification over networks –. In *1st International Conference on Formal Engineering Methods*, 1997.

[11] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000.

[12] R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of PLILP'93*, pages 184–200. Springer LNCS 714, 1993.

[13] S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1):1–61, 1998.

[14] S. Lucas. Termination of on-demand rewriting and termination of OBJ programs. In *Proc. of 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP'01*, pages 82–93. ACM Press, New York, 2001.

[15] S. Lucas. Lazy rewriting and context-sensitive rewriting. In *Electronic Notes in Theoretical Computer Science*, volume 64. Elsevier Sciences Publisher, 2002.

[16] J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224, 1992.

[17] T. Nagaya. *Reduction Strategies for Term Rewriting Systems*. PhD thesis, School of Information Science, Japan Advanced Institute of Science and Technology, March 1999.

[18] M. Nakamura and K. Ogata. The evaluation strategy for head normal form with and without on-demand flags. In *Electronic Notes in Theoretical Computer Science*, volume 36. Elsevier Sciences Publisher, 2001.

[19] K. Ogata and K. Futatsugi. Operational semantics of rewriting with the on-demand evaluation strategy. In *Proc. of 2000 International Symposium on Applied Computing, SAC'00*, pages 756–763. ACM Press, New York, 2000.

Domain-Specific Optimisation with User-Defined Rules in CodeBoost [1]

Otto Skrove Bagge [2] Magne Haveraaen [3]

Department of Informatics
University of Bergen
PO Box 7800, N-5020 Bergen, Norway

Abstract

The use of domain-specific optimisations can significantly enhance the performance of high-level programs. However, current programming languages have poor support for specifying such optimisations. In this paper we introduce user-defined rules in CodeBoost. CodeBoost is a tool for source-to-source transformation of C++ programs. With CodeBoost, domain-specific optimisations can be specified as rewrite rules in C++-like syntax, within the C++ program, together with the classes where they apply. We also illustrate the effectiveness of user-defined rules and domain-specific optimisation on a real-world example application.

1 Introduction

High-level, abstract programs communicate clearly the intent of the programmer. Much of this information is lost upon translation to a lower-level representation—either by the programmer or by the compiler. Traditional optimisation techniques focus much attention on various forms of analysis to rediscover this lost information. *Domain-specific optimisations* take advantage of domain knowledge for optimisations purposes. This allows the optimiser to optimise high-level code directly, with little need for analysis. This greatly simplifies implementation, and also makes is possible to implement otherwise infeasible special-purpose optimisations. However, this places additional demands on the programmer, who must formalise the knowledge of the abstractions used, and make this information available to the optimiser.

[1] This investigation has been carried out with the support of Chr. Michelsen Research AS, the Research Council of Norway (NFR), and by a grant of computing resources from NFR's Supercomputer Committee.

[2] Email: otto@ii.uib.no

[3] Email: magne@ii.uib.no

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

This paper describes user-defined rules in CodeBoost, which provide one way of formalising domain knowledge for use in optimisations.

CodeBoost [1,2,9] is a source-to-source transformation tool for C++ [20,13]. It has been developed as part of the SAGA project, to serve as a domain-specific optimiser for Sophus [12,16]. Sophus is a C++ library providing high-level abstractions for implementing partial differential equation solvers.

CodeBoost provides a framework for implementing domain-specific optimisations and other transformations. It contains a parser, a semantic analyser, a transformation library, and a pretty-printer. CodeBoost is implemented in the Stratego program transformation language [23,24], and new transformations can be implemented either in Stratego, or as user-defined rules embedded in a C++ program.

User-defined rules allow C++ programmers to specify domain-specific optimisations and other transformations within the C++ program. Rules are specified in C++-like syntax, eliminating the need to learn a separate transformation language. Because CodeBoost performs semantic analysis also on user-defined rules, it is trivial to perform semantic as well as syntactic matching.

The purpose of this paper is to give an overview of the capabilities and use of user-defined rules in CodeBoost. We will first introduce the basic ideas and concepts of user-defined rules. We then discuss some slightly more advanced concepts: list matching, conditions and generic rules. Then, we illustrate the practical use of CodeBoost on a real-world Sophus application, with a few simple, yet effective rules. Finally, we outline our future plans, discuss related work and offer some concluding remarks.

2 User-defined rules

User-defined rules in CodeBoost allow C++ programmers to specify rewriting of expressions. The rules are written in C++-like syntax, and can be placed within the C++ program, close to relevant functions and classes. User-defined rules are primarily intended as an aid in writing domain-specific optimisations, but they are useful also for other kinds of transformations which require systematic rewriting of expressions. Examples include replacing all uses of unsafe indexing operators in a library with boundary checking versions, or instrumenting code with extra checks to verify the validity of optimisation rules.

Since rules are specified in C++-like syntax, there is no need to learn a whole new programming language, nor is knowledge of compiler design or the inner workings of an optimiser required to write user-defined optimisation rules.

2.1 Semantic matching

In CodeBoost, all program transformations are performed on an abstract syntax tree (AST) representation of the program. To support advanced optimisations, the AST is annotated with semantic information. For instance, all variables are annotated with types, and all function calls are annotated with the unique function signature corresponding to the called function. This makes it easy to distinguish between overloaded functions. [4]

The ability to do matching on syntactic as well as semantic information is important when implementing domain-specific optimisations for a language which allows overloading. For instance, suppose that part of the domain knowledge for a numerical library is that

$$\texttt{pow}(\texttt{x}, 2) \equiv \texttt{x} * \texttt{x}. \tag{2.1}$$

There may be other unrelated `pow` functions for which this is not true. Any optimisation rule taking advantage of (2.1) must not interfere with such unrelated functions.

However, matching on semantic information is not as simple as it may seem. In Stratego, rewrite rule patterns can be specified in abstract syntax and/or in concrete syntax (i.e. the syntax of the object language). Unfortunately, concrete syntax cannot be used with CodeBoost, since this requires an SDF2 grammar [22] for C++, which is not available (yet). When rule patterns are written in abstract syntax, particular functions can be identified by specifying the full signature of the functions. In the case of CodeBoost (and C++), signatures are typically several lines long, and make rules difficult to read, write and maintain. As an alternative to making signatures part of the pattern, semantic constraints may be specified as some form of condition. This is not entirely unattractive, especially when signatures and such can be specified in concrete syntax. Even so, explicitly specifying function signatures and variable types in rule can be tedious and error-prone.

In CodeBoost, we have solved this problem by allowing rules to be specified within C++ programs, with patterns written in C++ syntax. When the program is run through CodeBoost, the patterns are converted to AST form and subjected to normal semantic analysis, together with the rest of the program. Hence, they are automatically annotated with the correct signatures. After semantic analysis, the rules are extracted and made available for use as rewrite rules in CodeBoost modules. Two transformation modules, `apply-user-rules` and `simplify` are provided with CodeBoost to integrate the application of user-defined transformations with the rest of the transformation pipeline.

[4] C++ allows overloading of both functions and operators. The distinction between functions and operators is often of little importance, and we will usually refer to both as simply 'functions'.

2.2 Rule syntax

All user-defined rules are placed inside one or more C++ functions named `rules()`. CodeBoost will recognise the function signature and interpret the body as a list of rules. The `rules()` definitions can occur anywhere a normal function can, and contain any number of rules. All names used in the rules should be in scope and accessible. Rules are best placed either close to the functions to which they apply (i.e. inside the same class), or in a separate header file.

A user-defined rule consists of a *rule name* , a *match pattern*, a *replacement pattern* and an optional *condition* (explained in Section 3.2). Fig. 2.1 shows the syntax for user-defined rules. A typical rule looks like this:

```
void rules()
{
  int x;
  simplify: pow(x, 2) = x * x, isTrivial(x);
}
```

where '`simplify`' is the rule name, '`pow(x, 2)`' is the match pattern, '`x * x`' is the replacement pattern, and '`isTrivial(x)`' is a condition. During semantic analysis the `pow` and `*` calls will be annotated with signatures, ensuring that the correct versions are used for both matching and replacement. The hypothetical condition `isTrivial` would check that `x` is a trivial expression, so that the rule does not cause work duplication.

The rule name implicitly controls the application strategy. For instance, the `simplify` transformation module will apply `simplify` rules according to a predefined topdown-repeat strategy. This is explained in more detail in Section 2.3.

When a rule is applied to an expression, the AST structure of the match pattern will be compared to the AST structure of the expression. If the pattern matches, the condition is checked; if it is true, rule application succeeds and the matched expression is replaced with the replacement pattern.

Locally declared variables (such as the `x` above) are treated as meta-variables. A meta-variable in the match pattern matches all expressions, and is bound to the first expression it matches. Subsequent occurrences of a bound meta-variable in the match or replacement pattern are replaced by its value. For example, the match pattern `f(x, x)`, where `x` is a meta-variable, will match a call to `f` where both arguments are identical.

rules ::= `void rules() {` (*vardecl* | *rule*)* `}`

rule ::= *rule-name*`:` *expr* `=` *expr* [`,` *condition*]`;`

Fig. 2.1. Syntax for user-defined rules

2.3 Rule sets and application strategies

All rules with the same name make up a *rule set*. When a rule set is applied, all rules in the set are tried until one of them succeeds. If no rule succeeds, the rule set application fails. A rule set is said to terminate if it can be applied repeatedly without getting stuck in an endless loop (i.e. it is guaranteed to fail sooner or later).

Rule set application is ultimately controlled by transformation modules written in Stratego. CodeBoost makes rule sets available to Stratego programs as ordinary Stratego rules. User-defined rules are typically used with one of two transformation modules, `simplify` and `apply-user-rules`, whose only purpose is to apply rule sets. With these two modules, users can write and apply their own rewrite rules without any knowledge of Stratego.

The rule sets applied by the transformation modules all have predefined names. However, the user is free to apply other rule sets in rule conditions (see Section 3.2).

The `simplify` rule set is used for simplification rules. It is assumed that the right-hand side of `simplify` rules is somehow "better" or preferable to the left-hand side, and that the rule set will terminate [5]. The transformation module `simplify` will apply the `simplify` rule set repeatedly (until it terminates) using top-down traversal.

The module `apply-user-rules` applies four rule sets. It uses a down-up traversal, applying `topdown` rules on the way down, and `bottomup` on the way up. For repeated application (as in `simplify`) there is `topdown_r` and `bottomup_r`.

In addition to these general purpose rule sets, some rule sets are use for specific purposes. For instance, some transformation modules use the `assoc` and `commute` rule sets to implement matching modulo associativity or commutativity.

3 Advanced Concepts

3.1 List Matching

When working with functions that accept a variable number of arguments, it is useful to be able to match all or part of the argument list with a single meta-variable. List matching makes this possible; the match pattern `_list_(x)` will match zero or more arguments in an argument list. When substituting meta-variables in the replacement pattern, the value of `_list_(x)` will be integrated into the argument list it appears in. For example, consider the rule

```
int f(int, int, ...);
int g(...);
```

[5] So, if the user specifies a rule that does not terminate, rule application may not terminate.

```
void rules()
{
  int a, b, c;
  topdown:  f(a, b, _list_(c)) = g(a, _list_(c), b);
}
```

If this rule is applied to `f(1, 2, 3, 4, 5)`, it will match with `_list_(c)` $\mapsto$ `3, 4, 5`, and the result of the rewriting will be `g(1, 3, 4, 5, 2)`.

During overload resolution, `_list_(x)` will appear as a single argument with the same type as `x`. Note, however, that *matching* is untyped, so `_list_(x)` can match arguments of any type—the type information is only used for overload resolution. This is consistent with the semantics of C++.

It is possible to use list matching to match the beginning or middle of argument lists:

```
topdown: f(_list_(a), g(b), _list_(c))
         = fg(b, _list_(a), _list_(c));
```

If this rule is applied to `f(1, 2, 3, g(4), 5)`, it will match, with `_list_(a)` $\mapsto$ `1, 2, 3`; `b` $\mapsto$ `4`; `_list_(c)` $\mapsto$ `5`; and the result will be `fg(4, 1, 2, 3, 5)`.

CodeBoost only looks ahead one element while deciding when to stop, and no backtracking is done. If the list match pattern is followed by something that matches anything (such as an unbound meta-variable), the list match will never match anything.

3.2 Rules with Conditions

Conditions are useful for specifying that a rule should only be applied in certain cases. Because conditions can have (local) side-effects, such as binding new meta-variables, they can also be useful for specifying more advanced rewriting. Conditions follow the rule body after a comma, and are written in function call notation. The comma should be read as 'where'. For example:

```
X x, y;
simplify: (x + y) = x, isZero(y);
```

In this rule, `isZero` is a hypothetical built-in condition, which uses dataflow information to determine whether `y` has a zero value. The rule will only be applied if the condition evaluates successfully.

Built-in conditions are available for doing advanced checking and rewriting. So far, the development of the condition language (and user-defined rules in general) has been driven by the needs of particular optimisations rules. Therefore, only a few, special built-ins have been added so far. Conditions can be combined using several primitives, including `&&` (and), `||` (or) and `not`.

As an example, the following rule is a better version of the `pow`-simplification rule from Section 2.2:

```
int x, t;
simplify: pow(x, 2) = t * t, t = tmp(x);
```

The rule uses the `tmp` built-in to make a temporary variable to hold the value of `x`, so that `x` is not evaluated twice. A declaration for the temporary is inserted before the containing statement, and `tmp` yields the name of the temporary, for use in the replacement.

In addition to the built-ins, all user-defined rules can be used as conditions. The following rule will switch the arguments of the plus operator so that multiplications are on the left side. If there already is a multiplication on the left side, nothing will be done.

```
topdown: z + (x * y) = (x * y) + z,
                    not(isMultExpr(z));
isMultExpr: (x * y) = true;
```

3.3 Generic Rules

So far, we have only written rules that work on *specific* functions and operators. Many rules have a basic structure that is independent of the particular functions to which they apply. For example, the structure of a commutativity rule is the same no matter which operators or types are involved (i.e., switch the two arguments of a binary function or operator). Writing the same rule over and over again quickly becomes tedious. It is better to write a generic rule, and then specify which operators are commutative—particularly since other rules may also depend on the commutativity property.

A *generic rule* can be used to specify rewrite rules independent of names, types and signatures. For example, if `f` is declared locally as a function pointer, the generic rule

```
topdown: f(x, y) = f(g(x), g(y));
```

will rewrite *any* call to *any* binary function or operator. The `f` will be bound to both the name and signature of the matched function, and can be used to construct new calls to the same function in the right-hand side of the rule.

As another example, the following fragment gives a generic commutativity rule:

```
void rules()
{
  T (*f)(T, T);  // declare f as function pointer
  T x, y;

  commute: f(x, y) = f(y, x),
                  commutative(!f(x, y));
}
```

The `!` primitive is used to build a literal expression for use in rule application.

The following rules specify that the operators * and + on integers are commutative:

```
void rules()
{
  int a, b;

  commutative: (a + b) = true;
  commutative: (a * b) = true;
}
```

Obviously, for simple cases like the above example, generic rules are not a huge benefit. However, they are beneficial for more complicated and non-obvious rules. Furthermore, generic rules provide the benefit of abstraction; the rule and its requirements are separated from, and independent of, the concrete functions that fulfil its requirements.

4 Application: Sophus

4.1 Sophus background

We will use the Sophus software library [10,12] as our test bed for the transformations. Sophus is developed for the numerical solution of partial differential equations. Such programs are often termed *solvers*. Solvers are often used to check whether a model of the real world, e.g., a geophysical model of a 1km^3 section of the earth, matches the real world which it is supposed to model. A check is performed by letting the solver simulate some measurement on the model, e.g., how seismic waves propagate, and compare the simulated results with real world measurements. If the comparison shows a problem with the model, the model is adjusted and the solver is run once more on the new model. Comparing results and adjusting the model is a creative process which requires human ingenuity and may easily take specialists a full working day. A simulation cycle is such a sequence of activities (run solver, investigate results, adjust parameters for the next run of the solver). In this field runtime efficiency is important. A solver may easily take several hours or days to reach a solution. Bringing the solution time down from say, 25 hours to 15 hours makes it possible to complete a simulation cycle every day rather than every 2–3 days.

Unlike most numerical libraries, Sophus is based on coordinate free notions, which are very high-level abstractions. This makes Sophus well suited for source level optimisations. We will focus on the SeisMod application, which simulates seismic waves (elastic waves) in geophysical models of sections of the Earth. A key component of SeisMod is the Mesh abstraction, akin to the notion of an array. Meshes are used to store data about the waves and the physical properties of the material where they propagate. If we are doing a coarse simulation on a 1km^3 section of the Earth, we only need to store data

with 5m resolution, which requires $200^3 = 8\,000\,000$ units of information. The information needed for a simulation may easily amount to 20 Mesh variables, each with 8 000 000 floating point numbers.

It is not always necessary to simulate a full 3D section. In many cases a 2D cross section of the model will give sufficient information. In our example this reduces storage (and computation requirements) for a 2D version by a factor of 200, to less than one percent of that of the 3D version.

4.2 Mesh, MeshPoint and MeshShape abstractions

A consequence of working with both 2D and 3D models is that we sometimes need to index meshes using three indices (3D case), other times we will need only two indices (2D case) for the same Mesh variables. In Sophus this has been solved by introducing an index type abstraction called a MeshPoint, which represents a list of indices with a given shape. The MeshShape abstraction tells how many indices are used, and the extent of each of them. In the example, the MeshShape would be $\langle 200, 200, 200\rangle$ and $\langle 200, 200\rangle$ in the 3D and 2D cases, respectively. So a Mesh M is indexed by $M[P]$, where the MeshPoint P may be a list of 2 or 3 indices. Actually, there are also cases where we need only one index, in others we need four or more.

For this paper, the mesh abstractions contain a few interesting operations:

`float operator[](const Mesh &, const MeshPoint &)` which is the Mesh indexing function, returning a floating point value for every appropriate MeshPoint.

`int getlex(const MeshPoint &)` which decodes a MeshPoint index to the unique lexicographic ordering it has within its shape.

`MeshShape getshape(const MeshPoint &)` which extracts the shape of a MeshPoint.

`int getsize(const MeshShape &)` which computes the number of distinct MeshPoints with the given shape.

`MeshPoint setlex(const MeshShape &, const int &)` which encodes an integer as the unique MeshPoint with that lexicographic ordering within the given MeshShape.

Obviously the operations `getlex` and `setlex` are inverses, in the sense that `getlex(setlex(S,i))==i` and that `setlex(getshape(P),getlex(P))==P` for MeshShape `S`, integer `i` and MeshPoint `P`. These properties of the MeshPoint abstraction may easily be encoded as user-defined simplification rules within the MeshPoint class:

```
void rules()
{ int i;
  MeshShape S;
  MeshPoint P;
```

```
  simplify: getlex(setlex(S,i)) = i;
  simplify: setlex(getshape(P),getlex(P)) = P;
}
```

A typical implementation of a Mesh class will represent the data of a Mesh with MeshShape `S` as an array `float A[getsize(S)];`. Placing the data in the Mesh in a lexicographic ordering, we can implement the indexing operation using the `getlex` function.

```
float operator[](const Mesh & M, const MeshPoint & P)
{ return M.A[getlex(P)];
}
```

A call `M[P]` computes the lexicographic position of the argument `P`, and then uses this integer to access the data in the array `A`.

4.3 Example: using user defined rules for simplification

When using high level abstractions, it is often the case that the functions have to decode its data structure, do some computation, and encode the result within the data structure of the abstraction. Sometimes a succession of calls to such functions will result in encode-decode sequences which eliminate each other (at run-time). Often this code cannot be removed from the program text without breaking the abstraction barriers. One example is the MeshPoint abstraction, with a computation `MeshPoint P=setlex(S,i); float r=M[P];` for a Mesh `M` of MeshShape `S`. As a computation this first encodes the integer `i` as a MeshPoint, then decodes the MeshPoint back to the integer value of `i` in order to access the data of the Mesh. But we cannot eliminate this extra computation without revealing the data structure and algorithms we are using for implementing the Mesh. Further, bypassing the MeshPoint abstraction will force problems in other parts of the code, such as the partial derivative functions (not further discussed here), where the list nature of the MeshPoint index is important.

In the SeisMod solver much of the computation is centred around traversing the Mesh data and performing operations on the elements of several meshes for every MeshPoint index. Since the MeshShape `S` for the MeshPoints is not known until runtime, traversal of the Mesh `M1` ... `Mn` data structures are typically done using a simple loop and the `setlex` function.

```
for (int i=0; i<getsize(S); i++)
{ P = setlex(S,i);
  ... M1[P] ... Mn[P] ...;
}
```

For every iteration of the loop, we encode i to a MeshPoint, then every indexing operation decodes the MeshPoint back to the same integer.

A code transformation tool like CodeBoost is allowed to bypass the

abstraction borders of the code, e.g., by inlining, and may thus exploit properties of the underlying implementation. In our case the user-defined rule `getlex(setlex(S,i)) = i` allows CodeBoost to get rid of the unneeded computations. The effect of this optimisation on the SeisMod code is startling. Table 4.1 summarises timings for the isotropic version of SeisMod, baseline version 1.21, with the accompanying small and large regression testing data sets.

	Not optimised	*Basic*	*Simplified*	*Speedup*
Small	827.0s	629.9s	110.5s	5.7
Large	25435s	19028s	3996s	4.8

Table 4.1
Timings for SeisMod for small and large data sets. *Not optimised* is results without any CodeBoost optimisation. *Basic* includes some Sophus-specific optimisations. *Simplified* adds inlining of indexing and `getlex`/`setlex` simplification. *Speedup* is speedup factor of *Simplified* relative to *Basic*.

5 Future Work

Although user-defined rules have already been used with great success in optimising Sophus applications, they are still an experimental feature under active development. Currently, our work is focused on the interaction between abstraction and optimisation, and the ordering of transformations. Many domain-specific optimisations are applicable only at a particular abstraction level. Inlining can be used to lower abstraction levels, but it must be done at precisely the right time, or optimisation opportunities may be lost.

Transformation rules can be classified into three different kinds: *Simplifying rules*, which perform actual optimisation, *Implementing rules*, which perform inlining, and rules such as commutativity, which do not change the abstraction level, and do not result in better code, but may enable other optimisations. A simple strategy exploiting this classification is: 1) Apply as many simplifications as possible; 2) Try other rules, to see if they enable further simplification; 3) Apply one level of inlining, go back to 1 if this succeeds. But even with strategies such this, there is still the question of choosing which individual rule to apply.

Sometimes, more than one rule matches a given expression. In this case, CodeBoost will currently pick the first rule. It would probably be better to pick the most specific rule (i.e. the one with fewest meta-variables), or the one that gives the "best" results. Ordering rules by specificity is easy, but deciding which rule is "best" will probably require input from the user. Furthermore, a rule that may seem "worse" could still enable other transformations resulting in overall better results. We have not studied this in detail yet; it may be

feasible to try several possibilities in parallel and then pick the best result.

6 Related Work

The concept of conditional rewrite rules is pretty standard in transformation languages such as Stratego [23,24], ASF+SDF [8], ELAN [3], TXL [7], TAMPR [4,5] and others. Unlike these systems, our user-defined rules facility is not intended as a general-purpose transformation language; we restrict ourselves to transforming C++, within the CodeBoost framework. Apart from this, the main difference compared to general-purpose systems is the embedding of rules within the program being transformed. This allows users to place domain-specific optimisation rules inside the modules to which they apply, and also allows the use of normal semantic analysis to generate semantic constraints.

Embedding optimisation rules in program text is not a new idea. We were first inspired by the rewrite rules in the Glasgow Haskell Compiler [14]. Our implementation is more advanced, however, as it supports both side conditions and multiple strategies.

User-definable strategies is available in Stratego and ELAN, and gives the programmer precise control over the application of rules. We offer only a selection of predefined strategies; however, when user-defined rules are used from Stratego, they can of course be applied using arbitrary strategies.

User-defined rules bear some resemblance to macro processing, as in Lisp [19] and the rather primitive C/C++ preprocessor [15,20]. However, macros are commonly used for *syntactic rewriting*, without taking into account context or semantics (this is actually a feature). The ability to match on semantic information is an important part of user-defined rules in CodeBoost.

CodeBoost, and user-defined rules, is most appropriately compared with similar frameworks for C++ and other languages. Simplicissimus [18] is a transformation system for C++ that allows user-specified conditional rewriting of expressions. Pattern matching is done with *expression templates* [21], and *traits* [17] are used to specify function properties, which can be used in rule conditions. Simplicissimus allows some degree of strategic control over the application of rules, through its *arbiters* (deciding which rules are applied), *stages* (deciding when they are applied), and *directors* (deciding how the program is traversed).

Simplicissimus is implemented as a compiler plug-in (currently using GCC), and uses the compiler's template processing for matching. The functionality is similar to that of user-defined rules. The main drawback with Simplicissimus is that the syntax for specifying rules is verbose and complex; rules that are written in one line using user-defined rules would need half a page of code in Simplicissimus. Our generic rules are inspired by Simplicissimus, and the ideas of arbiters, directors and stages will probably be useful in the further development of CodeBoost.

OpenC++ [6] provides a *meta-object protocol* for C++. A meta-object protocol is an object-oriented interface for specifying language extensions and transformations. As such, OpenC++ has many of the same capabilities as CodeBoost [6], and can be used to implement domain-specific optimisations, as well as other transformations, in an object-oriented fashion. OpenC++ is not restricted to working with expressions; it can also be used for language extensions, and generating functions and classes.

The Broadway compiler [11] allows library designers to annotate their C++ libraries with semantic information that will be used in high-level optimisations. The compiler is focused on the numerical domain, where for instance a high-level program may require the solution of a linear system of equations. There exist many variations of such equations solvers, and the more that is known about the properties of the linear system, the more efficient the solver algorithm. The Broadway compiler tries to automatically select optimal solvers by using annotations from the solver library to track properties of the data in the high-level program.

For numerical software the TAMPR program transformation system [4,5] has been used with remarkable success. A typical use is the derivation of efficient Fortran code from high-level functional specifications. Other uses include deriving an efficient implementation of TAMPR itself (which is specified in pure functional Lisp), reverse engineering, and solving the Year 2000 problem.

7 Conclusion

User-defined rewrite rules provide a convenient way of specifying domain-specific optimisations. Rules are written in C++-like syntax within the program text, and allow conditional rewriting of expressions. Conditions can be used both to control when and if a rule is applied, and to perform advanced rewriting by calling other rules or calling built-ins written in Stratego. Additionally, list matching is available to conveniently manipulate the argument lists of functions accepting a variable number of arguments.

Specifying rules within the program text allows us to subject rule patterns to semantic analysis together with the rest of the program. Automatically adding semantic information to rule patterns enables simple and easy specification of domain-specific optimisations, without the need for hand-written semantic constraints.

Rules are organised into rule sets with different application strategies. There are general rule sets for top-down and bottom-up application, and rule sets with specific semantics, suitable for applying commutativity laws, for instance.

User-defined rules are currently implemented as an interpreter written

[6] In fact, the parser of OpenC++ is used in CodeBoost's frontend

in Stratego. The interpreter makes user-defined rules available to Stratego programs. A predefined rule set can be used as any other Stratego rule, and CodeBoost can easily be extended to support new rule sets.

User-defined rules are only a small part of the CodeBoost framework, and more complex, generic optimisations are best implemented as separate transformation modules written in Stratego. Still, we have promising results from using user-defined rules for optimising Sophus, and we expect that most new Sophus optimisations will be written using user-defined rules.

References

[1] Otto Skrove Bagge. CodeBoost: A framework for transforming C++ programs. Master's thesis, University of Bergen, P.O.Box 7800, N-5020 Bergen, Norway, March 2003.

[2] Otto Skrove Bagge, Magne Haveraaen, and Eelco Visser. CodeBoost: A framework for the transformation of C++ programs. Technical Report UU-CS-2001-32, Institute of Information and Computing Sciences, Utrecht University, 2001.

[3] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. In C. and H. Kirchner, editors, *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications*, Pont-A-Mousson, France, September 1998. Elsevier Science.

[4] James M. Boyle. Abstract programming and program transformation—An approach to reusing programs. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability*, volume 1, pages 361–413. ACM Press, 1989.

[5] James M. Boyle, T.J. Harmer, and V.L. Winter. The TAMPR program transformation system: Simplifying the development of numerical software. In Erlend Arge, Are Magnus Bruaset, and Hans Petter Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 353–372. Birkhäuser, Boston, 1997.

[6] Shigeru Chiba. A metaobject protocol for C++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 285–299. ACM, October 1995.

[7] James R. Cordy, Charles D. Halpern, and Eric Promislow. TXL: A rapid prototyping system for programming language dialects. In *Proceedings of the IEEE 1988 International Conference on Computer Languages*, pages 280–285, October 1988.

[8] Arie van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

[9] T.B. Dinesh, Magne Haveraaen, and Jan Heering. An algebraic programming style for numerical software and its optimization. *Scientific Programming*, 8(4):247–259, 2000.

[10] Helmer André Friis, Tor Arne Johansen, Magne Haveraaen, Hans Munthe-Kaas, and Asmund Drottning. Use of coordinate free numerics in elastic wave simulation. *Applied Numerical Mathematics*, 39(2):151–171, 2001.

[11] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Proceedings of DSL'99: The Second Conference on Domain-Specific Languages*, Austin, Texas, USA, 1999. The USENIX Association.

[12] Magne Haveraaen, Helmer André Friis, and Tor Arne Johansen. Formal software engineering for computational modelling. *Nordic Journal of Computing*, 6(3):241–270, 1999.

[13] ISO/IEC JTC1 SC 22. *ISO/IEC 14882: Programming languages — C++*, 1998.

[14] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In Ralf Hinze, editor, *2001 Haskell Workshop*, Firenze, Italy, September 2001.

[15] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[16] Hans Munthe-Kaas and Magne Haveraaen. Coordinate free numerics — closing the gap between 'pure' and 'applied' mathematics? *Zeitschrift für Angewandte Mathematik und Mechanik*, 76, supplement 1:487–488, 1996.

[17] Nathan C Myers. Traits: a new and usefule template technique. *C++ Report*, June 1995.

[18] Sibylle Schupp, Douglas P. Gregor, David R. Musser, and Shin-Ming Liu. Semantic and behavioral library transformations. *Information and Software Technology*, 44(13):797–810, October 2002.

[19] Guy L. Steele, Jr. *Common LISP: The Language*. Digital Press, second edition, 1990.

[20] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, USA, third edition, 1997.

[21] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[22] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[23] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[24] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.

Design and implementation of the L+C modeling language

Radoslaw Karwowski and Przemyslaw Prusinkiewicz
Department of Computer Science
University of Calgary

Abstract

L-systems are parallel grammars that provide a theoretical foundation for a class of programs used in procedural image synthesis and simulation of plant development. In particular, the formalism of L-systems guides the construction of declarative languages for specifying input to these programs. We outline key factors that have motivated the development of L-system-based languages in the past, and introduce a new language, L+C, that addresses the shortcomings of its predecessors. We also describe the implementation of L+C, in which an existing language, C++, was extended with constructs specific to L-systems. This implementation methodology made it possible to develop a powerful modeling system in a relatively short period of time.

1. Background

L-systems were conceived as a rule-based formalism for reasoning on developing multicellular organisms that form linear or branching filaments [Lindenmayer, 1968]. Soon after their introduction, L-systems also began to be used as a foundation of visual modeling and simulation programs, and computer languages for specifying the models [Baker and Herman, 1972]. Subsequently they also found applications in the generation of fractals [Szilard and Quinton, 1979; Prusinkiewicz, 1986] and geometric modeling [Prusinkiewicz et al., 2003]. A common factor uniting these diverse applications is the treatment of structure and form as a result of development. A historical perspective of the L-system-based software and its practical applications is presented in [Prusinkiewicz, 1997].

According to the L-system approach, a developing structure is represented by a string of symbols over a predefined alphabet V. These symbols represent different components of the structure (e.g., points and lines of a geometric figure, cells of a bacterium, apices and internodes of a plant). The process of development is characterized in a declarative manner using a set of productions over the alphabet V. During the simulation of development, these productions are applied in parallel steps to all symbols of the string, thus capturing the development in discrete time slices.

Lindenmayer [1971] observed that L-system productions can be specified using standard notation of formal language theory. In the simplest, context-free case, productions have the form:

$$predecessor \rightarrow successor$$

where *predecessor* is a letter of alphabet V and *successor* is a (possibly empty) word over V. For example, the division of a cell A into cells B and C can be written as $A \rightarrow BC$. In the context-sensitive case, productions are often written as

$$lc < predecessor > rc \rightarrow successor\ ,$$

where symbols < and > separate the strict predecessor from the left context *lc* and the right context *rc* [Prusinkiewicz and Hanan,1989]. Both contexts are words over *V*. For example, the production pair:

$$Y < A > O \rightarrow LYS$$
$$O < A > Y \rightarrow SYL$$

describes asymmetric division of a mother cell *A* into a short daughter cell *S* and long daughter cell *L*, separated by a cell wall *Y*. The sequence of these cells in the filament is guided by the state of the walls that delimit the mother cell, which may be young (*Y*) or old (*O*). Obviously, a complete description of the filament's development would also require productions that characterize the growth of cells and walls over time.

Early L-system-based programming languages closely followed the above notation [Baker and Herman, 1982; Prusinkiewicz and Hanan, 1989]. The needs for expressing increasingly complex models led, however, to the addition of constructs found in other programming languages. A pivotal moment in this evolution was the introduction of parametric L-systems [Prusinkiewicz and Hanan, 1990; Hanan, 1992] and related constructs [Chien and Jurgensen, 1992], which associated numerical attributes to L-system symbols, similar to those found in attribute grammars [Knuth, 1968]. This created a need for calculating new parameter values (in the production successor) on the basis of old ones (found in the predecessor and its context). According to the original definition of parametric L-systems [Prusinkiewicz and Hanan, 1990; Hanan, 1992], these calculations were specified as arithmetic operations on the argument parameters, e.g.

$$A(x) < B(y) > C(z) \rightarrow D(x+y)\ E(y+z)\ .$$

In modeling practice, however, entire procedures soon became needed to calculate new parameter values. Recognizing this need, Hanan [1992] introduced the following syntax for L-system productions:

$$lc < predecessor > rc\ \{\alpha\} : cond\ \{\beta\} \rightarrow successor\ .$$

Here α and β are C-like compound statements, and *cond* is a logical expression that guards production application. A production is applied in stages. First, it is determined whether production predecessor *pred*, surrounded by the left context *lc* and the right context *rc*, matches the given symbol in the string. If this is the case, the compound statement α is executed, and condition *cond* is evaluated. If the result of this evaluation is non-zero ('true'), the second compound statement β is executed. On this basis, parameters values in the production successor are then determined, and the successor is inserted into the resulting string. For example, the following is a valid production:

$$A(x) < B(y) > C(z)\ \{r = x*x + y*y + z*z;\} : r > 2\ \{t = x+y+z;\} \rightarrow D(t)\ E(2*t).$$

At the top level, an L-system with productions in the above form operates in a declarative fashion, by rewriting elements of a string according to their type, context, and the associated parameters. Within each production, however, calculations are performed sequentially, using constructs borrowed from an imperative language. This combination of paradigms suggests two strategies for translating L-system-based languages into a representation directly used by simulation programs [Prusinkiewicz and Hanan, 1992]:

- extend the formal notation for productions with constructs borrowed from an imperative language, or
- extend an existing imperative language with constructs inherent in L-systems.

The modeling program cpfg [Hanan, 1992] and its modeling language [Prusinkiewicz et al., 2000] are representative of the first approach. The interpreter of the cpfg language was constructed following the standard steps of lexical analysis, parsing, and object code generation. Nevertheless, in spite of well-developed methodology for translator construction (e.g. [Aho et al, 1986]), construction of a compiler for a comprehensive language is a large task. Consequently, the cpfg language only includes a limited subset of C-like statements; for example, it does not support user-definable functions and typed parameters associated with the modules. As a result, while simple L-system models can be expressed using cpfg language in an elegant, compact manner, specification and maintenance of larger models becomes difficult.

An alternative approach, first suggested in [Prusinkiewicz and Hanan 1992], is to create an L-system-based programming environment by extending an existing language with support (classes, libraries) specific to L-systems. Using this approach, Hammel [1996] implemented differential L-systems [Prusinkiewicz et al., 1993] in SIMULA, and Erstad [2002] implemented an L-system-based programming environment in LISP. Both implementations preserve the syntax of the underlying languages (SIMULA and LISP). In contrast, Karwowski [2002] implemented the L-system-based programming language L+C by extending the syntax of C++ [Prusinkiewicz et al., 1999]. We describe here the design and implementation of this language.

2. The L+C modeling language

The key new elements introduced in the L+C modeling language are:

- typed module parameters, including all primitive and compound data types (structures) supported by C++,
- productions with multiple successors,
- extension of the notion of context-sensitivity with the 'new context' constructs, which speed up information transfer across simulated structures.

In addition, by virtue of being based on the C++ language, L+C has the full expressive power of C++. In particular, user-defined functions are supported as in C++.

At the top level, an L+C program is a set of declarations for:

- Structures and classes,
- Global variables,
- Functions,
- Modules,
- The axiom,
- The derivation length,
- Productions,
- Decomposition rules,
- Interpretation rules,
- Control statements.

The declarations of structures, classes, variables and functions have exactly the same syntax and meaning as in C++. The remaining declarations are specific to L+C, and are described below.

2.1. Module declarations

Modules are the elements of the L-system string[1]. A module consist of an identifier (which must follow the C++ syntax [Stroustroup, 1991]) and an optional list of parameters. In L+C modules have to be declared before they can be used. Declaration specifies the number and types of parameters that are associated with the given module type using the following syntax:

`module` *identifier* (*parameter-list*$_{opt}$);

Examples of valid module declarations are:

```
module A(); // module A with no parameters
module N(float); // module N with one parameter of type float
module Metamer(int, MetamerData); // module Metamer with a
                        // parameter of type int and
                        // a user-defined type MetamerData
```

2.2. Axiom declaration

The axiom declaration specifies the initial L-system string using the following syntax:

`axiom`: *parametric-string*;

where the *parametric-string* must be non-empty. Assuming that the modules have been declared as in Section 2.1, and `s_init` is a structure of type `MetamerData`, the following is a valid axiom declaration:

```
axiom: Metamer(1,s_init) N(0.25) A();
```

2.3. Derivation length specification

Derivation length is the number of derivation steps for the simulation. It is specified using the syntax:

`derivation length`: *integer-expression*;

2.4. Specification of productions

The syntax of productions is a combination of the formal L-system notation and the C++ syntax for function definition. In general, it has the syntax:

predecessor:
{
    production body
}

[1] This use of the term "module" originates in biology, where repetitive components of plant architecture are referred to as modules. It is also commonly used to denote L-system symbols with the associated parameters, (c.f. [Prusinkiewicz et al., 1997]) because in many applications they represent modules in the biological sense of the word.

The predecessor has one of the following forms:

new-left-context << *left-context* < *strict-predecessor* > *right-context* :
left-context < *strict-predecessor* > *right-context* >> *new-right context*:

The strict predecessor specifies the part of the string being rewritten by the production. It can be a single module, as assumed in the usual definition of L-systems, or a string of several modules, as defined for pseudo-L-systems [Prusinkiewicz, 1986]. The optional left and right contexts are strings of modules that need to be in the neighborhood of the strict predecessor in order for the production to apply. The new contexts specify the modules that must be present in the neighborhood of the production successor, in the string being derived. This information is easily available if the string is being rewritten in a particular direction: from left to right in the case of new left context, and from right to left in the case of new right context (Figure 1). In theory, two-sided new context could also be defined, but its implementation is more difficult and, therefore, it is not supported by L+C.

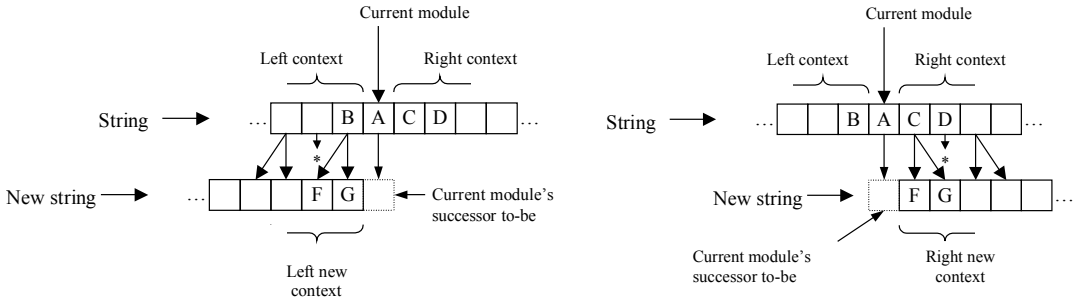


Figure 1. Context of L-system productions. Left new context is available if the successor string is built left-to-right (left figure). Right new context is available if the successor string is built right to left (right figure).

The parameters that appear in the production predecessor are formal parameters. All the formal parameters of every module in a production predecessor must be listed, even if they are not used in the production body. An example of a valid production predecessor that uses the modules declared in Section 2.1 is:

```
Metamer(i_l, d_l) N(w) < Metamer(i, d) > A()
```

Formal parameters have types determined by the declarations of the respective modules. They are bound to the actual parameters in the string during production application [Prusinkiewicz and Hanan, 1990]. The scope of the formal parameters is the same as the scope of formal parameters in C++ functions.

The production body is a compound statement that may contain any code allowed inside a C++ function. In addition, the production body may include one or more `produce` statements, which specify possible successors of the production. The `produce` statement has the syntax:

`produce` *parameteric-string*$_{opt}$;

where *parameteric-string* is defined as in the axiom (Section 2.2). Each `produce` statement is implicitly followed by a `return` statement. Thus, if several produce statements

are present in the production body, the first statement executed terminates the production application. Typically, the choice of alternative successors is controlled by C++ conditional statements.

2.5. Decomposition rules

As defined by Lindenmayer [1968], L-systems operate in discrete derivation steps. Each step consists of a (conceptually) parallel application of suitable productions to all symbols in the predecessor string. This parallelism is intended to capture progression of time by a given interval, the same for all components of the modeled structure. Thus, for example, the L-system production $A \rightarrow BC$ expresses the idea "module *A* develops into modules *B* and *C* over a given time interval." In practice, it is also often necessary to express the idea that a given module is a compound module, consisting of several elements. A logical analysis of the notions "develops over time" and "consists of" was presented by Woodger [1937]. Prusinkiewicz et al. [2000, 2001] showed that, in a grammar setting, these notions correspond to L-system productions and Chomsky context-free productions, respectively. In L+C, Chomsky productions are called decomposition rules. They are specified using the same syntax as context-free L-system productions, and are identified using the keyword `decomposition`, as in the following example:

```
decomposition:
Metamer(i, d) : { produce Internode(i, d) Leaf(d) Bud();}
```

This production characterizes a `Metamer` as a compound module consisting of an `Internode`, a `Leaf`, and a `Bud`. Obviously, all modules must have been declared earlier in the L+C program.

The integration of decomposition rules into the L-system framework affects the way in which a derivation step is performed [Prusinkiewicz et al., 2000]. In L+C, decomposition rules are applied recursively, after the definition of the initial string by the `axiom` statement (Section 2.2) and after each step of standard L-system production applications (Section 2.4).

2.6. Interpretation rules

Structures generated with L-systems may be visualized by assigning a graphical interpretation to a predefined set of modules [Szilard and Quinton, 1979; Prusinkiewicz, 1986, Prusinkiewicz et al., 2003]. For example, in L+C, a predefined module `F(float)` draws a line of a given length in the current direction (as defined in the turtle geometry [Abelson and diSessa, 1982]); `Line2D (point2D, point2D)` draws a line between two given points, and `SetColor(int)` assigns a color to geometric primitives. From the user perspective, however, it is often more convenient to express the model in terms of modules inherent in the modeling domain (e.g., apices, internodes, and leaves in the case of plant models) rather than directly in terms of modules with a geometric interpretation (e.g., points, lines, and polygons). In order to separate these conceptual and visual aspects of model specification, Kurth [1994] introduced the notion of interpretation rules. Interpretation rules are similar to decomposition rules in that they are context-free Chomsky productions, and are applied recursively, after each derivations step (specifically, after the decomposition rules have been applied). In contrast to decomposition rules, however, interpretation rules do not affect the outcome of the following derivation steps. In-

stead, they are applied "on the side", producing modules that are passed to the graphical part of the modeling program, and discarded once they have been interpreted (Figure 2).

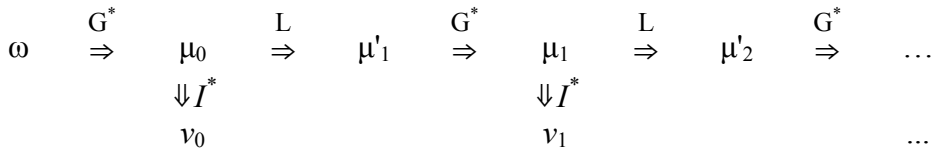


Figure 2. Generation of a developmental sequence using an L-system with decomposition and interpretation rules. Beginning with the axiom ω, the progressions of strings μ_1, μ_2, μ_3,… results from the interleaved application of decomposition rules G and L-system derivation steps L. The interpretation rules I map strings μ_i into strings ν_i , which are interpreted graphically.

In L+C, interpretation rules are identified using the keyword `interpretation`, as in the following example:

```
interpretation:
Internode(i, d) : { produce SetColor(1) F(d.length); }
```

The above production specifies that module `Internode` will be represented graphically as a straight line, (`F`) drawn using color with index `1`. The line length is specified by field `length` in data structure `d`.

2.7. Control statements

Control statements were introduced by Hanan [1992] (see also Prusinkiewicz et al., 2000]) to specify procedures that are executed at specific points during an L-system-based derivation. In L+C, they are specified using the syntax:

```
Start|StartEach|EndEach|End:
{
        compound statement
}
```

The control statements are executed as follows:

- `Start` is executed at the beginning of the program,
- `StartEach` is executed before every derivation step,
- `EndEach` is executed after every derivation step,
- `End` is executed after the last derivation step.

Any code that is allowed inside a C++ function can be specified as the *compound statement*. Typical uses of the control statements include initialization of global variables, opening and closing of I/O streams, and reporting of simulation statistics after each simulation step.

2.8. Example

A sample L+C program that generates a branching structure is presented below:

```
 1  #include <lpfgall.h>
 2  #include <math.h>
 3
 4  const int Delay = 1;
 5  const float BranchingAngle = 45.0;
 6  const float LengthGrowthRate = 1.33;
 7
 8  derivation length: 17;
 9
10  struct InternodeData
11  { float length, area; };
12
13  module Apex(int,float);
14  module Metamer(float);
15  module Internode(InternodeData);
16
17  Start: { Backward(); }
18  ignore: Right;
19
20  axiom: Apex(0,BranchingAngle);
21
22  Apex(t,angle) :
23  {
24    if (t<0) // young apex
25      produce Apex(t+1,angle);
26    else     // mature apex
27      produce Metamer(angle) Apex(0,-angle);
28  }
29
30  Internode(id) >> SB() Internode(idr1) EB() Internode(idr2) :
31  {
32    id.area = idr1.area + idr2.area;
33    id.length *= LengthGrowthRate;
34    produce Internode(id);
35  }
36
37  Internode(id) >> Internode(idr) :
38  {
39    id.area = idr.area;
40    id.length *= LengthGrowthRate;
41    produce Internode(id);
42  }
43
44  Internode(id) >> Apex(t,angle):
45  {
46    id.length *= LengthGrowthRate;
47    produce Internode(id);
48  }
49
50  decomposition:
51  Metamer(angle) :
52  {
53    InternodeData id = {1, 1};
54    produce
55        Internode(id)
56        SB() Right(angle) Apex(-Delay,angle) EB()
57        Internode(id);
58  }
59
60  interpretation:
61  Internode(id) :
62  {
63    produce SetColor(2) SetWidth(pow(id.area,.5)) F(id.length);
64  }
```

The modeled structure consists of three types of modules, which are given biologicaly meaningful names `Apex`, `Metamer`, and `Internode` (lines 13-15). The process of string derivation is performed backward (from right to left) as indicated in the Start statement (line 17). In the process of context matching module Right (used to specify the branching angle in line 56) is ignored (line 18). The initial structure defined by the axiom is a single apex. Its parameters characterize the developmental stage and the branching angle of the next branch that will be produced by this apex. According to the first production (lines 22-28), an immature apex will grow older, and a mature apex will produce a metamer, over the time interval associated with a derivation step. The decomposition rule (lines 51-58) specifies that the metamer consists of two internode segments and a lateral branch delimited by the language-predefined modules `SB()` (start branch) and `EB()` (end branch). The branch initially consists of a lateral apex, placed at a given `angle` with respect to its supporting internode. The development of internodes is described by the three productions in lines 30 to 48. They specify that an internode will grow in length by factor `LengthGrowthRate` per derivation step. They also determine the cross-section area of each internode as the sum of the cross-sections of internodes supported by it. Specifically, the new context construct is used to accumulate the cross-section of branches when moving from the apices toward the base of the structure. Finally, the interpretation rule (lines 61-64) specifies that each internode will be visualized as a line of length and width determined by the internode parameters. The structure generated by this L-system is shown in Figure 3.

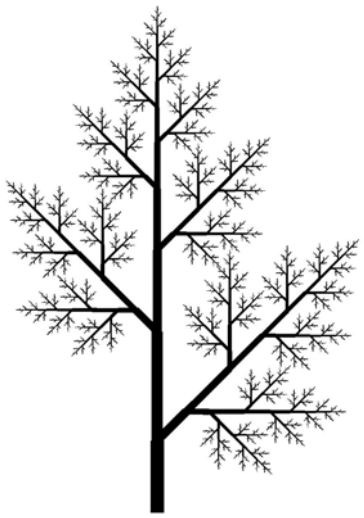


Figure 3. Example of a structure generated by the sample L-system.

3. Implementation of the L+C translator

The main difference between L+C and C++ is not at the level of syntax, but at the level of the programming paradigm: L+C is a declarative language, whereas C++ is an imperative language. Furthermore, L+C programs operate in a specific topological space [Gia-

vitto and Michel, 2001, 2002] of a linear or branching string, whereas C++ does not presuppose any such space. Despite these differences, most of the L+C grammar is the C++ grammar. Given that, the process of compiling and executing an L+C program consists of translating constructs specific to L+C into C++, while leaving other constructs intact. This leads to the modeling system design shown in Figure 4.

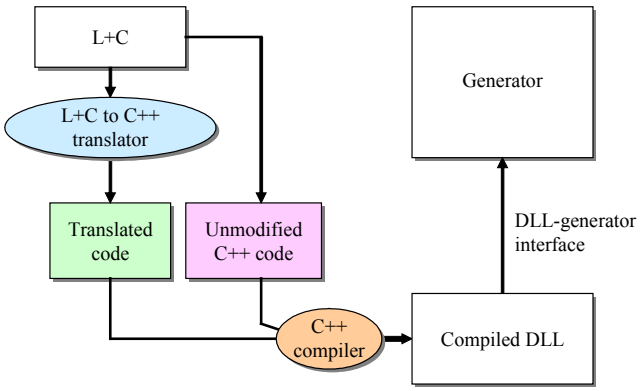


Figure 4. Components of our modeling system

Based on this design, the translator divides the input L+C code into two categories: the constructs specific to L+C, which are translated into a C++ code, and the remaining C++ constructs, which are passed verbatim to the compiler. The resulting C++ code is compiled using a standard C++ compiler into a DLL (dynamic link library). The actual execution of the L+C program is performed by a fixed component of a modeling program, called the generator (Figure 4). For the user's convenience, an L+C program can be modified, translated, compiled and run without a need for restarting the modeling program. Consequently, the L-system string derivation is performed based only on the information that can be provided by the DLL at the run time (since the generator is a fixed component and is not recompiled for every L+C program).

The DLL includes the interfacing information that makes the generator and the compiled L+C program communicate. We present this interface from the perspective of string derivation by the generator. The core of the generator is the `Execute()` function:

```
void Execute()
{
   Start();
   Axiom();
   DecomposeString();
   for (int i=0; i<DerivationLength(); ++i)
   {
      StartEach();
      Derive();
      DecomposeString();
      EndEach();
   }
   End();
}
```

where the functions written in boldface are defined in the process of translating the L+C program to C++ as follows:

- `Start()`, `StartEach()`, `EndEach()` and `End()` execute the compound statements specified in the corresponding L+C control statements (Section 2.7);
- `Axiom()` creates the initial L-system string (Section 2.2),
- `DerivationLength()` returns the value specified in the L-system `derivation length` statement (Section 2.3).

The translation of the L+C control statements into C++ functions is straightforward. For example, the L+C `Start` statement is translated as follows:

Original code	Translated code

`Start:`
`{`
`  …`
`}`
	`void Start()`
	`{`
	`  …`
	`}`

Analogous substitutions are made for the other L+C control statements. To process the `derivation length` statement, the translator replaces the L+C keyword with a C++ function prototype:

Original code	Translated code
`derivation length: 3;`	`int DerivationLength() { return 3; }`

In order to present the translation of productions, let us consider the following excerpt of L+C code as an example:

```
module A(data, float);
module B(int, float);

A(dl, xl) < B(n, a) :
{
   if (a>xl)
      produce B(n+1, xl);
   else
      produce B(n-1, xl);
}
```

Elements of the production typical for L+C are highlighted in boldface. The process of translation is based on the fact that productions are similar to functions in imperative programming languages [Prusinkiewicz and Hanan, 1992]. The similarities can be summarized into the following:

- A production is a piece of code to be executed,
- Its input is its predecessor and optionally, parameters of the predecessor's modules, and
- Its output is the successor.

The differences between productions and functions are as follows:

- L-system programs do not call productions explicitly. The general mechanism of matching productions at particular positions of the string determines which production should be applied and when.
- Productions do not return a value in the traditional sense. Instead, their output modifies the contents of the L-system string.

The first step in translating a production into a C++ function is to declare a function prototype, using the types declared in the relevant modules. For example, the following substitution is made:

Original code:	Translated code:
`A(dl, xl) < B(n, a)`	`void P1(data dl, float xl, int n, float a)`

Another element in the production code that needs to be translated is the produce statement. The code resulting from the translation of this statement must add the successor to the new string, and terminate the production. In our example, the produce statement is translated into code similar to this:

Original code:	Translated code:
`produce B(n+1, x);`	`{Append(B_id); Append(n+1); Append(x); return;}`

It should be noted that this translation process does not retain all the information that the generator needs to perform an L-system derivation. Specifically, the types of modules in the strict predecessor and the context are not reflected in the translated code. It is thus necessary to add information that bridges the generator and the translated L+C code for the purpose of finding the applicable production. To this end, the L+C translator creates a data structure that contains all the information about production predecessors needed for their matching to the string. Furthermore, in order to reconcile the L-system-independent generator with the L-system dependent production prototypes, the L+C translator creates a set of *caller functions,* one for each production. The caller functions bypass the C++ typing mechanism using low-level memory operations, and make it possible for the generator to call productions, even though their prototypes are not known at the time when the generator is compiled. Technical details of this implementation are given in [Karwowski, 2002].

4. Conclusions

We have described a modeling language L+C, which incorporates C++ into the framework of L-systems. We have also implemented a modeling system that uses L+C programs as input. To implement the L+C translator, we have introduced a methodology based on the separation of the constructs specific to L-systems from the C++ code. This methodology made it possible for a single person to implement the L+C translator in one month. The L-system-specific code is translated into C++ and combined with the C++ code taken verbatim from the L+C programs. The resulting code is translated into a DLL module using a standard C++ compiler. This module is linked with the generator that executes the L-systems. In practice, the DLL module is small in size compared to the generator and the graphical interpreter associated with it. Consequently, the DLL module

compiles and links fast (of the order of one second on the current Windows and Linux workstations), which allows for interactive manipulation and modification of the models. The increased expressiveness of L+C, compared to the previous L-system based languages, makes it possible to create models of a relatively greater complexity. L+C is currently being used to model aspects of plant genetics, physiology, and biomechanics.

References

Abelson, H. and diSessa, A. [1982]: *Turtle geometry*. M.I.T. Press, Cambridge.

Aho 1986: Aho, A., Sethi, R. and Ullman, J. [1986], *Compilers: Principles, techniques and tools*. Addison-Wesley, Reading.

Baker R. and Herman G. T. [1970]: Simulation of organisms using a developmental model, parts I and II. *International Journal of Bio-Medical Computing* **3**, pp. 201-215 and 251-267.

Chien, T. and Jurgensen, H. [1992]: Parameterized L systems for modelling: Potential and limitations. In: G. Rozenberg and A. Salomaa (Eds.): *Lindenmayer systems: Impacts on theoretical computer science, computer graphics, and developmental biology.* Springer, Berlin, pp. 213—229.

Erstad, K. [2002]: *L-systems, twining plants, Lisp*. M. Sc. thesis, University of Bergen.

Giavitto, J.-L. and Michel, O. [2001]: *MGS: A programming language for the transformation of topological collections*. Research Report, 61-2001, CNRS – Universite d'Evry Val d'Esonne.

Giavitto, J.-L. and Michel, O. [2002]: Data structures as topological spaces. Proceedings of the 3rd International Conference on Unconventional Models of Computation UMC02, *Lecture Notes in Computer Science* **2509**, pp. 137-150.

Hammel, M. [1996]: *Differential L-systems and their application to the simulation and visualization of plant development*. Ph. D. thesis, University of Calgary.

Hanan, J. [1992]: *Parametric L-systems*. Ph. D. thesis, University of Regina.

Karwowski, R. [2002]: *Improving the process of plant modeling: The L+C modeling language*. Ph. D. thesis, University of Calgary.

Knuth, D. [1968]: Semantics of context-free languages. *Mathematical Systems Theory* **2**, pp. 191-220.

Kurth, W. [1994]: *Growth grammar interpreter (GROGRA 2.4): A software tool for the 3-dimensional interpretation of stochastic, sensitive growth grammars in the context of plant modeling. Introduction and reference manual*. Forschungszentrum Waldokosysteme der Universitat Gottingen.

Lindenmayer, A. [1968]: Mathematical models for cellular interaction in development. *Journal of Theoretical Biology* **18**, pp. 280-315.

Lindenmayer, A. [1971]: Developmental systems without cellular interaction, their languages and grammars. *Journal of Theoretical Biology* **30**, pp. 455-494

Prusinkiewicz, P. [1986]: Graphical applications of L-systems. Proceedings of Graphics Interface '86 – Vision Interface '86, pp. 247-253.

Prusinkiewicz, P. and Hanan, J. [1989]: *Lindenmayer systems, fractals and plants*. Lecture Notes in Biomathematics **79**, Springer, Berlin.

Prusinkiewicz, P. and Hanan, J. [1990]: Visualization of botanical structures and processes using parametric L-systems. In: D. Thalmann (Ed.), *Scientific visualization and graphics simulation*, J. Wiley & Sons, Chichester, pp. 183-201.

Prusinkiewicz, P. and Hanan, J. [1992]: L-systems: From formalism to programming language*s*. In: G. Rozenberg and A. Salomaa (Eds.), *Lindenmayer systems: Impacts on theoretical computer science, computer graphics and developmental biology*. Springer, Berlin, pp. 193-211.

P. Prusinkiewicz, P., Hammel, M. and Mjolsness, E. [1993]: Animation of plant development. Proceedings of SIGGRAPH 93, pp. 351-360.

Prusinkiewicz, P. [1997]: A look at the visual modeling of plants using {{L}-systems. In R. Hofestadt and T. Lengauer and M. Loffler and D. Schomburg (Eds.): Bioinformatics. *Lecture Notes in Computer Science* **1278**, Springer, Berlin, pp.11-29.

Prusinkiewicz, P., Hammel, M., Hanan, J. and Mech, R. [1997]: Visual models of plant development. In G. Rozenberg and A. Salomaa (Eds.): *Handbook of formal languages*, vol. 3, Springer, Berlin, pp. 535-597.

Prusinkiewicz, P., Karwowski, R., Perttunen, J. and Sievanen, R. [1999]: Specification of L – a plant modeling language based on L-systems. Version 0.5. Internal Report, Department of Computer Science, University of Calgary, 1999.

Prusinkiewicz, P., Hanan, J., and Mech, R. [2000]: An L-system-based plant modeling language. In M. Nagl, A. Schuerr and M. Muench (Eds.): Applications of graph transformation with industrial relevance. *Lecture Notes in Computer Science* **1779**, Springer, Berlin, pp. 395-410.

Prusinkiewicz, P., Muendermann, L., Karwowski, R. and Lane, B. [2001]: The use of positional information in the modeling of plants. Proceedings of SIGGRAPH 2001, pp. 289-300.

Prusinkiewicz, P., Samavati, F., Smith, C. and Karwowski, R. [2003]: L-system description of subdivision curves. To appear in the *International Journal of Shape Modeling*.

Stroustroup, B. [1991]: *The C++ Programming Language*, Addison-Wesley, Reading.

Szilard, A. and Quinton, R. [1979]: An interpretation for D0L systems by computer graphics. *The Science Terrapin* **4**, pp. 8-13.

Woodger J. [1937]: *The axiomatic method in biology*, University Press, Cambridge.