ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

M P A Sellink (Ed)

2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam 1997

Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam, 25-26 September 1997

Strategies of ELAN: Meta-Interpretation and Partial Evaluation

P. Borovansky and H. Kirchner

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

ISBN: 3-540-76228-0

Strategies of ELAN: Meta-interpretation and Partial Evaluation

Peter Borovanský

CRIN-CNRS & INRIA-Lorraine, BP 239
54506 Vandoeuvre lès Nancy Cedex, France

Hélène Kirchner

CRIN-CNRS & INRIA-Lorraine, BP 239
54506 Vandoeuvre lès Nancy Cedex, France

Abstract

ELAN is an environment for prototyping and combining different deduction systems described using rewrite rules and strategies. Two languages of strategies used for controlling rewriting are presented in this paper. The first one, called *built-in strategy language*, is hard-wired with the implementation of ELAN, and thus, non-extensible from the user's point of view. However, it provides an efficient implementation of the more flexible second one, the *defined strategy language*.

This paper illustrates the defined strategy language on an example of the leftmost innermost normalization strategy, and describes its implementation in ELAN through a meta-interpreter. An optimization technique based on partial evaluation of strategies is presented in this paper. This technique applied to the meta-interpreter of the defined strategy language gives promising results.

1 Introduction

ELAN is a flexible system for prototyping and studying different deduction systems used for modeling constraint solving, theorem proving and logic programming applications [4]. The kernel of ELAN consists of an interpreter and an efficient compiler of rewrite systems with strategies, called *computational systems* [10, 13, 4].

In this paper, two languages of strategies are presented. The first one, called *built-in strategy language*, is hard-wired with the implementation of ELAN, and thus, non-extensible from the user's point of view. The implementation of the built-in strategy language, thanks to the ELAN compiler [14], is efficient enough to be used as an implementation tool for the second framework, the *defined strategy language*. The second approach is related to a view of strategies in reflective logics (in particular, rewriting logic) developed in [5].

The defined strategy language is implemented in ELAN using rewriting logic. This means that definitions of strategies are given as computational systems, i.e. ELAN programs. This approach is more flexible but less efficient than implementing in C++. This was our motivation to use *partial evaluation techniques* for transforming computational systems in order to improve efficiency of the meta-interpreter for the defined strategy language.

The structure of this paper is the following: Section 2 presents two strategy languages used for controlling rewriting, Section 3 discusses several implementation issues concerning the interpreter of defined strategies. Section 4 illustrates an optimization technique based on partial evaluation giving quite promising speed-up.

2 Strategies of ELAN

Non-deterministic strategies used for controlling rewriting is one of the main originalities of ELAN compared to other algebraic specification systems based on rewriting. In this section, we present two

strategy languages. The first one is the *built-in strategy language* implemented in C++ and therefore built-in in ELAN. The second one is the *defined strategy language* implemented in rewriting logic. The main difference between them is expressive power. While strategies of the built-in strategy language are regular expressions over a set of rules, combined with a few built-in strategy constructors, strategies of the defined strategy language can also be defined by recursive and parameterized rewrite rules. From the user's point of view, the implementation of the built-in language is invisible, and thus, non-extensible, while the implementation of the defined strategy language is based on its transformation into a computational system (a rewrite system with strategies), which is further evaluated thanks to the ELAN interpreter. The only part which is invisible for the user, is this automatic transformation into a computational system.

2.1 The Built-in Strategy Language

The ELAN system [10] allows prototyping non-deterministic computations thanks to rules and built-in strategies. An ELAN program is composed of a signature part describing operators with their types, a set of rules and a set of strategies. A strategy is a way to describe which computations the user is interested in, and specifies where a given rule should be applied in the term to be reduced. We describe informally here the evaluation mechanism and how it deals with rewrite rules and strategies.

Rules are labeled conditional rewrite rules with local variable assignments

$$[\ell] : l \Rightarrow r \quad \text{if } v \quad \text{where } y := (s)u$$

where ℓ is the label, l and r the respective left and right-hand sides, v the condition and $y := (s)u$ a local assignment assigning to the local variable y the result of the strategy s applied to the term u .

For applying such a rule on a term t , say at top position, first l is matched against t , then the expressions introduced by **where** and **if** are instantiated with the matching substitution and evaluated in order. Instantiations of local variables (such as y , after **where**) extend the matching substitution. When every condition is satisfied, the replacement by the instantiated right-hand side is performed.

Instantiations of local variables after **where** invoke ELAN built-in strategies. These strategies are regular expressions built on the alphabet of rule labels and several built-in strategy constructors. Some of them used in this paper are: **;**, **dc**, **dk**, **id**, **fail**, others can be found in [2]. Two strategies can be concatenated by the symbol “;”, i.e. the second strategy is applied on all results of the first one. Non-determinism is handled with two operators: **dc** standing for dont-care-choose and **dk** standing for dont-know-choose. For rewrite rules $\ell_1, \dots, \ell_n$, the strategy **dc**($\ell_1, \dots, \ell_n$) returns the results of one non-failing rule, un-deterministically chosen among the ℓ_i . On the contrary, for the strategy **dk**($\ell_1, \dots, \ell_n$), all possible results are computed and returned. This is implemented by backtracking on all rules $\ell_1, \dots, \ell_n$. **dc**($s_1, \dots, s_n$) un-deterministically chooses one of the non-failing strategies s_i , i.e. whose application gives some results, and those results are also results of **dc**($s_1, \dots, s_n$). If all sub-strategies fail, then it fails too. **dk**($s_1, \dots, s_n$) returns all results of all sub-strategies s_i . The identity strategy **id** does not change a term, while the strategy **fail** always fails, and never gives any result.

The ELAN built-in strategies can be named, and thus, they can be referred to from another strategy or a rule using assigned names. However, they cannot be either recursive or parameterized. Examples of built-in strategies can be found later in Figures 4 and 5.

2.2 The Defined Strategy Language

This section illustrates the idea of *controlling rewriting by rewriting* by defining a strategy language based on rewriting.

An *elementary strategy* es representing a non-recursive, non-deterministic computation is an element of the set of terms $\mathcal{T}(\mathcal{F} \cup \mathcal{L} \cup \{;, \text{dc}, \text{dk}, \text{first}, \text{id}, \text{fail}\})$. In addition to **dc**, **dk**, **id**

[DK]	$[\mathbf{dk}(es_1, es_2)]t$	$\Rightarrow [es_1]t \cup [es_2]t$
[DC ₁]	$[\mathbf{dc}(es_1, es_2)]t$	$\Rightarrow ([es_1]t)_\downarrow \text{ if } true = (t \in dom(es_1))_\downarrow$
[DC ₂]	$[\mathbf{dc}(es_1, es_2)]t$	$\Rightarrow ([es_2]t)_\downarrow \text{ if } true = (t \in dom(es_2))_\downarrow$
[DC ₃]	$[\mathbf{dc}(es_1, es_2)]t$	$\Rightarrow \emptyset \text{ if } true \neq (t \in dom(es_1))_\downarrow \text{ and } true \neq (t \in dom(es_2))_\downarrow$
[FIRST ₁]	$[\mathbf{first}(es_1, es_2)]t$	$\Rightarrow ([es_1]t)_\downarrow \text{ if } true = (t \in dom(es_1))_\downarrow$
[FIRST ₂]	$[\mathbf{first}(es_1, es_2)]t$	$\Rightarrow ([es_2]t)_\downarrow \text{ if } true \neq (t \in dom(es_1))_\downarrow$
[ID]	$[\mathbf{id}]t$	$\Rightarrow \{t\}$
[FAIL]	$[\mathbf{fail}]t$	$\Rightarrow \emptyset$
[FSYM ₁]	$[\mathbf{f}(es_1, \dots, es_n)]f(x_1, \dots, x_n)$	$\Rightarrow f([es_1]x_1, \dots, [es_n]x_n)$
[FSYM ₂]	$[\mathbf{f}(es_1, \dots, es_n)]g(x_1, \dots, x_m)$	$\Rightarrow \emptyset \text{ if } f \neq g$
[LAB ₁]	$[\mathbf{l}(es_1, \dots, es_n)]u(x_1, \dots, x_n)$	$\Rightarrow u'(\dots, [es_i]x_i, \dots)$ where $[\ell]u \Rightarrow u' \in \mathcal{R}$
[LAB ₂]	$[\mathbf{l}(es_1, \dots, es_n)]t$	$\Rightarrow \emptyset \text{ if } \emptyset = (u \ll^? t)_\downarrow \text{ where } [\ell]u \Rightarrow u' \in \mathcal{R}$
[CONC]	$[(es_1; es_2)]t$	$\Rightarrow [[es_2]] [es_1]t$

Figure 1: The interpreter of elementary strategies

and **fail** that reflect built-in strategies presented in Section 2.1, **first** is a “sequential version” of **dont-care-choose**, which takes always the first, in textual order, successful branch; $\mathbf{l}(es_1, \dots, es_n)$ corresponds to an application of a rewrite rule labeled by $\ell \in \mathcal{L}$, which also applies sub-strategies es_i on values of variables x_i of the rewrite rule, after matching and before replacing the matched subterm by the instantiated right-hand side of the rewrite rule; $\mathbf{f}(es_1, \dots, es_n)$ is an application of sub-strategies es_i to subterms t_i of the term $f(t_1, \dots, t_n)$ with root $f \in \mathcal{F}$.

The description of operational semantics of elementary strategies is achieved through the definition of an interpreter described by labeled rewrite rules, given in Figure 1. These rules formally define the binary function symbol $[-]_-$ that stands for the non-deterministic application of an elementary strategy es to the term t . The evaluation of the term $[es]t$ is not deterministic, due to the rules DC1 and DC2, defining $[\mathbf{dc}(es_1, es_2)]t$, that can apply simultaneously. So we can obtain as many results as **dc** operators contained in the strategy es .

The result of an application is a finite subset of terms. The operator $[[[-]]_-$ is the generalization of $[-]_-$ on sets of terms. Function application and replacement are also generalized to sets of terms ws_i as follows:

$$\begin{aligned} f(ws_1, \dots, ws_n) &= \{\mathbf{f}(e_1, \dots, e_n) \mid e_i \in ws_i, i = 1..n\}, \text{ and} \\ t(ws_1, \dots, ws_n) &= \{\{x_i \mapsto e_i\}(t) \mid x_i \in Var(t), e_i \in ws_i, i = 1..n\}, \end{aligned}$$

where $\{x_i \mapsto e_i\}(t)$ stands for a replacement of x_i by e_i in t .

The notation $([es]t)_\downarrow$ denotes a set in normal form according to the rules in Figure 1. Note that this is in general just *one of all* irreducible forms of $[es]t$, because the system of rules in Figure 1 is not deterministic. The notation $true = (t \in dom(es))_\downarrow$ means that $([es]t)_\downarrow$ is a non-empty set. Evaluation of the relation $_ \in dom(_)$ is similarly described by a rewrite theory in [3]. The symbol $\ll^?$ stands for matching in the empty theory. A rewrite theory for matching (as given in [9]) is added to the interpreter.

User-defined strategies extend elementary strategies by parameterized and recursive definitions using rewrite rules. Let us take the example of **map**, that can be defined by a strategy rewrite rule in the following *implicit form*:

$$\mathbf{map}(s) \Rightarrow \mathbf{dc}(\mathbf{nil}, s \bullet \mathbf{map}(s))$$

The right-hand side of this definition means that whenever the strategy **map** with its argument s (i.e. $\mathbf{map}(s)$) is applied to a term t , either t is *nil*, or the strategy s is applied on the head of t

(i.e. t should be a non-empty list) and $\mathbf{map} (s)$ is further applied on the tail of t . This definition is called implicit since the term, which the strategy is applied on, is implicit.

It can be reformulated using the strategy application symbol $[_]$ into the following *explicit form*:

$$\begin{aligned} [\mathbf{map} (s)]nil &\Rightarrow nil \\ [\mathbf{map} (s)]a.as &\Rightarrow [s]a.[\mathbf{map} (s)]as \end{aligned}$$

The difference relies on the fact that the list, which the functional $\mathbf{map}$ is applied on, is an explicit argument in the second definition, while in the first one, it is implicit. One can object that the explicit definition viewed as a rewrite theory is convergent, while the implicit one does not terminate. A natural solution to this problem is to distinguish the convergent and the divergent parts of a rewrite theory. Rules of the convergent part may be freely used for the normalization of strategy expressions. Examples of such rules for defined strategies are for instance:

$$\begin{aligned} \mathbf{map} (s_1) ; \mathbf{map} (s_2) &\Rightarrow \mathbf{map} (s_1 ; s_2) \\ \mathbf{map} (\mathbf{id}) &\Rightarrow \mathbf{id} \end{aligned}$$

Rules of the divergent part should be applied under certain restrictions, which invokes the concept of meta-strategies, i.e. strategies which control the execution by rewriting of defined strategies.

There is a subtle difference between bold face symbols, like $\mathbf{nil}$, $\mathbf{-} \bullet \mathbf{-}$ representing constructors of elementary strategies, and symbols nil , $_{-}$, which are constructors of the sort $\mathbf{list}[X]$. The operational semantics of $s_1 \bullet s_2$ is to apply the strategy s_1 to the head, and s_2 to the tail of a non-empty list, while the symbol $_{-}$ constructs a non-empty list. The typing reflects also this difference. In [3], a *typing system* is given for defined strategies, that introduces strategy sorts $\langle S \rightarrow S \rangle$ for any sort S of the user's signature. Elementary and defined strategies transforming a term t of sort S into a term t' of the same sort, are typed by the strategy sort $\langle S \rightarrow S \rangle$. In our example, there are the following *sort preserving* strategy constructors: $\mathbf{nil} : \langle \mathbf{list}[X] \rightarrow \mathbf{list}[X] \rangle$, resp. $\mathbf{-} \bullet \mathbf{-} : (\langle X \rightarrow X \rangle \langle \mathbf{list}[X] \rightarrow \mathbf{list}[X] \rangle) \langle \mathbf{list}[X] \rightarrow \mathbf{list}[X] \rangle$, and $\mathbf{map} (-) : (\langle X \rightarrow X \rangle) \langle \mathbf{list}[X] \rightarrow \mathbf{list}[X] \rangle$. More complex typing illustrated in [3] deals also with *sort-changing* strategies.

2.3 An Example

Before explaining several implementation details in Section 3, let us start with an example of the strategy language, which describes the normalization of λ -terms using the leftmost innermost and the leftmost outermost strategies. λ -terms are represented using de Bruijn notation [6], and we suppose that there are two primitive operations: $free(v, t)$ meaning that a variable v is free in a λ -term t , and $replace(v, a, t)$ standing for $\{v \mapsto a\}(t)$. The only difference with traditional definitions of $free$ and $replace$ is that, in the de Bruijn calculus, these functions must increment the index v when passing through λ construction, i.e. $replace(v, \lambda N, M) \Rightarrow \lambda replace(v+1, N, M)$. The definitions of λ -terms and β and η transition rules are straightforward in ELAN (see Figure 2). However, the definition of two normalization strategies $lis(s)$ and $los(s)$ is less evident (see Figure 3). These strategies $lis(s)$ and $los(s)$ fail if the input λ -term does not contain any s -redex reducible with the sub-strategy s (later instantiated to $\mathbf{dc}(\beta, \eta)$). The strategy $lis(s)$ tries to apply the sub-strategy $(lis(s) \mathbf{id})$, which succeeds, if the input term has a form $(M N)$ and $lis(s)$ is applicable on M , i.e. M contains s -redex. Otherwise, it continues on the right λ -subterm N by the application of $(\mathbf{id} lis(s))$. If the input term has a form λM , the strategy $lis(s)$ is propagated towards M by the application of $la lis(s)$. If none of the three cases above succeeds, the strategy s is applied on the top of this λ -term. The fact, that the sub-strategy s is applied last, makes the crucial difference between strategies $lis(s)$ and $los(s)$.

We use the Church's numbers defined as $zero = \lambda \lambda 1$ and $succ = \lambda \lambda \lambda (2 ((3 2) 1))$ to illustrate normalization of λ -terms. Iterating the leftmost innermost strategy $lis(\mathbf{dc}(\beta, \eta))$ on $two = (succ (succ zero))$, we get: $(\lambda \lambda \lambda (2 ((32) 1)) \lambda \lambda (2 ((\lambda \lambda 12) 1))) \Rightarrow (\lambda \lambda \lambda (2 ((32) 1)) \lambda \lambda (2 (\lambda 1 1))) \Rightarrow (\lambda \lambda \lambda (2 ((32) 1)) \lambda \lambda (2 (2 1))) \Rightarrow \lambda \lambda (2 ((\lambda \lambda (2 1) 2) 1)) \Rightarrow \lambda \lambda (2 (\lambda (2 1) 1)) \Rightarrow \lambda \lambda (2 (2 1))$, while using the

```

import global int; local bool; end
sort lterm; end
operators global
  @      : (int) lterm;
  la @   : (lterm) lterm;
  (@ @)  : (lterm lterm) lterm;
end
rules for lterm
M, N    : lterm;
global
  [beta]      (la M N) => replace(1,M,N)      end
  [eta]       (la M 1) => M if not free(1,M) end
end
    
```

 Figure 2: λ -terms

```

import global strat[lterm];
end
operators global
  lis(@): (<lterm->lterm) <lterm->lterm;
  los(@): (<lterm->lterm) <lterm->lterm;
end
strategies for lterm->lterm
s      : <lterm->lterm;
implicit
  [.]   lis(s) => first( (lis(s) id), (id lis(s)), la lis(s), s ) end
  [.]   los(s) => first( s, (los(s) id), (id los(s)), la los(s) ) end
end
    
```

Figure 3: Basic normalization strategies

leftmost outermost strategy $los(dc(beta, eta))$, we obtain:

$$\lambda \lambda (2 ((\lambda \lambda \lambda (2 ((3 2) 1)) \lambda \lambda 1) 2) 1)) \Rightarrow \lambda \lambda (2 ((\lambda \lambda (2 ((\lambda \lambda 1 2) 1)) 2) 1)) \Rightarrow \lambda \lambda (2 (\lambda (2 ((\lambda \lambda 1 2) 1)) 1)) \Rightarrow \lambda \lambda (2 (2 ((\lambda \lambda 1 2) 1))) \Rightarrow \lambda \lambda (2 (2 (\lambda 1 1))) \Rightarrow \lambda \lambda (2 (2 1)).$$

The definition of normalization strategies can be generalized from λ -calculus to any rewrite system, and a polymorphic version of *lis* and *los* is available in our strategy language.

3 An Interpreter of the Strategy Language

In this section, we discuss several implementation issues of the defined strategy language, and we give a simplified ELAN code of the strategy interpreter. The interpreter described in Figure 1 has been first implemented in ELAN as a non-deterministic rewrite theory. It computes several results corresponding to different **dc**-choices made during the computation, each of them representing a set of different solutions corresponding to **dk**-choices. This implementation approach was adequate for modeling semantics, however not realistic for the purpose of a real controlling language. For this reason, it was further improved by a few design decisions: 1) built-in strategies have been used to restrict computations of the strategy interpreter, 2) **dc**-non-determinism takes always the first found solution, and 3) **dk**-non-determinism is modeled by backtracking of the built-in strategy **dk**. According to 2, there is no longer implementation difference between **dc** and **first**. According to 2 and 3, the profile of the strategy application symbol $[_]_-$ is $(\langle S \rightarrow S \rangle S) S$. As a consequence of using built-in strategies, some guarding conditions from the interpreter rules can be removed. For example, it is clear that the two rules of the interpreter labeled by $FSYM_1$, $FSYM_2$ are mutually exclusive, thanks to their conditions and left-hand sides. Thus we can remove the condition $f \neq g$, if we assume that these rules are applied with a built-in strategy $dc(FSYM_1, FSYM_2)$ trying them in this order. This results in a more efficient interpreter described in Figure 4.

```

rules for X
a, b, c : X;  s, s1, s2 : <X->X>;  ss1 : list[<X->X>]; cond : bool;
global
[DK1]  [dk(s1,ss1)]a => b where b:=(eval)[s1]a                                end
[DK2]  [dk(s1,ss1)]a => b where b:=(eval)[dk(ss1)]a                          end
[DC1]  [dc(s1,ss1)]a => b where b:=(eval)[s1]a                                end
[DC2]  [dc(s1,ss1)]a => b where b:=(eval)[dc(ss1)]a                          end
[IF1]  [if true then s1 else s2 fi]a => b where b:=(eval)[s1]a                end
[IF2]  [if false then s1 else s2 fi]a => b where b:=(eval)[s2]a               end
[IFO1] [if s then s1 orelse s2 fi]a =>[s1]b where b:=(eval)[s]a               end
[IFO2] [if s then s1 orelse s2 fi]a =>[s2]a                                   end
[IFO3] [s1]a => b where b:=(eval)[s1]a                                         end
[CONC] [s1 ; s2]a => c  where b:=(eval)[s1]a where c:=(eval)[s2]b            end
[ID]   [id]a => a                                                                end
[CONST] [a]a  => a                                                              end
[FIN]  b => b where cond:=(dc(NF1, NF2))is_nf(b) if cond                      end
end
rules for bool
a : X;          b : X;          s : <X->X>;
local
[NF1]  is_nf([s]a) => false                                                    end
[NF2]  is_nf(b)    => true                                                      end
end
strategies for X // evaluation ELAN-strategy of the interpreter
global
[]      eval => dc(
                dk(FSYM),      dk(DSTR),      dk(LAB),      dc(ID),
                dk(DK1, DK2), dk(DC1),      dk(DC2),
                dc(CONST),     dk(CONC),      dk(IF1),      dk(IF2),
                dc(dk(IFO1), dk(IFO2));      dk(IFO3)
            ) ;
        dc(FIN)                                                                end
end
    
```

Figure 4: The kernel of the strategy interpreter

The key point of the strategy interpreter is the built-in strategy *eval*, which evaluates an application $[s]t$ of the strategy s on the term t . The application $[s]t$ is syntactically expressed in the interpreter as **where** $_ := (eval)[s]t$. For a strategy s (built-in, or not) and a term t , it is important here to carefully distinguish between application of a built-in strategy s on t denoted $(s)t$ and the application of an elementary or user-defined strategy of t denoted $[s]t$.

The strategy *eval* has two parts. The first part is a case analysis of different language constructions (simplified for the purpose of this paper). The second part, $dc(FIN)$ with an auxiliary built-in strategy $dc(NF1, NF2)$, tests in a tricky way if the application $[s]t$ has been reduced to a term in $\mathcal{T}(\mathcal{F})$, or not. A boolean symbol *is_nf* applied to b is reduced due to the strategy $dc(NF1, NF2)$ to *true*, if $b \in \mathcal{T}(\mathcal{F})$. Otherwise, if b has not been reduced, i.e. has a form $[s]t, is_nf(b)$ is false. This condition is checked by the rule *FIN*.

The strategy *eval* refers to several rules:

- rules labeled *FSYM* for the interpretation of functional strategy constructors (e.g. **nil**, or **- • -**),
- rules labeled *DSTR* for the interpretation of user-defined strategies (e.g. **map**),
- rules labeled *LAB* for strategies of labeled rules,
- rules labeled *DC*, *DK*, *ID*, etc. for the interpretation of elementary strategy constructors, and
- several other rules, e.g. *IF*, *IFO*, for the interpretation of additional language constructions (not explained in this paper but easy to understand).

Rules labeled FSYM, LAB, DSTR, not specified in the code of the interpreter in Figure 4, represent three sets of rules generated from the user's signature and strategy definitions as follows:

- FSYM rules interpret the elementary strategy $f(es_1, \dots, es_n)$. For any symbol $f : (s_1 \dots s_n) s$, a rule FSYM is generated:

$$[\text{FSYM}] \quad [f(s_1, \dots, s_n)]f(x_1, \dots, x_n) \Rightarrow f(y_1, \dots, y_n) \text{ where } y_i := (eval)[s_i]x_i, i = 1..n$$

- LAB rules interpret the elementary strategy $l(es_1, \dots, es_n)$. For any rewrite rule from $\mathcal{R}$ of the form:

$$[\ell(x_1, \dots, x_n)] \quad u \Rightarrow u'$$

where $x_i \in Var(u)$ for $1 \leq i \leq n$, a rule LAB is generated:

$$[\text{LAB}] \quad [l(s_1, \dots, s_n)]u(x_1, \dots, x_n) \Rightarrow u'(y_1, \dots, y_n) \\ \text{where } y_i := (eval)[s_i]x_i, i = 1..n$$

- DSTR rules interpret user-defined strategies. For any implicit definition of a user-defined strategy d of the form $d(s_1, \dots, s_n) \Rightarrow s$, a rule DSTR is produced:

$$[\text{DSTR}] \quad [d(s_1, \dots, s_n)]x \Rightarrow y \quad \text{where } y := (eval)[s]x$$

For the traditional definition of the sort $\text{list}[X]$ with two constructors ($nil, \cdot$) and a rewrite rule: $[\ell(x_1 : X, x_2 : X)] \quad x_1 + 0 \Rightarrow x_1 + x_2$, the following elementary strategy constructors are generated:

$$\begin{aligned} \text{nil} & : \quad \langle \text{list}[X] \rightarrow \text{list}[X] \rangle \\ _ \bullet _ & : \quad (\langle X \rightarrow X \rangle \langle \text{list}[X] \rightarrow \text{list}[X] \rangle) \langle \text{list}[X] \rightarrow \text{list}[X] \rangle \\ l(_, _) & : \quad (\langle X \rightarrow X \rangle X) \langle X \rightarrow X \rangle \end{aligned}$$

which are interpreted by the following rules:

$$\begin{aligned} [\text{FSYM}] \quad [\text{nil}] nil &\Rightarrow nil \\ [\text{FSYM}] \quad [s_1 \bullet s_2] x_1.x_2 &\Rightarrow y_1.y_2 \quad \text{where } y_1 := (eval)[s_1]x_1 \text{ where } y_2 := (eval)[s_2]x_2 \\ [\text{LAB}] \quad [l(s)] x + 0 &\Rightarrow y + y \quad \text{where } y := (eval)[s]x \end{aligned}$$

Moreover, the following rule with the label DSTR is added for the interpretation of defined strategy **map**:

$$[\text{DSTR}] \quad [\text{map}(s)]x \Rightarrow y \quad \text{where } y := (eval)[dc(\text{nil}, s \bullet \text{map}(s))]x$$

The definition of rewrite rules DC2, resp. DK2 can be slightly optimized using program transformation techniques studied in Section 4. These transformations produce a speed-up of approximately 30 – 50% of the interpreter time.

Let us show these optimizations on the rule DC2. In the local assignment **where** $b := (eval)[dc(ss1)]a$, the strategy $eval$ is invoked to reduce the application term $[dc(ss1)]a$. From the structure of the strategy $eval$ and labeled rules referred in it, it is clear, that only rules DC1 and DC2 can be used for reduction of $[dc(ss1)]a$. Thus, instead of applying the complex strategy $eval$ in this assignment, a simpler strategy $eval_dc = dc(dk(DC1), dk(DC2)); dc(FIN)$ can be used, i.e. one can write **where** $b := (eval_dc)[dc(ss1)]a$.

When the application of the strategy $eval_dc$ is performed, the term $[dc(ss1)]a$ is matched with left-hand sides of the rules DC1 and DC2 used in the strategy $eval_dc$, i.e. with the terms of the form $[dc(s1, ss1)]a$. Since the symbols $[_]_$ and dc are present in both terms, two decomposition steps of the matching algorithm made at run-time can be saved thanks to partial evaluation: $[dc(s1', ss1')]a' \ll^? [dc(ss1)]a$ is decomposed to $dc(s1', ss1') \ll^? dc(ss1)$ and $a' \ll^? a$, and then

to $s1', ss1' \ll^? ss1$ and $a' \ll^? a$, where the variables $s1, ss1, a$ on left-hand sides have been renamed to $s1', ss1', a'$. All components of the decomposed matching problem can be regrouped into a term using a new auxiliary function symbol $((_))_- : (\text{list}[X \rightarrow X]) X \rightarrow X$, such that the matching of the initial terms is at compile-time reduced to the matching of $((s1', ss1'))a' \ll^? ((ss1))a$. Then, the rules DC1 and DC2 can be transformed to the following form:

$$\begin{aligned} \text{[DC1]} \quad ((s1, ss1))a &\Rightarrow b \quad \text{where } b := (eval)[s1]a \\ \text{[DC2]} \quad ((s1, ss1))a &\Rightarrow b \quad \text{where } b := (eval_dc)((ss1))a \end{aligned}$$

and the following rule makes a simple interface between the previous term $[dc(ss1)]a$ and the new one $((ss1))a$:

$$\text{[DC]} \quad [dc(ss1)]a \Rightarrow b \quad \text{where } b := (eval_dc)((ss1))a$$

In this transformation, the term $((ss1))a$ in the **where** statement is slightly more compact than the original one $[dc(ss1)]a$ since it contains less function symbols. This minimizes allocation and freeing of dynamic data structures at run-time, which is one of the most expensive operations. When this technique is applied on larger examples as in appendix, it gives fruitful results.

To summarize, rules DC1,2, resp. DK1,2 and the strategy *eval* have been replaced by the following ones, and the strategies *eval_dc*, *eval_dk* have been added to the optimized kernel (Figure 5). Optimizations obtained with these transformations motivate us to study them in the more

```
rules for X
a, b : X;  s1 : <X->X>;  ss1 : list[<X->X>;
global
[DK]    [dk(ss1)]a => b where b:=(eval_dk)((ss1))a           end
[DK1]   ((s1,ss1))a => b where b:=(eval)[s1]a               end
[DK2]   ((s1,ss1))a => b where b:=(eval_dk)((ss1))a         end
[DC]    [dc(ss1)]a => b where b:=(eval_dc)((ss1))a          end
[DC1]   ((s1,ss1))a => b where b:=(eval)[s1]a               end
[DC2]   ((s1,ss1))a => b where b:=(eval_dc)((ss1))a         end
end
strategies for X
global
[] eval_dc => dc(dk(DC1), dk(DC2)) ; dc(FIN)                 end
[] eval_dk => dk(DK1, DK2) ; dc(FIN)                         end
[] eval    => dc(
                    dk(FSYM),    dk(DSTR),    dk(LAB),    dc(ID),
                    dk(DK),      dk(DC),
                    dc(CONST),   dk(CONC),   dk(IF1),    dk(IF2),
                    dc(dk(IFO1), dk(IFO2)); dk(IFO3)
                ) ;
dc(FIN)
end
```

Figure 5: The optimized kernel

general framework of partial evaluation.

4 Partial Evaluation of Strategies

The notion of *partial evaluation* is well-known and has been heavily explored in the 80's for program transformations and optimizations [7], transformation of higher-order functional programs used to eliminate higher-order functions [1], and automatic compiler generation [8]. Furthermore, partial evaluation can be viewed as an instance of the more general method of *super-compilation* [11, 12].

The classical concept of partial evaluation is the following: a partial evaluator takes a subject program and a part of its input and produces a residual program. This residual program applied

to any remaining part of the input produces the same result(s) as the subject program applied to whole input. There are two substantial differences between this classical and our concept of partial evaluation:

- our method works also with input terms with variables from $\mathcal{T}(\mathcal{F}, \mathcal{X})$, not only with ground input terms from $\mathcal{T}(\mathcal{F})$,
- our method partially evaluates built-in ELAN strategies, not a whole ELAN program as a computational system.

Thus, in the following, by *program* we mean an ELAN built-in strategy, and its input represents a non-ground term, which this strategy is applied on.

We describe a general method of transformation for ELAN programs. The key idea of this method is an optimization of tried and applied ELAN rules during execution. The principal motivation for this kind of optimization comes from the strategy interpreter described in Section 3. However, the strategy interpreter is just an example on which this method is illustrated. This optimization method done at compile-time is general enough to work for any arbitrary ELAN strategy.

4.1 The Optimization Method

The strategy *eval* driving the interpreter (in Figure 5) handles several exclusive cases. Each of them corresponds either to one elementary strategy constructor, like **dc**, **dk**, **id**, or to a set of rewrite rules providing the application of function strategy symbols (FSYM), rule strategy symbols (LAB), or defined strategy symbols (DSTR). The idea of the optimization is to statically eliminate, for each application of a strategy (e.g. *eval*) on a term, most of the cases, where the strategy cannot succeed. Thus, a specialization of the original strategy takes advantage of the information obtained from the original one. This strategy may refer to new rewrite rules, which are instances of original ones.

We sketch the optimization on the example of the defined strategy **map** :

$$\mathbf{map}(s) \Rightarrow \mathbf{dc}(\mathbf{nil}, \mathbf{cons}(s, \mathbf{map}(s)))$$

which is transformed into the following rewrite rule with the label DSTR:

$$[\text{DSTR}] \quad [\mathbf{map}(s)]t \Rightarrow y \quad \text{where } y := (eval)[\mathbf{dc}(\mathbf{nil}, s \bullet \mathbf{map}(s))]t$$

For any value of the variable y during the execution, the strategy *eval* (in Section 3) tries to match the defined strategy constructor **dc** using a lot of unsatisfiable cases (e.g. the **dk**(FSYM), **dk**(DSTR), **dk**(DK), etc.) These failing cases can be eliminated thanks to the fact, that the term, which the strategy *eval* is applied to, is partially known at compile time. This term should be an instantiation of $[\mathbf{dc}(\mathbf{nil}, s \bullet \mathbf{map}(s))]t$ for specific values of s and t . Clearly, the only rules that ought to be tried are the rules interpreting the **dc** strategy constructor. So, the strategy *eval* can be replaced by its simplified form $\mathbf{dc}(\mathbf{dk}(\mathbf{DC})); \mathbf{dc}(\mathbf{FIN}) = \mathbf{dk}(\mathbf{DC}); \mathbf{dc}(\mathbf{FIN})$. This could be viewed as the first step of the optimization, where the huge strategy *eval* is reduced, due to the constructor **dc**, into a smaller one. From the input term $\mathbf{dc}(\mathbf{nil}, \mathbf{cons}(s, \mathbf{map}(s)))$ we only use the information that its root constructor is **dc**. We can take more advantage of the input term structure and optimize more than only one application of the strategy *eval*.

The strategy *eval* in the original rule can be specialized to a new particular strategy $eval' = \mathbf{dk}(\mathbf{DC}'); \mathbf{dc}(\mathbf{FIN})$ applied to the term $[\mathbf{dc}(\mathbf{nil}, \mathbf{cons}(s, \mathbf{map}(s))]t$. This new partial strategy refers to an optimized and instantiated version $\mathbf{DC}'$ of the original rule:

$$[\mathbf{DC}'] \quad [\mathbf{dc}(\mathbf{nil}, s \bullet \mathbf{map}(s))]t \Rightarrow y \quad \text{where } y := (eval_dc)((\mathbf{nil}, s \bullet \mathbf{map}(s)))t$$

and the strategy *eval* in the original rule can be replaced by $eval'$. The rule **FIN** following in the strategy $eval'$ the strategy $\mathbf{dk}(\mathbf{DC}')$ is not further optimized by this technique. The reason is

that, when the right-hand side of the rule DC' is a variable, our partial evaluation method is not able to deduce a structure for intermediate results of the sub-strategy $dk(DC')$. Since the sub-strategy $dc(FIN)$ of $eval'$ is applied to these results with unknown structure at compile time, our method is not able to optimize either the rule FIN , or the sub-strategy $dc(FIN)$.

The rule DC' contains an application of the strategy $eval_dc$ on a redex $((nil, s \bullet map(s)))t$. In the same way, the sketched method optimizes this occurrence of the strategy $eval_dc$. A new obtained strategy $eval_dc' = dc(dk(DC1'), dk(DC2')) ; dc(FIN)$ refers to new rewrite rules $DC1'$ and $DC2'$:

$$\begin{aligned} [DC1'] \quad & ((nil, s \bullet map(s)))t \Rightarrow y \quad \text{where } y := (eval)[nil]t \\ [DC2'] \quad & ((nil, s \bullet map(s)))t \Rightarrow y \quad \text{where } y := (eval_dc)((s \bullet map(s)))t \end{aligned}$$

which are instances of the original ones ($DC1$ and $DC2$). These two new rules bring two new redexes, so, we can continue to eliminate them. The whole partial evaluation of map is shown in appendix.

4.2 Description of the Transformation

In this section, we formalize the optimization method sketched above. As illustrated before, the method optimizes strategies applied on terms called *redexes*. These strategies are elements of the built-in strategy language, i.e. are recursively constructed using the constructors dk , dc , id , $;$, ℓ , and s , where ℓ stands for a label of a rewrite rule, and s stands for a built-in strategy.

A built-in strategy s applied on t in the redex $(s)t$ is reduced to a new strategy s' in order to minimize the number of tried and used labeled rules during the evaluation of $(s)t$. The price of this specialization is an enlargement of the user's rewrite system in terms of number of rules and strategies.

The specialization method works with terms with variables constructed over the user's signature, i.e. in $\mathcal{T}(\mathcal{F}, \mathcal{X})$, which are called *patterns*. A pattern p is a finite representation (or schematization) of its ground instances, obtained by instantiating its variables by terms of $\mathcal{T}(\mathcal{F})$. Input patterns represent the class of ground terms which a built-in strategy is applied to. Output patterns represent the set of all ground terms $\mathcal{T}(\mathcal{F})$, which can be produced by the application of this strategy. Unlabeled rewrite rules used by ELAN for leftmost innermost normalization are not considered in this transformation, and thus, patterns do not contain any function symbol which occurs at the top of a left-hand side of an unlabeled rewrite rule.

Specialization rules are described in Figure 6. The relation

$$s \diamond \{p_1, \dots, p_n\} \gg s' \circ \{q_1, \dots, q_m\}$$

where $\{p_1, \dots, p_n\}$ and $\{q_1, \dots, q_m\}$ are finite sets of patterns, and s, s' are built-in strategies, means that the strategy s can be replaced by the strategy s' , provided that all possible inputs of s are schematized by the set of input patterns $\{p_1, \dots, p_n\}$. The transformation preserves the meaning of the ELAN program, i.e. applications of strategies s and s' on t give the same results provided t is an instance of some p_i . Any result of these applications is also an instance of some q_j . Simultaneously, the user's rewrite system is enriched by new labeled rules and strategies.

If an ELAN program contains a redex $(s)t$ of a built-in strategy s applied to a term $t \in \mathcal{T}(\mathcal{F})$, the method consists first in building s' and O such that $s \diamond \{t\} \gg s' \circ O$, according to rules in Figure 6. The obtained strategy s' is simplified by several primitive transformations of strategies, e.g. $dc(s) \Rightarrow s$, $dk(s) \Rightarrow s$, $id; s \Rightarrow s$, $fail; s \Rightarrow fail$, etc., and then, the original strategy s is replaced by s' . If the output set of patterns O is empty, the strategy s is replaced by $fail$.

The specialization of the concatenation of two strategies $s_1 ; s_2$ is obtained from the specializations of s_1 and s_2 , while the set of output patterns Q of s_1 becomes a set of input patterns for s_2 . Moreover, the output patterns O of $s_1 ; s_2$ are output patterns of s_2 started with the input patterns Q . The specialization of $dk(s_1, \dots, s_n)$ or $dc(s_1, \dots, s_n)$ is obtained from specializations of each

case of concatenation $s_1; s_2$

$$\frac{s_1 \diamond I \gg s'_1 \circ Q \text{ and } s_2 \diamond Q \gg s'_2 \circ O}{s_1; s_2 \diamond I \gg s'_1; s'_2 \circ O}$$

case of dk, resp. dc, over strategies $\text{dk}(s_1, \dots, s_n)$, resp. $\text{dc}(s_1, \dots, s_n)$

$$\frac{s_j \diamond I \gg s'_j \circ Q_j}{\text{dk}(s_1, \dots, s_n) \diamond I \gg \text{dk}(s'_1, \dots, s'_n) \circ \bigcup_{j=1}^n Q_j}$$

case of id, resp. fail

$$\text{id} \diamond I \gg \text{id} \circ I \qquad \text{fail} \diamond I \gg \text{fail} \circ \emptyset$$

case of dk, resp. dc, over rules $\text{dk}(r_1, \dots, r_n)$, resp. $\text{dc}(r_1, \dots, r_n)$

$$\text{dk}(r_1, \dots, r_n) \diamond \{p_1, \dots, p_m\} \gg \text{dk}(\{r'_{i,j}\}) \circ \{q_{i,j}\} \quad (i = 1..m, j = 1..n)$$

where rules $r'_{i,j}$ and patterns $q_{i,j}$ are computed in the following way:

- if the pattern p_i and the left-hand side of the rule r_j are unifiable, i.e. there exists $\theta_{i,j} = \text{mgu}(p_i, \text{lhs}(r_j))$, then $q_{i,j} = \theta_{i,j}(\text{rhs}(r_j))$ and
 - if $\theta_{i,j}$ instantiates the rule r_j , then a new rule $r'_{i,j} = \theta_{i,j}(r_j)$ is added to the program,
 - else $r'_{i,j} = r_j$,
- else, $r'_{i,j} = \emptyset$.

Figure 6: The specialization rules

s_j with the same input patterns, and the set of output patterns is the union of all output patterns of all s_j . This union over sets of patterns removes pairs of subsumed terms, thus sets of patterns are always finite and do not contain any pair of terms t_1, t_2 , such that t_1 is an instance of t_2 . The specialization of $\text{dk}(r_1, \dots, r_n)$ is more technical, because several instantiations of rules have to be considered. For any pair of a labeled rule r_j and an input pattern p_i , an instance $\theta(r_j)$ of the rule r_j (where θ is m.g.u. of p_i and the left-hand side of r_j) is added to the rewrite system, provided the substitution θ instantiates any variable of the rule r_j , i.e. $r_j \neq \theta(r_j)$. This new rule may contain new redexes, so the transformation method is applied until it generates new instantiated rules that cannot be transformed anymore, or until a loop over redexes is detected. The loop detection eliminates non-terminating development of recursive definitions.

In general, the described optimization technique reduces the number of tried and used labeled rewrite rules. However, as we can see on the example of DC1' and DC2', newly generated rules are more complex than the original ones, because they are instances of them. This fact implies that the naive matching algorithm used in the ELAN interpreter, whose complexity is dependent on the size of terms, spends more time in the matching process. That is why, the last optimization step, called *term compression*, simplifies left-hand sides of rules with their corresponding expressions in **where** statements by introducing new auxiliary symbols as illustrated in Section 3.

This method is currently implemented in C++, but it can be re-implemented also in ELAN thanks to its reflexive features. Any correct program in ELAN has an external representation, which is also readable and reducible by the ELAN system itself. This allows us to implement this transformation in ELAN too.

Results presented below have been obtained with the strategy `repeat lis(dc(beta, eta))` applied to the term `(succ(succ(succ(succ(succ(succ(succ(succ(succ zero))))))))`. The strategy `repeat` is defined as `repeat s ⇒ first(s; repeat s, id)` and `lis` has been shown in Section 2.3. The method of partial evaluation generates 43 new rewrite rules and 36 new strategies. The reduction of the execution time is shown in Figure 7.

	total time (in sec)	rules applied	rules tried
non-optimized kernel	100.469	12.748	2.297.788
optimized library kernel	69.937	14.416	1.342.064
partial evaluation	25.794	7.404	376.215
partial evaluation & term compression	18.319	7.404	376.215

Figure 7: Statistics

5 Conclusion

This paper describes an approach to the design and implementation of the defined strategy language of ELAN. First, the defined strategy language has been outlined and illustrated on the example of leftmost innermost strategy, then, several aspects of its meta-interpretation have been considered. There are two possible continuations of the implementation development: compilation of the defined strategy language in C++, or optimization of the meta-interpreter of the defined strategy language using techniques of partial evaluation. In this paper, we have presented several promising results of the second approach. However, the obtained speed-up, which may be beaten by the compiler of the defined strategy language, seems to be less important than program transformation techniques illustrated here. These transformations are valuable for any ELAN programs, and can be adapted for similar meta-interpreters written in rewriting logic, as for instance the interpreter of Maude strategies [5].

Acknowledgments

We sincerely thank Claude Kirchner and Pierre-Etienne Moreau for all our interactions on ELAN from the design to the very implementation. We sincerely thank Manuel Clavel and José Meseguer for many valuable discussions on strategies.

References

- [1] F. Bellegarde. Program transformation and rewriting. In R.V. Book, editor, *Rewriting Techniques and Applications*, volume 488 of *Lecture Notes in Computer Science*, pages 226 – 239, Como, Italy, April 1991. Springer-Verlag.
- [2] P. Borovanský, C. Kirchner, Hélène Kirchner, P.E. Moreau, and Marian Vittek. *ELAN V 2.00 User Manual*. Inria Lorraine & Crin, Nancy (France), first edition, may 1997.
- [3] Peter Borovanský, Claude Kirchner, and Hélène Kirchner. Controlling rewriting by rewriting. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.
- [4] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. *ELAN: A logical framework based on computational systems*. In José Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.
- [5] M. G. Clavel and J. Meseguer. Axiomatizing Reflective Logics and Languages. In G. Kiczales, editor, *Proceedings of Reflection’96, San Francisco, California, April 1996*, pages 263–288. Xerox PARC, 1996.

- [6] N. G. de Bruijn. Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Proc. Koninkl. Nederl. Akademie van Wetenschappen*, 75(5):381–392, 1972.
- [7] Y. Futamura. Partial computation of programs. In *Symp. Software Science and Engineering*, volume 147 of *Lecture Notes in Computer Science*, pages 1 – 35, Kyoto, Japan, 1983. Springer-Verlag.
- [8] N. D. Jones, P. Sestof, and H. Sondergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science*, pages 124 – 140, Dijon, France, May 1983. Springer-Verlag.
- [9] J.-P. Jouannaud and Claude Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In Jean-Louis Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.
- [10] Claude Kirchner, Hélène Kirchner, and Marian Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, chapter 8, pages 131–158. The MIT press, 1995.
- [11] V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
- [12] V.F. Turchin. Program transformation with metasystem transitions. *J.Functional Programming*, 3(3):283–313, July 1993.
- [13] Marian Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes*. Thèse de Doctorat d’Université, Université Henri Poincaré – Nancy 1, October 1994.
- [14] Marian Vittek. A compiler for nondeterministic term rewriting systems. In Harald Ganzinger, editor, *Proceedings 7th Conference on Rewriting Techniques and Applications, New Brunswick (New Jersey, USA)*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–168. Springer-Verlag, July 1996.

Appendix

Here, we completely develop the partial evaluation of `map` example. In each step, a strategy applied on a redex with the built-in strategy is underlined. Original rules of the interpreter in Figures 4 resp. 5, e.g. `FSYM`, are twice indexed: `FSYM2(4)`. The lower index 2 makes difference between all original rules `FSYM` generated from the user’s signature. The upper index (4) indicates the generation of this rule, and makes difference between different instances of the same original rule. The same generation index is used for new built-in strategies.

1. In the rule `[DSTR2]`, the strategy `eval` is applied on a redex `[dc( nil , x1 • map (x1))]x0`
`[DSTR2] [map (x1)]x0 ⇒ x6 where x6 := (eval)[dc( nil , x1 • map (x1))]x0`
is replaced by `eval(0) = DC(0) ; dc(F|N)` and a new rule `DC(0)` is generated:
`[dc( nil , x0 • map (x0))]x1 ⇒ x2 where x2 := (eval_dc)(( nil , x0 • map (x0)))x1`

2. In the rule $[DC^{(0)}]$, the strategy $eval_dc$ is applied on a redex $((nil, x_0 \bullet map(x_0)))x_1$
 $[DC^{(0)}] \quad dc(nil, x_0 \bullet map(x_0))x_1 \Rightarrow x_2 \text{ where } x_2 := (eval_dc)((nil, x_0 \bullet map(x_0)))x_1$
 is replaced by $eval^{(1)} = dk(dk(DC1^{(1)}), dk(DC2^{(1)})); dc(FIN)$ and new rules are generated:
 $[DC1^{(1)}] \quad ((nil, x_0 \bullet map(x_0)))x_1 \Rightarrow x_2 \text{ where } x_2 := (eval)[nil]x_1$
 $[DC2^{(1)}] \quad ((nil, x_0 \bullet map(x_0)))x_1 \Rightarrow x_2 \text{ where } x_2 := (eval_dc)((x_0 \bullet map(x_0)))x_1$
3. In the rule $[DC2^{(1)}]$, the strategy $eval_dc$ is applied on a redex $((x_0 \bullet map(x_0)))x_1$
 $[DC2^{(1)}] \quad ((nil, x_0 \bullet map(x_0)))x_1 \Rightarrow x_2 \text{ where } x_2 := (eval_dc)((x_0 \bullet map(x_0)))x_1$
 is replaced by $eval^{(2)} = dk(dk(DC1^{(2)}), dk(DC2^{(2)})); dc(FIN)$ and new rules are generated:
 $[DC1^{(2)}] \quad ((x_0 \bullet map(x_0)))x_1 \Rightarrow x_3 \text{ where } x_3 := (eval)[x_0 \bullet map(x_0)]x_1$
 $[DC2^{(2)}] \quad ((x_0 \bullet map(x_0)))x_1 \Rightarrow x_3 \text{ where } x_3 := (eval_dc)((x_0 \bullet map(x_0)))x_1$
4. In the rule $[DC1^{(1)}]$, the strategy $eval$ is applied on a redex $[nil]x_1$
 $[DC1^{(1)}] \quad ((nil, x_0 \bullet map(x_0)))x_1 \Rightarrow x_2 \text{ where } x_2 := (eval)[nil]x_1$
 is replaced by $eval^{(3)} = dk(FSYM_3); dc(FIN^{(3)})$ and a new rule $FIN^{(3)}$ is generated:
 $nil \Rightarrow nil \text{ where } x_0 := (dc(NF1, NF2))is_nf(nil) \text{ if } x_0$
5. In the rule $[DC1^{(2)}]$, the strategy $eval$ is applied on a redex $[x_0 \bullet map(x_0)]x_1$
 $[DC1^{(2)}] \quad ((x_0 \bullet map(x_0)))x_1 \Rightarrow x_3 \text{ where } x_3 := (eval)[x_0 \bullet map(x_0)]x_1$
 is replaced by $eval^{(4)} = dk(FSYM_2^{(4)}); dc(FIN^{(4)})$ and new rules are generated:
 $[FSYM_2^{(4)}] \quad [x_0 \bullet map(x_0)]x_1 \bullet x_2 \Rightarrow x_4 \bullet x_3$
 $\text{where } x_3 := (eval)[map(x_0)]x_2 \quad \text{where } x_4 := (eval)[x_0]x_1$
 $[FIN^{(4)}] \quad x_0 \bullet x_1 \Rightarrow x_0 \bullet x_1 \text{ where } x_3 := (dc(NF1, NF2))is_nf(x_0 \bullet x_1) \text{ if } x_3$
6. In the rule $[DC2^{(2)}]$, the strategy $eval_dc$ is applied on a redex $((x_0 \bullet map(x_0)))x_1$
 $[DC2^{(2)}] \quad ((x_0 \bullet map(x_0)))x_1 \Rightarrow x_3 \text{ where } x_3 := (eval_dc)((x_0 \bullet map(x_0)))x_1$
 is replaced by **fail**.
7. In the rule $[FSYM_2^{(4)}]$, the strategy $eval$ is applied on a redex $[x_0]x_1$
 $[FSYM_2^{(4)}] \quad [x_0 \bullet map(x_0)]x_1 \bullet x_2 \Rightarrow x_4 \bullet x_3$
 $\text{where } x_3 := (eval)[map(x_0)]x_2 \text{ where } x_4 := (eval)[x_0]x_1$
 is replaced by $eval^{(5)} = dk(DSTR_2); dc(FIN)$ and no new rule is generated.
8. In the rule $[FIN^{(3)}]$, the strategy $dc(NF1, NF2)$ is applied on a redex $is_nf(nil)$
 $[FIN^{(3)}] \quad nil \Rightarrow nil \text{ where } x_0 := (dc(NF1, NF2))is_nf(nil) \text{ if } x_0$
 is replaced by $dc(NF2^{(6)})$ and a new rule $NF2^{(6)}$ is generated:
 $is_nf(nil) \Rightarrow true$
9. In the rule $[FIN^{(4)}]$, the strategy $dc(NF1, NF2)$ is applied on a redex $is_nf(x_0 \bullet x_1)$
 $[FIN^{(4)}] \quad x_0 \bullet x_1 \Rightarrow x_0 \bullet x_1 \text{ where } x_3 := (dc(NF1, NF2))is_nf(x_0 \bullet x_1) \text{ if } x_3$
 is replaced by $dc(NF2^{(7)})$ and a new rule $NF2^{(7)}$ is generated:
 $is_nf(x_0 \bullet x_1) \Rightarrow true$

For a better visualization of the process of partial evaluation, we show dependencies between redexes of rules on a *dependency tree* (in Figure 8). Nodes of this tree are labels of rewrite rules with their redexes. Edges enumerated by phases of partial evaluation indicates that children nodes (rules) were generated due to partial evaluation of the parent node (rule). The detection of loops mentioned in Section 4.2 is made on this dependency tree. If a redex r of the dependency tree is an instance of another redex R occurring on the path between r and the root of the dependency tree, the redex r is no longer partially evaluated.

Figure 8: Dependency tree