M P A Sellink (Ed)


Strategies of ELAN: Meta-Interpretation and Partial Evaluation

P. Borovansky and H. Kirchner
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

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

where \( \ell \) 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: \( ; \), \( \mathbf{dc} \), \( \mathbf{dk} \), \( \mathbf{id} \), \( \mathbf{fail} \), 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: \( \mathbf{dc} \) standing for dont-care-choose and \( \mathbf{dk} \) standing for dont-know-choose. For rewrite rules \( \ell_1, \ldots, \ell_n \), the strategy \( \mathbf{dc}(\ell_1, \ldots, \ell_n) \) returns the results of one non-failing rule, un-deterministically chosen among the \( \ell_i \). On the contrary, for the strategy \( \mathbf{dk}(\ell_1, \ldots, \ell_n) \), all possible results are computed and returned. This is implemented by backtracking on all rules \( \ell_1, \ldots, \ell_n \), \( \mathbf{dc}(s_1, \ldots, s_n) \) 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 \( \mathbf{dc}(s_1, \ldots, s_n) \). If all sub-strategies fail, then it fails too. \( \mathbf{dk}(s_1, \ldots, s_n) \) returns all results of all sub-strategies \( s_i \). The identity strategy \( \mathbf{id} \) does not change a term, while the strategy \( \mathbf{fail} \) 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 \( T(\mathcal{F} \cup \mathcal{L} \cup \{ ;, \mathbf{dc}, \mathbf{dk}, \mathbf{first}, \mathbf{id}, \mathbf{fail} \}) \). In addition to \( \mathbf{dc}, \mathbf{dk}, \mathbf{id} \)
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; (s1, . . . , sn) corresponds to an application of a rewrite rule labeled by t 2 L, which also applies sub-strategies
esi on values of variables xi of the rewrite rule, after matching and before replacing the matched
subterm by the instantiated right-hand side of the rewrite rule; f(s1, . . . , sn) is an application of
sub-strategies esi to subterms ti of the term f(t1, . . . , tn) with root f 2 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 [de(s1, s2)]t, that can apply simultaneously. So we can obtain as
many results as de 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 ws
as follows:

\[
\begin{align*}
    f(ws_1, \ldots, ws_n) &= \{ f(e_1, \ldots, e_n) \mid e_i \in ws_i, \ i = 1 \ldots n \}, \text{ and} \\
    t(ws_1, \ldots, ws_n) &= \{ \{ x_i \mapsto e_i \}(t) \mid x_i \in Var(t), \ e_i \in ws_i, \ i = 1 \ldots n \},
\end{align*}
\]

where \( x_i \mapsto e_i \) stands for the 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. That
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 \( \in dom(\_ \downarrow) \) is similarly described by a rewrite theory in [3]. The symbol
\( \ll\\downarrow \) 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:

\[
map(s) \Rightarrow de(nil, s \cdot map(s))
\]

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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>[DK]</td>
<td>( dk(es_1, es_2) ) t ( \Rightarrow ) ([es_1] \cup [es_2] ) t</td>
</tr>
<tr>
<td>[DC1]</td>
<td>( dc(es_1, es_2) ) t ( \Rightarrow { [es_1] \mid t \in dom(es_1) } \downarrow )</td>
</tr>
<tr>
<td>[DC2]</td>
<td>( dc(es_1, es_2) ) t ( \Rightarrow { [es_2] \mid t \in dom(es_2) } \downarrow )</td>
</tr>
<tr>
<td>[DC3]</td>
<td>( dc(es_1, es_2) ) t ( \Rightarrow [es_1, es_2 ] ) t</td>
</tr>
<tr>
<td>[FIRST1]</td>
<td>( f(st(es_1, es_2)) ) t ( \Rightarrow { [es_1 \mid t \in dom(es_1) } \downarrow )</td>
</tr>
<tr>
<td>[FIRST2]</td>
<td>( f(st(es_1, es_2)) ) t ( \Rightarrow { [es_2 \mid t \in dom(es_2) } \downarrow )</td>
</tr>
<tr>
<td>[ID]</td>
<td>( id(t) ) ( \Rightarrow { t } )</td>
</tr>
<tr>
<td>[FAIL]</td>
<td>( fail(t) ) ( \Rightarrow { } )</td>
</tr>
<tr>
<td>[FSYM1]</td>
<td>( f(es_1, \ldots, es_n) f(x_1, \ldots, x_n) \Rightarrow f([es_1] x_1, \ldots, [es_n] x_n) )</td>
</tr>
<tr>
<td>[FSYM2]</td>
<td>( g(es_1, \ldots, es_n) g(x_1, \ldots, x_m) \Rightarrow { } ) if ( f \neq g )</td>
</tr>
<tr>
<td>[LAB1]</td>
<td>( u(es_1, \ldots, es_n) u(x_1, \ldots, x_n) \Rightarrow (t_1, \ldots, [es_1] x_1, \ldots) )</td>
</tr>
<tr>
<td>[LAB2]</td>
<td>( u(es_1, \ldots, es_n) u(x_1, \ldots, x_n) \Rightarrow { } ) if ( u \ll\downarrow ) where ( t \Rightarrow t' \in R )</td>
</tr>
<tr>
<td>[CONC]</td>
<td>( { es_1, es_2 } \Rightarrow { [es_2] \cup [es_1] } ) t</td>
</tr>
</tbody>
</table>

Figure 1: The interpreter of elementary strategies
(i.e. \( t \) should be a non-empty list) and \( \text{map} \ (s) \) 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:

\[
\begin{align*}
\text{map} \ (s) \ nil & \Rightarrow \ nil \\
\text{map} \ (s) \ a.\ as & \Rightarrow \ [s]a.\ [\text{map} \ (s)] \ as
\end{align*}
\]

The difference relies on the fact that the list, which the functional \( \text{map} \) 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:

\[
\begin{align*}
\text{map} \ (s_1) \ ; \ \text{map} \ (s_2) & \Rightarrow \ \text{map} \ (s_1 \ ; \ s_2) \\
\text{map} \ (\text{id}) & \Rightarrow \ \text{id}
\end{align*}
\]

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 \( \text{nil} \), \( \text{id} \), representing constructors of elementary strategies, and symbols \( \text{nil} \), \( \text{id} \), which are constructors of the sort \( \text{list}[X] \). The operational semantics of \( s_1 \ ; \ s_2 \) is to apply the strategy \( s_1 \) to the head, and \( s_2 \) to the tail of a non-empty list, while the symbol \( \text{id} \) 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: \( \text{nil} : \langle \text{list}[X] \rightarrow \text{list}[X] \rangle \), resp. \( \text{id} : \langle X \rightarrow X \rangle \langle \text{list}[X] \rightarrow \text{list}[X] \rangle \) \( \langle \text{list}[X] \rightarrow \text{list}[X] \rangle \), and \( \text{map} \ (\_) : \langle X \rightarrow X \rangle \langle \text{list}[X] \rightarrow \text{list}[X] \rangle \). 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 \( \lambda \)-terms using the leftmost innermost and the leftmost outermost strategies. \( \lambda \)-terms are represented using de Bruijn notation [6], and we suppose that there are two primitive operations: \( \text{free}(v, t) \) meaning that a variable \( v \) is free in a \( \lambda \)-term \( t \), and \( \text{replace}(v, a, t) \) standing for \( \{ v \mapsto a \} \{ t \} \). The only difference with traditional definitions of \( \text{free} \) and \( \text{replace} \) is that, in the de Bruijn calculus, these functions must increment the index \( v \) when passing through \( \lambda \) construction, i.e. \( \text{replace}(v, \lambda N, M) \Rightarrow \lambda \text{replace}(v+1, N, M) \). The definitions of \( \lambda \)-terms and \( \text{beta} \) and \( \text{eta} \) transition rules are straightforward in ELAN (see Figure 2). However, the definition of two normalization strategies \( \text{lis}(s) \) and \( \text{los}(s) \) is less evident (see Figure 3). These strategies \( \text{lis}(s) \) and \( \text{los}(s) \) fail if the input \( \lambda \)-term does not contain any \( s \)-redex reducible with the sub-strategy \( s \) (later instantiated to \( \text{de}(\text{beta}, \text{eta}) \)). The strategy \( \text{lis}(s) \) tries to apply the sub-strategy \( \langle \text{lis}(s) \rangle \ id \), which succeeds, if the input term has a form \( \langle M \ N \rangle \) and \( \text{lis}(s) \) is applicable on \( M \), i.e. \( M \) contains \( s \)-redex. Otherwise, it continues on the right \( \lambda \)-subterm \( N \) by the application of \( \langle \text{id} \langle \text{lis}(s) \rangle \rangle \). If the input term has a form \( \lambda M \), the strategy \( \text{lis}(s) \) is propagated towards \( M \) by the application of \( \text{la} \langle \text{lis}(s) \rangle \). If none of the three cases above succeeds, the strategy \( s \) is applied on the top of this \( \lambda \)-term. The fact, that the sub-strategy \( s \) is applied last, makes the crucial difference between strategies \( \text{lis}(s) \) and \( \text{los}(s) \).

We use the Church’s numbers defined as \( \text{zero} = \lambda \ \lambda \ 1 \) and \( \text{succ} = \lambda \ \lambda \ (2 \ ((3 \ 2) \ 1)) \) to illustrate normalization of \( \lambda \)-terms. Iterating the leftmost innermost strategy \( \text{lis}(\text{de}(\text{beta}, \text{eta})) \) on \( \text{two} = \langle \text{succ} \langle \text{succ} \text{zero} \rangle \rangle \), we get:

\( \langle \lambda \lambda \lambda (2 ((32)1)) \lambda \lambda (2 ((1 \lambda 12)1)) \rangle \Rightarrow \langle \lambda \lambda \lambda (2 ((32)1)) \lambda \lambda (2 (\lambda 11)) \rangle \Rightarrow \langle \lambda \lambda \lambda (2 ((32)1)) \lambda \lambda (2) \rangle \Rightarrow \lambda \lambda (2 (\lambda (21) 2) 1)) \Rightarrow \lambda \lambda (2 (\lambda (21) 2 1)) \Rightarrow \lambda \lambda (2 (\lambda (21) 2 1)) \Rightarrow \lambda \lambda (2 (\lambda (21) 2 1)) \Rightarrow \lambda \lambda (2 (\lambda (21) 2 1)) \Rightarrow \lambda \lambda (2 (\lambda (21) 2 1)) \), 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: \(\lambda\)-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:
\[
\lambda \lambda (2((\lambda \lambda (2((32)1)1)1)1)1)11 = \lambda \lambda (2(\lambda (2(\lambda (2((\lambda (22)1)1)1)1)1)1)1)11 = \lambda \lambda (2(\lambda (22)1)1)11 = \lambda \lambda (2(2(\lambda 1)(1))) = \lambda \lambda (2(21)).
\]

The definition of normalization strategies can be generalized from \(\lambda\)-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 \((S \to S) S\). 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 FSYM1, FSYM2 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(FSYM1,FSYM2) trying them in this order. This results in a more efficient interpreter described in Figure 4.
Strategies of ELAN: Meta-interpretation and Partial Evaluation

rules for X
a, b, c : X; s, sl, s2 : <X->X>; ssl : list(<X->X>); cond : bool;
global
[DK1] [dk(s1, ssl1) a] => b where b:=(eval)[s1]a
[DK2] [dk(s1, ssl1) a] => b where b:=(eval)[dk(ssl1)]a
[DC1] [dc(s1, ssl1) a] => b where b:=(eval)[dc(ssl1)]a
[DC2] [dc(s1, ssl1) a] => b where b:=(eval)[s1]a
[IF1] [if true then s1 else s2 fi] a => b where b:=(eval)[s1]a
[IF2] [if false then s1 else s2 fi] a => b where b:=(eval)[s2]a
[IFO1] [if s then s1 orelse s2 fi] a => [s1] b where b:=(eval)[s1]a
[IFO2] [if s then s1 orelse s2 fi] a => [s2] a
[IFO3] [s1] a => b where b:=(eval)[s1]a
[CONC] [s1 ; s2] a => c where b:=(eval)[s1]a where c:=(eval)[s2]b
[ID] [id] a => a
[CONST] [a] a => a
[FIN] b => b where cond:=(dc(NF1, NF2)) is_nf(b) if cond
end

rules for bool
a : X; b : X; s : <X->X>;
local
[NF1] is_nf([s]a) => false
[NF2] is_nf(b) => true
end

strategies for X // evaluation ELAN-strategy of the interpreter

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 T(F), or not. A boolean symbol is_nf applied to b is reduced due to the strategy dc(NF1, NF2) to true, if b ∈ T(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).

Theory and Practice of Algebraic Specifications ASF+SDF’97
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 \( f(\varepsilon s_1, \ldots, \varepsilon s_n) \). For any symbol \( f : (s_1 \ldots s_n) \to s \), a rule FSYM is generated:
  \[
  [\text{FSYM}] \quad [f(s_1, \ldots, s_n)] f(x_1, \ldots, x_n) \Rightarrow f(y_1, \ldots, y_n) \quad \text{where} \quad y_i := (\text{eval}[s_i] x_i, i = 1..n
  \]

- **LAB** rules interpret the elementary strategy \( l(\varepsilon s_1, \ldots, \varepsilon s_n) \). For any rewrite rule from \( R \) of the form:
  \[
  [l(x_1, \ldots, x_n)] u \Rightarrow u'
  \]
  where \( x_i \in Var(u) \) for \( 1 \leq i \leq n \), a rule LAB is generated:
  \[
  [\text{LAB}] \quad [l(s_1, \ldots, s_n)] u(x_1, \ldots, x_n) \Rightarrow u'(y_1, \ldots, y_n)
  \quad \text{where} \quad y_i := (\text{eval}[s_i] x_i, i = 1..n
  \]

- **DSTR** rules interpret user-defined strategies. For any implicit definition of a user-defined strategy \( d \) of the form \( d(s_1, \ldots, s_n) \Rightarrow s \), a rule DSTR is produced:
  \[
  [\text{DSTR}] \quad [d(s_1, \ldots, s_n)] x \Rightarrow y \quad \text{where} \quad y := (\text{eval}[s] x
  \]

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

- \( \text{nil} \) : \( \langle \text{list}[X] \rangle \)
- \( \cdot \) : \( \langle X \rightarrow X \rangle \langle \text{list}[X] \rightarrow \text{list}[X] \rangle \langle \text{list}[X] \rightarrow \text{list}[X] \rangle \)
- \( l(\cdot \cdot) \) : \( \langle X \rightarrow X \rangle X \langle X \rightarrow X \rangle \)

which are interpreted by the following rules:

- \[
  [\text{FSYM}] \quad [\text{nil}] \text{nil} \Rightarrow \text{nil}
  \]
- \[
  [\text{FSYM}] \quad [s_1 \cdot s_2] x_1, x_2 \Rightarrow y_1, y_2 \quad \text{where} \quad y_1 := (\text{eval}[s_1] x_1 \text{ where} \quad y_2 := (\text{eval}[s_2] x_2
  \]
- \[
  [\text{LAB}] \quad [l(s)] x + 0 \Rightarrow y + y \quad \text{where} \quad y := (\text{eval}[s] x
  \]

Moreover, the following rule with the label DSTR is added for the interpretation of defined strategy \( \text{map} : \)
\[
[D\text{STR}] \quad [\text{map}(s)] x \Rightarrow y \quad \text{where} \quad y := (\text{eval}[\text{dc}(\text{nil}), s \cdot \text{map}(s)]) x
  \]

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 \( \text{where} \quad b := (\text{eval}[\text{dc}(\text{ss1})] a, \) the strategy \( \text{eval} \) is invoked to reduce the application term \( [\text{dc}(\text{ss1})] a \). From the structure of the strategy \( \text{eval} \) and labeled rules referred in it, it is clear, that only rules DC1 and DC2 can be used for reduction of \( [\text{dc}(\text{ss1})] a \). Thus, instead of applying the complex strategy \( \text{eval} \) in this assignment, a simpler strategy \( \text{eval}_\text{dc} = \text{dc}(\text{dk}(\text{DC1}), \text{dk}(\text{DC2})); \text{dc}(\text{fn}) \) can be used, i.e. one can write
\[
\text{where} \quad b := (\text{eval}_\text{dc})[\text{dc}(\text{ss1})] a.
\]

When the application of the strategy \( \text{eval}_\text{dc} \) is performed, the term \( [\text{dc}(\text{ss1})] a \) is matched with left-hand sides of the rules DC1 and DC2 used in the strategy \( \text{eval}_\text{dc} \), i.e. with the terms of the form \( [\text{dc}(s), \text{ss1}] a \). Since the symbols \( \cdot \cdot \) and \( \text{dc} \) are present in both terms, two decomposition steps of the matching algorithm made at run-time can be saved thanks to partial evaluation:
\[
[\text{dc}(s'), \text{ss1}'] a \leftrightarrow \text{dc}(s', \text{ss1}') \leftrightarrow \text{dc}(\text{ss1}) a \text{ and } a' \leftrightarrow a,
\]
and then
to $s'_1, ss'_1 \ll s'_{ss'_1}$ and $a' \ll a$, where the variables $s_1, ss_1, a$ on left-hand sides have been renamed to $s'_1, ss'_1, a'$. All components of the decomposed matching problem can be regrouped into a term using a new auxiliary function symbol $(.)_a : ([X 	o X] \times X) \to X$, such that the matching of the initial terms is at compile-time reduced to the matching of $(s'_1, ss'_1) \ll (ss_1) a$. Then, the rules DC1 and DC2 can be transformed to the following form:

$$\text{[DC1]} \quad ((s_1, ss_1))a \Rightarrow b \quad \text{where} \quad b := (eval)[s_1]a$$

$$\text{[DC2]} \quad ((s_1, ss_1))a \Rightarrow b \quad \text{where} \quad b := (eval_{dc})(ss_1) a$$

and the following rule makes a simple interface between the previous term $[dc(ss_1)]a$ and the new one $(ss_1) a$:

$$\text{[DC]} \quad [dc(ss_1)]a \Rightarrow b \quad \text{where} \quad b := (eval_{dc})(ss_1) a$$

In this transformation, the term $(ss_1) a$ in the where statement is slightly more compact than the original one $[dc(ss_1)]a$ 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 general framework of partial evaluation.

![Figure 5: The optimized kernel](image-url)

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 $T(\mathcal{F}, \lambda')$, not only with ground input terms from $T(\mathcal{F})$, 
- 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 $map$:

$$map(s) \Rightarrow dc(ni1, cons(s, map(s)))$$

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

$$[DSTR] \quad [map(s)]t \Rightarrow y \quad where \quad y := (eval)[dc(ni1, s \cdot map(s))][t]$$

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 $[dc(ni1, s \cdot map(s))][t]$ 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 $dc(dk(DC)); dc(FIN) = dk(DC); dc(FIN)$. 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 $dc(ni1, cons(s, map(s)))$ 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 $eval' = dk(DC'); dc(FIN)$ applied to the term $[dc(ni1, cons(s, map(s)))]$. This new partial strategy refers to an optimized and instantiated version $DC'$ of the original rule:

$$[DC'] \quad [dc(ni1, s \cdot map(s))][t] \Rightarrow y \quad where \quad y := (eval_{dc})(ni1, s \cdot map(s))[t]$$

and the strategy $eval$ in the original rule can be replaced by $eval'$. The rule $FIN$ following in the strategy $eval'$ the strategy $dk(DC')$ 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 dc(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 • 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(dc(DC'), dc(DC2')); dc(FIN) refers to new rewrite rules DC1' and DC2':

\[
\begin{align*}
[DC1'] \quad ((\text{nil}, s \cdot \text{map}(s))| t) & \Rightarrow y \quad \text{where } y := (\text{eval}[\text{nil}]| t) \\
[DC2'] \quad ((\text{nil}, s \cdot \text{map}(s))| t) & \Rightarrow y \quad \text{where } y := (\text{eval dc}((s \cdot \text{map}(s))| t)
\end{align*}
\]

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 dc, dc, id, :, ;, \ell, and s, where \ell 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 T(F, X), which are called patterns. A pattern p is a finite representation (or schematization) of its ground intances, obtained by instantiating its variables by terms of T(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 T(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

\[ s \circ \{p_1, \ldots, p_n\} \Rightarrow s' \circ \{q_1, \ldots, q_m\} \]

where \{p_1, \ldots, p_n\} and \{q_1, \ldots, 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, \ldots, 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 \( \in T(F) \), the method consists first in building s' and O such that s \( \circ \{t\} \Rightarrow s' \circ O \), according to rules in Figure 6. The obtained strategy s' is simplified by several primitive transformations of strategies, e.g. \( dc(s) \Rightarrow s \), \( dc(s) \Rightarrow s, dc(s) \Rightarrow s, dc(s) \Rightarrow s \), \( dc(s) \Rightarrow \text{fail} \), \( dc(s) \Rightarrow \text{fail} \), 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 \( \text{fail} \).

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 dc(s_1, \ldots, s_n) or dc(s_1, \ldots, s_n) is obtained from specializations of each
case of concatenation \( s_1 \cdot s_2 \)

\[
s_1 \circ I \gg s'_1 \circ Q \text{ and } s_2 \circ Q \gg s'_2 \circ O \\
s_1 \cdot s_2 \circ I \gg s'_1 \cdot s'_2 \circ O
\]

case of \( \text{clk}, \text{resp. dc}, \text{over strategies} \)

\[
\text{clk}(s_1, \ldots, s_n), \text{resp. dc}(s_1, \ldots, s_n)
\]

\[
s_j \circ I \gg s'_j \circ Q_j \\
\text{clk}(s_1, \ldots, s_n) \circ I \gg \text{clk}(s'_1, \ldots, s'_n) \circ \bigcup_{j=1}^n Q_j
\]

case of \( \text{id}, \text{resp. fail} \)

\[
\text{id} \circ I \gg \text{id} \circ I \\
\text{fail} \circ I \gg \text{fail} \emptyset
\]

case of \( \text{clk}, \text{resp. dc}, \text{over rules} \)

\[
\text{clk}(r_1, \ldots, r_n), \text{resp. dc}(r_1, \ldots, r_n)
\]

\[
\text{clk}(r_1, \ldots, r_n) \circ \{p_1, \ldots, p_m\} \gg \text{clk}(r'_1, \ldots, r'_n) \circ \{q_{i,j}\} \quad (i = 1..m, j = 1..n)
\]

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
## 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


**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: FSYM$^4_4$. 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 [DSTR$^2_2$], the strategy eval is applied on a redex [def nil, $x_1$ $\bullet$ map]$^2_2$($x_2$)$]x_0$

[DSTR$^2_2$] [map]$^2_2$($x_1$)$]x_0 \Rightarrow x_0$ where $x_0 := (eval)$[def nil, $x_1$ $\bullet$ map]$^2_2$($x_2$)$]x_0$

is replaced by eval$^{(6)}_0 = DC^{(0)}_0$[def FIN] and a new rule DC$^{(0)}_0$ is generated:

[def nil, $x_0$ $\bullet$ map]$^{(6)}_0$($x_1$)$]x_1 \Rightarrow x_2$ where $x_2 := (eval_1^0)$(def nil, $x_0$ $\bullet$ map)$^{(6)}_0$($x_0$)$]x_1$
2. In the rule \([DC(0)]\), the strategy \(eval \cdot dc\) is applied on a redex \(\{\text{nil}, x_0 \cdot \text{map}(x_0)\}\).\(x_1\):
\[
[DC(0)] [\text{dc}\{\text{nil}, x_0 \cdot \text{map}(x_0)\}]x_1 \Rightarrow x_2 \text{ where } x_2 \equiv (eval \cdot dc)(\{\text{nil}, x_0 \cdot \text{map}(x_0)\})x_1
\]
is replaced by
\[
(eval \cdot dc) = \text{dc}(\text{dk}(DC(1)), \text{dc}(DC(2))); \text{dc}(\text{FIN}) \text{ and new rules are generated:}
\]
\[
[DC(1)] (\{\text{nil}, x_0 \cdot \text{map}(x_0)\})x_1 \Rightarrow x_2 \text{ where } x_2 \equiv (eval \cdot dc)(\{\text{nil} \cdot x_0 \cdot \text{map}(x_0)\})x_1
\]
\[
[DC(2)] (\{\text{nil}, x_0 \cdot \text{map}(x_0)\})x_1 \Rightarrow x_2 \text{ where } x_2 \equiv (eval \cdot dc)(\{\text{nil} \cdot x_0 \cdot \text{map}(x_0)\})x_1
\]
3. In the rule \([DC(1)]\), the strategy \(eval \cdot dc\) is applied on a redex \(\{x_0 \cdot \text{map}(x_0)\}\):\(x_1\):
\[
[DC(1)] (\{x_0 \cdot \text{map}(x_0)\})x_1 \Rightarrow x_2 \text{ where } x_2 \equiv (eval \cdot dc)(\{x_0 \cdot \text{map}(x_0)\})x_1
\]
is replaced by
\[
(eval \cdot dc) = \text{dc}(\text{dk}(DC(1)), \text{dc}(DC(2))); \text{dc}(\text{FIN}) \text{ and new rules are generated:}
\]
\[
[DC(1)] (\{x_0 \cdot \text{map}(x_0)\})x_1 \Rightarrow x_2 \text{ where } x_2 \equiv (eval \cdot dc)(\{x_0 \cdot \text{map}(x_0)\})x_1
\]
\[
[DC(2)] (\{x_0 \cdot \text{map}(x_0)\})x_1 \Rightarrow x_2 \text{ where } x_2 \equiv (eval \cdot dc)(\{x_0 \cdot \text{map}(x_0)\})x_1
\]
4. In the rule \([DC(1)]\), the strategy \(eval \cdot dc\) is applied on a redex \(\{\text{nil}\}\):
\[
[DC(1)] (\{\text{nil}\})x_1 \Rightarrow x_2 \text{ where } x_2 \equiv (eval \cdot dc)(\{\text{nil}\})x_1
\]
is replaced by
\[
(eval \cdot dc) = \text{dk}(\text{FSYM}_2); \text{dc}(\text{FIN}) \text{ and a new rule } \text{FIN}^{(0)} \text{ is generated:}
\]
\[
\text{nil} \Rightarrow \text{nil} \text{ where } x_0 \equiv (\text{dc}(\text{NF1, NF2})); isomega f(\text{nil}) \text{ if } x_0
\]
5. In the rule \([DC(2)]\), the strategy \(eval \cdot dc\) is applied on a redex \(\{x_0 \cdot \text{map}(x_0)\}\):
\[
[DC(2)] (\{x_0 \cdot \text{map}(x_0)\})x_1 \Rightarrow x_2 \text{ where } x_2 \equiv (eval \cdot dc)(\{x_0 \cdot \text{map}(x_0)\})x_1
\]
is replaced by
\[
(eval \cdot dc) = \text{dk}(\text{FSYM}_2^{(1)}); \text{dc}(\text{FIN}) \text{ and new rules are generated:}
\]
\[
[FSYM_2^{(1)}] x_0 \cdot \text{map}(x_0)x_1 \Rightarrow x_2 \text{ where } x_2 \equiv (eval \cdot dc)(\{x_0 \cdot \text{map}(x_0)\})x_1
\]
\[
[FIN^{(1)}] x_0 \cdot x_1 \Rightarrow x_0 \cdot x_1 \text{ where } x_2 \equiv (\text{dc}(\text{NF1, NF2})); isomega f(x_0 \cdot x_1) \text{ if } x_3
\]
6. In the rule \([DC(2)]\), the strategy \(eval \cdot dc\) is applied on a redex \(\{()\}\):
\[
[DC(2)] (\{()\})x_1 \Rightarrow x_2 \text{ where } x_2 \equiv (eval \cdot dc)(\{()\})x_1
\]
is replaced by \(\text{fail}\).
7. In the rule \([FSYM_2^{(1)}]\), the strategy \(eval \cdot dc\) is applied on a redex \(\{x_0\}\):
\[
[FSYM_2^{(1)}] x_0 \cdot \text{map}(x_0)x_1 \Rightarrow x_2 \text{ where } x_2 \equiv (eval \cdot dc)(\{x_0\})x_1
\]
is replaced by
\[
(eval \cdot dc) = \text{dk}(\text{DSTR}_2); \text{dc}(\text{FIN}) \text{ and no new rule is generated.}
\]
8. In the rule \([FIN^{(1)}]\), the strategy \(\text{dc}(\text{NF1, NF2})\) is applied on a redex \(isomega f(\text{nil})\):
\[
[FIN^{(1)}] \text{nil} \Rightarrow \text{nil} \text{ where } x_0 \equiv (\text{dc}(\text{NF1, NF2})); isomega f(\text{nil}) \text{ if } x_0
\]
is replaced by \(\text{dc}(\text{NF2}^{(0)})\) \(\) and a new rule \(\text{NF2}^{(0)}\) \(\) is generated:
\[
isomega f(\text{nil}) \Rightarrow true
\]
9. In the rule \([FIN^{(1)}]\), the strategy \(\text{dc}(\text{NF1, NF2})\) is applied on a redex \(isomega f(x_0 \cdot x_1)\):
\[
[FIN^{(1)}] x_0 \cdot x_1 \Rightarrow x_0 \cdot x_1 \text{ where } x_2 \equiv (\text{dc}(\text{NF1, NF2})); isomega f(x_0 \cdot x_1) \text{ if } x_3
\]
is replaced by \(\text{dc}(\text{NF2}^{(1)})\) \(\) and a new rule \(\text{NF2}^{(1)}\) \(\) is generated:
\[
isomega f(x_0 \cdot x_1) \Rightarrow true
\]

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.

---

Theory and Practice of Algebraic Specifications ASF+SDF’97 14
Figure 8: Dependency tree