Controlling Rewriting: study and implementation of a strategy formalism

(Abstract)

Peter Borovanský

LORIA-INRIA
615, rue du Jardin Botanique, BP 101,
54602 Villers-lès-Nancy Cedex, France
e-mail: Peter.Borovansky@loria.fr

Abstract

This paper summarizes my PhD thesis devoted to an introduction of a new strategy formalism for the first-order rewrite system, called ELAN. Goals of my PhD thesis are proposing and studying different constructions expressing the control of rewriting at the level of rules and strategies, studying a strategy-directed cooperation of procedures (i.e. solvers), and finally, exploring certain reflexive aspects of computational systems to be able to express their transformations by computational systems. The principal goal is a design of a declarative, strictly typed and extensible strategy language based on rewriting logic within the existing framework ELAN. A programming style of the strategy language, different language constructions and extensions (e.g. high-level or polymorphic strategies) and several used implementation techniques (e.g. partial evaluation, or compilation) are also studied in this thesis. This paper outlines the principal problems attacked in this thesis, highlights several new ideas and proposed solutions.

Keywords: Rewriting Logic, Computational Systems, Control, Strategies, Programming Language, Partial Evaluation, Compilation.

1 Introduction

Since the last three decades, rewriting has been studied as an execution method of formal specifications. Different specification systems are usually based on the first-order logic, e.g. some variation of the equational logic.

[1] This work has been partially supported by the Esprit Basic Research Working Group 22457 - Construction of Computational Logics II.

Specifications are usually considered as equational programs and interpreted by rewriting, whose theoretical model is represented by *rewriting logic* introduced by J. Meseguer [10].

The interpretation tools of rewrite systems have to be expressive and efficient enough for prototyping more complex specifications. A stimulating problem of the acceleration of these tools is attacked by several research teams. The increasing expressiveness of the specification tools, e.g. by placing them to a context of the order-sorted or high-order logic, can be an interesting feature, when these tools are used as programming environments. However, it often decreases their efficiency. The speed-up of these specification tools can be easily measured, however, new language features and constructions introduced in order to increase their expressiveness have to be *based on* a semantics coherent to the initial one, *motivated by* the user's needs, and also *justified by* their efficient implementation. Following this philosophy, results presented in [1] propose language enriching constructions for controlling rewrite and computational systems in a declarative way. Efficiency aspects of these constructions are always considered even if it is not a primary goal of this thesis.

We are interesting in rewriting as a computational paradigm and its corresponding logic. In this framework, we study different possibilities of controlling rewrite derivations. We propose conditional rewrite rules with structured conditions, which are flexible enough to specify simple or structured conditions, local variable or pattern assignments. Having rewrite rules identified by their names, the control of rewriting can be declaratively expressed by specifying a set of all acceptable sequences of labeled rewrite rules, or eventually, their instances. A proof calculus of rewriting logic representing the theoretical model of rewriting, offers the possibility to generalize these sequences of rewrite rules to proof terms specifying *which* instance of *which* rewrite rule is applied on *which* term. The study of a language describing sets of proof terms, also called strategies, is the main contribution of the thesis [1]. Different aspects of this *strategy language* are considered: its semantics, programming paradigm and implementation techniques.

The principal features of the proposed strategy language are the semantics based on rewriting logic, a possibility to express non-deterministic and concurrent derivations, strict typing of all language constructions in a many-sorted signature and its extensibility by new constructions. The strategy language is defined as a transformation of a rewrite theory associated with several strategy definitions, called *computational system* [9], to a strategy theory, which is another rewrite theory. The construction of this transformation allows to build-up a hierarchy of strategies, where some high-order strategies can control derivations of the other ones. Two implementation methods are proposed and studied: the meta-interpretation optimized by partial evaluation and the compilation of strategies.

The research presented in [1] has been realized in a context of an existing framework for prototyping of solvers and programming languages, called

ELAN. This system was designed in PROTHEO team (cf. the overview of ELAN in [4]) and mainly implemented by M. Vittek [12]. It is used for specification of constraint solvers over different domains (e.g. syntactic theories, finite domains, etc.), for prototyping different logical frameworks (e.g. logic programming, CLP, etc.), and for studying different properties of equational and rewrite systems (e.g. termination, completion, etc.). Because of the existence of a lot of applications developed during its five years history, one of the main priorities of the research is to integrate elegantly and coherently our new ideas and constructions to ELAN.

The strategy language of Maude [8], which represents a related approach, is based on similar motivations. However, it is built on different basic principles, and thus, it has a different internal philosophy and architecture. It is based on the *reflexivity* of rewriting logic interconnecting a specification of an object theory with a specification of strategies. Maude offers the possibility to create more complicated high-order strategies due to a *reflexive tower* of rewriting. In our approach, the strategy language is a syntactic transformation of an object theory enriched by several specifications of strategies into a strategy theory. By iteration of this construction, we can also create more complicated strategies, however, our *tower of strategies* is created differently.

The structure of this paper is the following: Section 2 overviews the several contributions of the thesis [1], while Section 3 illustrates more precisely several ideas of its principal contribution concerning the strategy language. Section 4 presents several perspectives for the further research.

2 Control of rewriting

The process of rewriting can be controlled at several levels, and therefore, several different controlling formalisms can be introduced. The control of the normalization process by *local evaluation strategies* attached to function symbols of a signature is one example of such an approach introduced in the system OBJ-2. We have studied the problem of the control of rewrite derivations at three different formalization levels:

- **Rewrite rules** – The control can be directly expressed in the rules of a rewrite system in a form of boolean conditions, structured conditions, local (matching) assignments, factorization of common parts, etc. At this level, we can naturally express simple controlling notions, e.g. an application order of rules, or elimination of repeating sub-derivations of common parts of these rules, etc.
- **Strategies** – In order to construct more complex controlling procedures, we introduce a separate formalism of strategies, called strategy language, expressing in a more fine way where, when and which rewrite rule is ought to be applied under which conditions. The first concept of a non-deterministic strategy language for rewriting was introduced in the Vittek's thesis [12].

Its strategy constructions correspond to regular expressions over the rule names. More sophisticated strategy languages have been recently proposed and/or developed for the systems Maude [7] and ASF+SDF [11]. The strategy language of ELAN is another example of a domain-oriented language allowing to specify non-deterministic, recursive, parameterized and well-typed strategies in a many-sorted logic.

- **Cooperation of solvers** – The most typical ELAN applications are various (constraint) solvers, which are usually very complex procedures. The possibility to express their cooperation and concurrent execution is an example how the solvers can be controlled at the level of applications. As an example, *concurrent strategies* offer the possibility to control several sub-solvers in a declarative way. In a more general framework, these complex solvers (i.e. whole procedures) are even not necessarily specified in ELAN.

Rewrite rules

In this section, we illustrate some of proposed controlling constructions at the level of rewrite rules. We briefly introduce the syntax of simple ELAN rules, which are (labeled) conditional rewrite rules with local variable assignments:

$$[\ell] \quad l \Rightarrow r \ [\ \textbf{if} \quad v \mid \ \textbf{where} \quad y{:=}(S)u \ ]^*$$

where ℓ is a label of this rule (eventually empty), the terms l and r are, respectively, the left and the right-hand side of this rule, v is a boolean condition, and $y{:=}(S)u$ is a local assignment of all results of a strategy S applied on a term u to a local variable y. The labeled version of this rewrite rule is always applied on the *top* of a term t such that its left-hand side l is first matched against t, then all expressions introduced by **where** and **if** constructions are instantiated by the corresponding matching substitution. Each instantiation of a local variable extends the matching substitution, and a newly instantiated local variable, e.g. y, may occur in the following expressions. Therefore, these expressions are evaluated in textual order. When all conditions are satisfied and local assignments realized, the replacement of the matched term by the fully instantiated right-hand side is performed.

A local variable assignment may invoke a strategy-directed sub-derivation simplifying a term u. If there is no specified strategy in a local assignment, the default leftmost-innermost normalization strategy of ELAN is applied. This normalization strategy is parameterized by a set of unlabeled rewrite rules (i.e. when ℓ is empty) and it also respects local evaluation strategies associated to the function symbols of a signature. Labels of rewrite rules allow to refer these rules in the user defined strategies.

As it was mentioned before, the first possibility is to express a controlling mechanism directly in the rewrite rules. Existing tools based on rewriting usually offer only conditional rewrite rules, whose boolean conditions can con-

trol their applications. Either a *factorization* of common parts of conditional rewrite rules, or a *construction of rules with structured conditions* collecting several rewrite rules having the same labels and the left-hand sides, are two examples of the control expressed directly in rewrite rules. The following figure illustrates both these constructions on small examples [2] . While the factorization (on the left) eliminates repetitions of common sub-derivations, the structured condition (on the right) specifies an application order of rules.

$[\ell]\ l \Rightarrow r(l, y_1, y_3)$
where $y_1{:=}(S_1)u_1$
choose
try **if** v_2'
try **if** v_2''
end
where $y_3{:=}(S_3)u_3$

$[\ell]\ l \Rightarrow$
where $y_1{:=}(S_1)u_1$
switch
case r_1 **if** v_2' $y_3{:=}(S_3)u_3$
case r_2 **if** v_2'' $y_3{:=}(S_3)u_3$
otherwise r_3
end

The previous factorized rule can be unfolded as follows:

$$[\ell]\ l \Rightarrow r \ \ \textbf{where}\ y_1{:=}(S_1)u_1 \ \ \textbf{if}\ v_2' \ \ \textbf{where}\ y_3{:=}(S_3)u_3$$
$$[\ell]\ l \Rightarrow r \ \ \textbf{where}\ y_1{:=}(S_1)u_1 \ \ \textbf{if}\ v_2'' \ \ \textbf{where}\ y_3{:=}(S_3)u_3$$

while the rule with the structured condition represents the following rules:

$$[\ell]\ l \Rightarrow r_1 \ \ \textbf{where}\ y_1{:=}(S_1)u_1 \ \ \textbf{if}\ v_2' \qquad\qquad \textbf{where}\ y_3{:=}(S_3)u_3$$
$$[\ell]\ l \Rightarrow r_2 \ \ \textbf{where}\ y_1{:=}(S_1)u_1 \ \ \textbf{if}\ \mathit{not}\ v_2' \ \ \textbf{if}\ v_2'' \ \ \textbf{where}\ y_3{:=}(S_3)u_3$$
$$[\ell]\ l \Rightarrow r_3 \ \ \textbf{where}\ y_1{:=}(S_1)u_1 \ \ \textbf{if}\ \mathit{not}\ v_2' \ \ \textbf{if}\ \mathit{not}\ v_2''$$

Several common parts of an application of the unfolded rules above, e.g. matching of the left-hand side l and the assignment of results of the strategy application S_1 on u_1 to the variable y_1, are executed only once. The structured condition of a rewrite rule avoids re-evaluations of boolean conditions. Another generalization of local assignments is matching results of those sub-derivations with patterns, where the non-linear patterns containing several variables composed of constructors can be used instead of local variables.

Strategies

The principal contribution of the PhD thesis [1] is studying a controlling formalism of rewrite derivations, called strategies. The notion of strategies defined in the context of *computational systems* [9] is characterized by the following equation:

$$\mathsf{Programming} = \mathsf{Rules} + \mathsf{Strategies}$$

which integrates object rewrite rules, as a logical part of computations, with their control in a form of strategies. We propose a declarative strategy lan-

[2] their syntax is described in [4]

guage, which offers a uniform and common formalism for object and strategy descriptions, which allows to reuse similar tools for studying of different properties of rewrite and strategy systems, and which enables to use the same implementation techniques at both levels. An advantage of having a declarative strategy language is in the possibility of studying partial evaluation methods as implementation techniques, which can benefit from a strategy formalism without side effects. Principal characteristics of the proposed strategy language are the following:

- The operational semantics is based on rewriting.
- Strategies are typed in a many-sorted signature.
- Strategies allow to express a non-determinism of non-confluent rewrite systems. Two non-determinisms, i.e. *don't-know* and *don't-care*, are considered.
- Strategies can parallelize a derivation to several concurrent sub-derivations.
- Strategies are implemented both by an interpreter optimized by partial evaluation techniques and by a compiler into C++.

The strategy language is a transformation of a first-order object rewrite theory extended by several strategy definitions to a new strategy rewrite theory, where these strategy definitions obtain clear semantics based on rewriting and represented by a computational system. This strategy theory contains an interpreter of strategies independent on the user's specifications. This independent part represents the semantics of pre-defined constructions of the strategy language. The dependent part of this strategy theory consists of several constructions coming from the user's specification, e.g. new strategy sorts and symbols added to the user's signature, and rewrite rules defining the semantics of these symbols. The whole construction of the strategy language consists of three types of strategies - primal, elementary and defined strategies:

- **Primal strategies** correspond to rewrite rules, and their application corresponds to the standard replacement axiom of rewriting logic [10]. In the empty rewrite theory, these strategies represent deterministic and atomic steps of rewriting.
  Example: Let $[add1]\ x \Rightarrow x+1$ be a simple rewrite rule. The corresponding primal strategy $[x \Rightarrow x+1]$, as an object of the strategy theory, can be abbreviated to the notation **add1** using the rule label [3]. Rewrite rules can be also parameterized, e.g. $[add(n)]\ x \Rightarrow x+n$. Then, the primal strategy **add**(1) naturally corresponds to **add1**.
- **Elementary strategies** represent several pre-defined strategy constructors, e.g. $';'$, **dk**, **dc**, **first**, **id**, **fail**, etc., standing for concatenation of strategies, *don't-know*, *don't-care* non-deterministic choices, a deterministic choice, etc. They can be also constructed using functional symbols respecting their arities. Using these pre-defined constructors, we can express

[3] the bold-face font is used for strategies

non-deterministic finite (i.e. non-recursive) strategies, whose semantics is defined in terms of sets of proof terms in rewriting logic. If two strategies are concatenated by the symbol ';', the second strategy is applied on all results of the first one. For any strategies $S_1, \ldots, S_n$, the strategy $\mathbf{dc}(S_1, \ldots, S_n)$ returns all results of one successful and non-deterministically chosen strategy among S_i. For the strategy $\mathbf{dk}(S_1, \ldots, S_n)$, all possible results are returned. The identity strategy **id** does not change a term, while the strategy **fail** always fails, and never gives any result. The strategy $\mathbf{f}(S_1, \ldots, S_n)$ applies all sub-strategies S_i to the sub-terms t_i of the term $f(t_1, \ldots, t_n)$ with the root symbol f.
Example: The elementary strategy $\mathbf{dk}(\mathbf{add}(1), \mathbf{add}(2))$ applied on a term 0 returns two results 1 and 2, while $\mathbf{dc}(\mathbf{add}(1), \mathbf{add}(2))$ gives either 1, or 2.

- **Defined strategies** extend the elementary strategies by parameterized and recursive definitions using strategy rewrite rules.
  Example: Let us take a strategy **map** defined by the following rule:

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

  The right-hand side of this definition means that whenever the strategy **map** with its argument S (i.e. $\mathbf{map}(S)$) is applied on 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. Strategies, as objects of the strategy theory, can be also simplified (i.e. normalized) by a set of convergent rewrite rules, e.g.:

$$\begin{array}{lll} [\mathsf{Dm}] & \mathbf{map}(S_1)\ ;\ \mathbf{map}(S_2) & \Rightarrow \mathbf{map}(S_1\ ;\ S_2) \\ [\mathsf{Im}] & \mathbf{map}(\mathbf{id}) & \Rightarrow \mathbf{id} \\ [\mathsf{Fm}] & \mathbf{map}(\mathbf{fail}) & \Rightarrow \mathbf{fail} \end{array}$$

  These rules simplify a strategy before its application, e.g. the strategy $\mathbf{map}(\mathbf{add}(1))\ ;\ \mathbf{map}(\mathbf{add}(2))$ applied on an integer list is first simplified to $\mathbf{map}(\mathbf{add}(1)\ ;\ \mathbf{add}(2))$, and then applied.

All constructions proposed at these three levels are well-typed in a many-sorted signature of the strategy theory (strategy typing is sketched in Section 3).

Cooperation of solvers

Different constraint solvers are typical ELAN applications, and ELAN allows to formalize their cooperation as concurrent processes[4]. Several low-level UNIX-like controlling primitives of processes (i.e. solvers) are proposed. They initialize (i.e. create), terminate (i.e. kill) and communicate (i.e. read and write via pipes) with solvers in several ways (cf. [2]). The high-level process

[4] even if they are not necessarily defined in ELAN

controlling primitives are represented by concurrent versions of elementary strategy constructors, e.g. $\mathbf{dk}(S_1 \mid\mid \ldots \mid\mid S_n)$, or $\mathbf{dc}(S_1 \mid\mid \ldots \mid\mid S_n)$. They concurrently execute all sub-strategies S_i, and ELAN as a dispatcher, collects all results of one or all sub-strategies, depending on the used strategy constructor. Using strategies $\mathbf{dkcall}(P)$ and $\mathbf{dccall}(P)$, ELAN invokes an external process P for obtaining one or all results. Combining these strategy primitives, e.g. $\mathbf{dk}(\mathbf{dkcall}(P_1) \mid\mid \mathbf{dkcall}(P_2))$, we can parallelize the execution of external solvers in several ways. Using low-level controlling primitives, we can design a process communication in a more fine way, but these atomic primitives work *a priori* in a non-declarative way. However, the semantics of the high-level concurrent strategies corresponds to their sequential version. Communication protocols between concurrent solvers are specified by their implementation. More details and examples are given in [2].

3 Strategy Language

In this section, we illustrate more detailed aspects concerning the semantics, typing and implementation of the proposed strategy language.

Semantics and Typing

The strategy language is based on rewriting logic, therefore, the axioms of rewriting logic are expressed in the operational semantics of the strategy language. The denotational semantics is expressed by sets of proof terms representing the basis of the proof calculus of rewriting logic.

The *operational semantics* defines a set of all results of an application of a strategy S on a term t, and it is represented by a definition of a *strategy application symbol* $[\_]\_$ specified in the interpreter of strategies. Because of the presence of two non-determinisms *don't-know* and *don't-care*, results of an application $[S]t$ are structured at two levels: as a set of all *don't-care* results, and each of them is a set of *don't-know* results, where each of them is a term. The following table illustrates the application symbol on several examples:

$[\mathbf{dk}(\mathbf{add(1)}, \mathbf{add(2)})]0$	$\{\{1,\ 2\}\}$
$[\mathbf{dc}(\mathbf{add(1)}, \mathbf{add(2)})]0$	$\{\{1\}, \{2\}\}$
$[\mathbf{first}(\mathbf{add(1)}, \mathbf{add(2)})]0$	$\{\{1\}\}$
$[\mathbf{cons}(\mathbf{dk}(\mathbf{add(1)}, \mathbf{add(2)}), \mathbf{nil})]cons(0, nil)$	$\{\{cons(1, nil), cons(2, nil)\}\}$
$[\mathbf{cons}(\mathbf{dc}(\mathbf{add(1)}, \mathbf{add(2)}), \mathbf{nil})]cons(0, nil)$	$\{\{cons(1, nil)\}, \{cons(2, nil)\}\}$

The *denotational semantics* $\mathcal{D}$ characterizes a strategy as a set of corresponding proofs (i.e. proof terms). These proof terms are constructed from rule labels, function symbols and the concatenation symbol ';' (respecting their arities), and one-to-one correspond to rewrite derivations. The following table illustrates the denotational semantics $\mathcal{D}$ on several examples of non-

deterministic strategies:

$$\begin{aligned}\mathcal{D}(\mathbf{dk}(\mathbf{add(1)},\ \mathbf{add(2)})) &= \{\{\mathbf{add(1)},\ \mathbf{add(2)}\}\}\\ \mathcal{D}(\mathbf{dc}(\mathbf{add(1)},\ \mathbf{add(2)})) &= \{\{\mathbf{add(1)}\},\{\mathbf{add(2)}\}\}\\ \mathcal{D}(\mathbf{first}(\mathbf{add(1)},\ \mathbf{add(2)})) &= \{\{\mathbf{add(1)}\}\}\end{aligned}$$

Both semantics are related in the usual way (cf. more details in [1]). Several relations presented in the form of equivalences of elementary strategies w.r.t. the both semantics are studied in [1], e.g.:

$$\begin{array}{ll}\mathbf{dk}(S',\ S'');S \equiv \mathbf{dk}(S';S\ ,\ \ S'';S) & S;\mathbf{dk}(S',\ S'') \equiv \mathbf{dk}(S;S',\ \ S;S'')\\ \mathbf{dc}(S',\ S'');S \not\equiv \mathbf{dc}(S';S,\ \ S'';S) & S;\mathbf{dc}(S',\ S'') \not\equiv \mathbf{dc}(S;S',\ \ S;S'')\end{array}$$

The construction of the strategy language introduces a system of strategy sorts $\langle s_1 \mapsto s_2\rangle$ representing strategies transforming terms of a sort s_1 to terms of a sort s_2. The case $s_1 = s_2$ represents *sort preserving strategies*, and strategy typing rules for this simple case are the following:

- if $f : (s_1 \dots s_n) \mapsto s$, then the strategy symbol **f** has the following profile: $(\langle s_1 \mapsto s_1\rangle \dots \langle s_n \mapsto s_n\rangle) \mapsto \langle s \mapsto s\rangle$,
- if both sides of a rule ℓ have the sort s, then a primal strategy **l** is of the sort $\langle s \mapsto s\rangle$,
- if φ is an elementary strategy constructor, like $\mathbf{dk}, \mathbf{dc}, \mathbf{id}, \mathbf{fail}, \dots$, then it has the profile $(\langle s \mapsto s\rangle \dots \langle s \mapsto s\rangle) \mapsto \langle s \mapsto s\rangle$, respecting its arity.

In the example of the **map** strategy, there are the following *sort preserving* strategy constructors: $\mathbf{nil} : \langle List \mapsto List\rangle$,
$\mathbf{cons}(\_,\_) : (\langle Elem \mapsto Elem\rangle\ \langle List \mapsto List\rangle) \mapsto \langle List \mapsto List\rangle$, and using these typing rules, the sort of **map** is $(\langle Elem \mapsto Elem\rangle) \mapsto \langle List \mapsto List\rangle$.

By iteration of the strategy language construction, more complicated strategy sorts can be created, e.g. the primal strategy **Dm** defined by a rewrite rule in the previous section has the sort $\langle\langle List \mapsto List\rangle \mapsto \langle List \mapsto List\rangle\rangle$. This primal strategy **Dm** can be used to define high-order strategies applied on strategies of the sort $\langle List \mapsto List\rangle$, e.g. $\mathbf{map}(\mathbf{add}(1))$; $\mathbf{map}(\mathbf{add}(2))$. In such a way, a *strategy tower* can be created. More complex typing rules, illustrated in [3], deals also with the case of *sort-changing* strategies.

Implementation

The strategy language was prototyped by its interpreter defined as a computational system written in ELAN. Advantages of this implementation are the *readability* of its ELAN specification and its *extensibility* by new language constructions. The low efficiency, as its main drawback, can be partially eliminated using the ELAN compiler, because the strategy interpreter is an ELAN program. This implementation technique not related to particularities of this interpreter, gives promising results; the speed-up is about 100. Different ways

of implementation are studied in the thesis, e.g. an adaptation of partial evaluation to the strategy language, or compilation techniques of strategies.

Partial Evaluation – The strategy language is a transformation, whose result is a computational system containing rewrite rules and atomic (or built-in) ELAN strategies. There is an atomic strategy *eval* controlling the execution of the strategy interpreter, which consists of several cases corresponding to different language constructions. A program specialization, as a partial evaluation technique, is used in order to eliminate unusable cases of this atomic strategy *eval*, and also to specialize rules of eventually usable cases in each application of *eval*. An input of this specialization method is an atomic ELAN strategy (e.g. *eval*) and a partially known term on which it is applied. The static structure of this term allows, in several cases, to eliminate unusable rules of this strategy. A result is a specialized version of this atomic strategy preserving the operational semantics, and possibly referring to new rewrite rules. These new specialized rewrite rules are instances of the original ones. Due to this instantiation, the structure of terms in new rules is more specialized, which allows to perform this specialization technique also on these specialized rules. The more detailed description of this method is presented in [5]. The speed-up of this method is about 2–4, which is a good result for program specialization and transformation techniques. This method first implemented in C++ and integrated into the ELAN system is also redefined in ELAN using a new meta-representation of ELAN programs, called REF [6]. This meta-representation of computational systems was designed as an external exchange format between several ELAN tools, and it allows to specify different program transformations in ELAN itself.

Compilation – Three methods of compilation of strategies are proposed in [1]. They substantially improve results obtained by a *naive* compilation method, when computational systems containing the strategy interpreter are compiled by the ELAN compiler.

In the first compilation method, the evaluation strategy *eval* (a crucial part of the strategy interpreter written in ELAN) is manually pre-compiled into C++ and each call of *eval* is then replaced by its pre-compiled version $eval_c$, e.g. the rule interpreting **map**:

$$[\mathbf{map}(S)]t \quad \Rightarrow \quad y \ \mathbf{where}\ y := (\underline{eval_c})[\mathbf{dc}(\mathbf{nil}, \mathbf{cons}(S, \mathbf{map}(S)))]t$$

The main drawback of the first approach is that the construction of strategy terms, e.g. $\mathbf{dc}(\mathbf{nil}, \mathbf{cons}(S, \mathbf{map}(S)))$, as an expensive operation, is not optimized by this method.

The second technique compiles strategy applications, i.e. strategy terms applied on object terms under the control of the strategy *eval*. In the rule:

$$[\mathbf{map}(S)]t \quad \Rightarrow \quad y \ \mathbf{where}\ y := (eval)[\underline{\mathbf{dc}(\mathbf{nil}, \mathbf{cons}(S, \mathbf{map}(S)))}]t$$

the application of a strategy term $\mathbf{dc}(\mathbf{nil}, \mathbf{cons}(S, \mathbf{map}(S)))$ is compiled into

C++ instead of its construction and interpretation by the strategy *eval*, or $eval_c$. This is done by several compilation schemas introduced for each predefined elementary strategy constructor (e.g. **dk**, **dc**, **nil**, etc.). This method compiles a strategy term $\mathbf{dc}(\mathbf{nil}, \mathbf{cons}(S, \mathbf{map}(S)))$ into a sequence of C++ instructions, but a call of the strategy interpreter *eval* is generated for a substrategy S unknown during compilation. Therefore, the strategy S is always interpreted, e.g. if $\mathbf{dk}(S_1, S_2)$ is an argument of the strategy $\mathbf{map}(\mathbf{dk}(S_1, S_2))$, it is always interpreted by the strategy *eval*, or $eval_c$.

The third method eliminates this drawback by compilation not only applied strategies but also strategies non-applied on terms. Strategies can be either applied on terms, or simplified. That is why, this method treats an internal representation suitable for both these operations. The representation in the form of terms is more adequate for simplifications, while for applications, pointers to their pre-compiled C++ codes are more adequate.

4 Perspectives

The proposed strategy language is the main contribution of the thesis [1]. Our research perspectives are the following:

- improving the implementation of the strategy language, in particular:
  - combination of partial evaluation and compilation methods,
  - implementation of the third proposed compilation method,
  - integration of standard optimization techniques used in compilers, e.g. tail-recursion optimization, in order to improve the compilation methods.
- studying reflexive aspects of computational systems:
  - implementation of the strategy language based on these reflexive aspects.
- studying the strategy language as a platform of a cooperation of solvers.

Acknowledgments

I am grateful to Hélène Kirchner and Claude Kirchner for supervising my research, and to Igor Prívara for providing useful comments and remarks on an earlier version of this paper that allowed me to improve this work.

References

[1] P. Borovanský. *Le contrôle de la réécriture: étude et implantation d'un formalisme de stratégies.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, France, 1998. To appear.

[2] P. Borovanský and C. Castro. Cooperation of Constraint Solvers: Using the New Process Control Facilities of ELAN. In C.Kirchner and H.Kirchner, editors, *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications, RWLA'98, Pont-à-Mousson (France)*, volume 15, pages 379 – 398. Electronic Notes in Theoretical Computer Science, September 1998.

[3] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling Rewriting by Rewriting. In J. Meseguer, editor, *Proceedings of the 1st International Workshop on Rewriting Logic and its Applications, RWLA'96, Asilomar, Pacific Grove (CA, USA)*, volume 4. Electronic Notes in Theoretical Computer Science, September 1996.

[4] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An Overview of ELAN. In C.Kirchner and H.Kirchner, editors, *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications, RWLA'98, Pont-à-Mousson (France)*, volume 15, pages 329 – 344. Electronic Notes in Theoretical Computer Science, September 1998.

[5] P. Borovanský and H. Kirchner. Strategies of ELAN: meta-interpretation and partial evaluation. In *Proceedings of International Workshop on Theory and Practice of Algebraic Specifications ASF+SDF 97, Amsterdam (The Nederlands)*, Workshops in Computing. Springer-Verlag, September 1997.

[6] P. Borovanský, C. Ringeissen, and P.-E. Moreau. Handling ELAN Rewrite Programs via an Exchange Format. In C.Kirchner and H.Kirchner, editors, *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications, RWLA'98, Pont-à-Mousson (France)*, volume 15, pages 207 – 224. Electronic Notes in Theoretical Computer Science, September 1998.

[7] M. Clavel. Reflection in General Logics and in Rewriting Logic with Applications to the Maude Language. In C.Kirchner and H.Kirchner, editors, *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications, RWLA'98, Pont-à-Mousson (France)*, volume 15, pages 317 – 328. Electronic Notes in Theoretical Computer Science, September 1998.

[8] M. Clavel. *Reflection in general logics, rewriting logic, and Maude.* PhD thesis, University of Navarre (Spain), 1998.

[9] C. Kirchner, H. Kirchner, and M. 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.

[10] N. Martì-Oliet and J. Meseguer. Rewriting logic as a Logical and Semantical Framework. Technical report, SRI International, Computer Science Laboratory, Menlo Park (CA, USA), May 1993.

[11] E. Visser and Z. Benaissa. A Core Language for Rewriting. In C.Kirchner and H.Kirchner, editors, *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications, RWLA'98, Pont-à-Mousson (France)*, volume 15, pages 25 – 44. Electronic Notes in Theoretical Computer Science, September 1998.

[12] M. 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.