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

Rewriting as a Unified Specification Tool for Logic and Control: The ELAN Language

P. Borovansky, C. Kirchner 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

Rewriting as a Unified Specification Tool for Logic and Control: The ELAN Language. Extended Abstract

Peter Borovanský
LORIA & INRIA

Claude Kirchner
LORIA & INRIA

Hélène Kirchner
LORIA & CNRS

BP 101, 54602 Villers-les-Nancy CEDEX, FRANCE
<http://www.loria.fr/equipes/protheo>

Abstract

We introduce computational systems to formalise the notion of rewriting directed by user defined strategies. This provides a semantics for ELAN, an environment dedicated to prototype, experiment and study the combination of different deduction systems for constraint solving, theorem proving and logic programming paradigms. Formally, a computational system can be represented as a rewrite theory in rewriting logic together with a notion of strategy to select relevant computations. We show how conveniently the strategies can also be specified using again computational systems. Several non-trivial examples of strategy description are described including a search space library and its use for solving problems like game winning strategies.

1 Introduction

ELAN is an environment dedicated to prototype, experiment and study the combination of different deduction systems for constraint solving, theorem proving and logic programming paradigms. Its evaluation mechanism is rewriting, whose elegance and expressiveness as a computational paradigm are no more to be stressed as evidenced by systems like ASF+SDF [Kli93] or OBJ3 [GKK⁺87]. Less evident is the difficulty that comes from the absence of explicit control mechanism over rewriting. In many existing rewriting-based languages or systems, the term reduction strategy is hard-wired and is not accessible to the designer of an application. Indeed controlling rewriting is a fundamental issue as soon as one wants to use rewriting as a specification language. We show in this paper how the control can itself be specified using rewriting.

The ELAN language is based on the concept of computational systems [KKV95, Vit94, BKK⁺96b] given by a signature providing the syntax, a set of conditional rewrite rules describing the deduction mechanism, and a strategy to guide application of rewrite rules. Formally, this is a rewrite theory in rewriting logic [Mes92], [MOM93a], together with a notion of strategy to select relevant computations. Each ELAN module defines its own signature, labelled rewrite rules and strategies. ELAN is implemented in C++.

We first show in Section 2 how non-deterministic computations are handled in ELAN and combined with deterministic ones. Then Section 3 describes the strategy language of ELAN which provides some predefined primitives and the possibility for the user to define his own strategies in a very flexible way using the same paradigm of rewriting. For instance the user may want to develop his own search space library and use it for solving different problems. This is illustrated in Section 4. The conclusion addresses some further research perspectives.

2 Evaluation with rules and strategies

ELAN is a system for prototyping non-deterministic computations thanks to rules and strategies. An ELAN program is composed of a signature part describing operators with their types, a set of rules and a set of strategies. Strategies

is one of the main originality of ELAN compared to other algebraic specification languages based on rewriting. 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 labelled conditional rewrite rules with local variable affectations

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

ℓ is the rule label, l and r the respective left and right-hand sides, v the condition and $y := [S]u$ a local affectation, 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 , using if necessary, associative commutative matching (if some operators in the signature are declared as associative and commutative). Then the expressions in **where** and **if** parts, instantiated with the matching substitution, are evaluated in order. When there is no failure, this usually instantiates the local variables (such as y) in **where** expressions, and this extends the matching substitution. When every condition is satisfied, the replacement by the instantiated left-hand side is performed. Indeed the power of this mechanism comes from the fact that **where** expressions may invoke strategy evaluation.

Application of a rewrite rule in ELAN yields, in general, several results. This is first due to equational matching (for instance AC-matching) and second to the **where** assignment, since it may itself return several possible assignments for variables, due to the use of strategies. When a rewrite rule or a strategy returns an empty set of terms, we say that it *fails*.

Thus the language provides a way to handle this non-determinism. This is done using a few basic strategy operators: **first** (or **dc**) standing for first non-failing result, and **dk** standing for dont know choose. For a rewrite rule $[\ell] : l \Rightarrow r$ the strategy **first**(ℓ) returns the first non-failing result of the application of the rule. On the contrary, if the ℓ rule is applied using the **dk**(ℓ) strategy, then *all* possible results are computed and returned by the strategy. The implementation handles these several results by an appropriate back-chaining operation. This is extended to the application of several rules: the **dk** strategy results in the application of all substrategies and yields the concatenation of all results; the application of the **first** strategy returns the set of results of the first non-failing rule application. If all sub-strategies fail, then it fails too, i.e. it yields the empty set.

Strategies can be described using **dk** and **first** operators, a concatenation operator **;**, as well as with the elementary strategies that are rule labels and the strategies **fail** for failure, and **id** for identity. Iterators are also provided and user defined strategies operators as well as their semantics can be expressed in the language. This is done using rewriting itself as detailed in the next section.

3 The strategy language

We are now describing how strategies can be specified using rewriting itself. This is done at two levels. First, the user defined strategies are specified using rewrite rules that can themselves be controlled by (meta) strategies. Second, the operational semantics of the strategy language is also defined as a rewrite theory with strategy, making the whole language description supported by the single concept of computational system.

We are using in this section the basic notions of rewriting logic and of proof terms. Our notations are consistent with the ones of [KKV95, Mes92, MOM93b].

The notion of strategy as subset of proof terms of a rewrite theory was proposed in [KKV95, Vit94]. The set of (*closed*) *proof terms* Π is defined as the set of terms $\mathcal{T}(\mathcal{F} \cup \mathcal{L} \cup \{ ; \})$ built on function symbols $\mathcal{F}$ of the rewrite theory, labels $\mathcal{L}$ and the symbol **;** standing for concatenation. The symbol $_ \Rightarrow _$ is defined on $\Pi \times \mathcal{T}(\mathcal{F}) \times \mathcal{T}(\mathcal{F})$ and $\pi : t \Rightarrow t'$ means that there is a chain of rewrite steps from the term t to the term t' encoded by the proof term π . It is easy to see that for each proof term $\pi \in \Pi$, the sets:

$$dom(\pi) = \{t \mid \exists t' \text{ s.t. } \pi : t \Rightarrow t'\} \text{ and } cod(\pi) = \{t' \mid \exists t \text{ s.t. } \pi : t \Rightarrow t'\}$$

are either both singletons or both empty. Thus, we can also define a partial function $[_]_ : \Pi \times \mathcal{T}(\mathcal{F}) \mapsto \mathcal{T}(\mathcal{F})$, such that $[\pi]t = t'$ when $dom(\pi) = \{t\}$ and $cod(\pi) = \{t'\}$.

A *strategy* S is defined as a subset of Π . The definition of the symbol $_{-} : _{-} \Rightarrow _{-}$ is extended to strategies as follows:

$$S : t \Rightarrow t' \text{ if there exists } \pi \in S, \text{ such that } \pi : t \Rightarrow t'$$

and

$$[S]t = \{t' \mid \pi \in S, \pi : t \Rightarrow t'\}.$$

The definition of domain can be extended too:

$$\text{dom}(S) = \{t \mid \exists t' \in \mathcal{T}(\mathcal{F}), \exists \pi \in S, \text{ s.t. } \pi : t \Rightarrow t'\}.$$

An arbitrary strategy $S \subseteq \Pi$ could be very complicated or irregular from the computational point of view (for instance a non-recursive set). This is why we concentrate on describing special subclasses of strategies and define two subclasses of general strategies, called elementary and defined strategies whose operational semantics are given using rewriting logic. The main difference between these two classes is that elementary strategies are predefined in ELAN while defined strategies are user-definable, and may be recursive, parameterised and typed as well.

3.1 Elementary strategies

The first step is the construction of elementary strategies, where *an elementary strategy* S is an element of the set of terms $\mathcal{T}(\mathcal{F} \cup \mathcal{L} \cup \{\text{;}, \text{id}, \text{dc}, \text{dk}, \text{first}, \text{fail}\})$. Roughly speaking, elementary strategies represent non-recursive non-deterministic computations. We already introduced the strategy operators $\text{;}, \text{dc}, \text{dk}, \text{first}, \text{fail}, \text{id}$. In addition, $\text{l}(S_1, \dots, S_n)$ corresponds to an application of a rewrite rule labelled by $\ell \in \mathcal{L}$, which also applies substrategies S_i on values of variables x_i after matching and before replacing the matched subterm by the instantiated right-hand side of the rewrite rule. $\text{f}(S_1, \dots, S_n)$ is an application of substrategies S_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 this volume (see [BK97]).

Elementary strategies can be given sorts, based on the set of sorts $\mathcal{S}$ defined in the user's rewrite theory. This introduces new strategy sorts $\langle s' \mapsto s'' \rangle$ for the sort of all elementary strategies transforming terms of sort s' into terms of sort s'' .

3.2 Defined strategies

The goal is now to be able to define new strategies by rewrite rules. Let us take the very easy example of **map**, that can be defined by a strategy rewrite rule in the following way:

$$\text{map}(S) \Rightarrow \text{dc}(\text{nil}, S.\text{map}(S)) \quad (1)$$

The right-hand side of this definition means that whenever the strategy **map** with an argument S (i.e. $\text{map}(S)$) 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 $\text{map}(S)$ is further applied on the tail of t .

This strategy definition can be formulated also using the strategy application symbol $[_{-}]_{-}$:

$$\begin{aligned} [\text{map}(S)]\text{nil} &\Rightarrow \text{nil} \\ [\text{map}(S)]a.as &\Rightarrow [S]a.[\text{map}(S)]as \end{aligned} \quad (2)$$

The difference relies on the fact that a list, which the functional **map** is applied on, is an explicit argument in the second (or third) definition, while in the first one, it is implicit.

Rule (1) applied without care may lead to infinite computations, for example when taking a left-most inner-most reduction strategy. This justifies the concept of meta-strategies, i.e. strategies which control the execution by rewriting of defined strategies. Such a strategy can be either constructed by enriching the strategy interpreter, or explicitly defined by the user from labelled rules on strategies.

A rule like $\text{map}(S) \Rightarrow \text{dc}(\text{nil}, S.\text{map}(S))$ defines recursively the strategy **map**. Indeed other rules involving **map** can be written, to express in particular some equivalence on defined strategies, for instance the distributivity of **map** on the concatenation ; or its application on the identity:

$$\begin{array}{lcl} \text{map}(S_1) ; \text{map}(S_2) & \Rightarrow & \text{map}(S_1 ; S_2) \\ \text{map}(\text{id}) & \Rightarrow & \text{id} \end{array}$$

Moreover, properties of strategies can be described by rewrite rules such as:

$$\begin{array}{lcl} \text{id} ; S & \Rightarrow & S \\ \text{first}(S, S) & \Rightarrow & S \end{array} \quad \begin{array}{lcl} S ; \text{id} & \Rightarrow & S \\ \text{dk}(S, S) & \Rightarrow & S \end{array}$$

In ELAN, a *strategy definition* is given by:

- A set of strategy symbols $\mathcal{F}_{\mathcal{D}}$ with signatures: $\mathbf{d} : (s_1, \dots, s_n) \langle s' \mapsto s'' \rangle$ where $s_1, \dots, s_n$ are strategy sorts of the arguments, and $\langle s' \mapsto s'' \rangle$ is the sort of the result.
- A finite set of rewrite rules:
 - either in an implicit form $\mathbf{d}(S_1, \dots, S_n) \Rightarrow S$
 - or in an explicit form $[\mathbf{d}(S_1, \dots, S_n)]x \Rightarrow [S]x$.

where $\mathbf{d} \in \mathcal{F}_{\mathcal{D}}$, S_i for $i = 1, \dots, n$ are elementary strategies, $x : s'$ is a new variable in $\mathcal{X}$, S is a term built from defined strategy symbols, elementary strategy symbols, rule labels and variables.

The interpreter for elementary strategies is expanded in both cases of the previous definition with new rules, labelled with DSTR:

$$[DSTR] \quad [\mathbf{d}(S_1, \dots, S_n)]x \Rightarrow [S]x$$

Different evaluation modes for strategy application are offered via different possibilities for labels. A rewrite rule on strategy labelled by $[]$ is applied just as unlabelled rules on terms in an innermost-outermost way. A rewrite rule on strategy labelled by the special label $[\cdot]$ is applied in a lazy way using the built-in strategy of the interpreter of strategies, described in [BK97]. A rewrite rule on strategy labelled by $[\ell]$ is applied just as labelled rules on terms with a user-defined strategy built from these labels.

Example 1 The definition of the strategy **map** is for instance given by the definition of the strategy symbol

$$\text{map} : (\langle s \mapsto s \rangle) \langle \text{list}[s] \mapsto \text{list}[s] \rangle$$

and by the rule

$$\text{map}(S) \Rightarrow \text{dc}(\text{nil}, S \cdot \text{map}(S)),$$

where $S : \langle s \mapsto s \rangle$ and $s \in \mathcal{S}$. The following rule:

$$\text{DSTR}_{\text{List}[\text{list}[s]]} [\text{map}(S)]x \Rightarrow [\text{dc}(\text{nil}, S \cdot \text{map}(S))]x$$

is added to the strategy evaluator.

This way of expressing strategies is extremely powerful and can be used for defining simple strategies like the following ones, where the variables in use have the sorts: $x : \langle s \mapsto s \rangle$, $xs : \langle \text{list}[s] \mapsto \text{list}[s] \rangle$, $ys : \langle u \mapsto v \rangle$.

$$\begin{array}{lcl} \text{iterate} : & (\langle s \mapsto s \rangle) \langle s \mapsto s \rangle & \text{iterate}(x) \Rightarrow \text{dk}(x; \text{iterate}(x), \text{id}) \\ \text{repeat} : & (\langle s \mapsto s \rangle) \langle s \mapsto s \rangle & \text{repeat}(x) \Rightarrow \text{first}(x; \text{repeat}(x), \text{id}) \\ \text{map1} : & (\langle s \mapsto s \rangle) \langle \text{list}[s] \mapsto \text{list}[s] \rangle & \text{map1}(x) \Rightarrow \text{first}(\text{nil}, x \cdot \text{map1}(x)) \\ \text{map2} : & (\langle \text{list}[s] \mapsto \text{list}[s] \rangle) \langle \text{list}[s] \mapsto \text{list}[s] \rangle & \text{map2}(\text{nil}) \Rightarrow \text{nil} \\ & & \text{map2}(x.xs) \Rightarrow x \cdot \text{map2}(xs) \end{array}$$

The **iterate** strategy differs from **repeat** in a such way that it returns all intermediate forms during the normalisation of a term by the application of a strategy x , while **repeat** returns only the last one (i.e. normal form). The strategy **map1** applies a fixed strategy x on all elements of a list and produces a new list of transformed elements. The strategy **map2** is driven by a list of strategies which are respectively applied to elements of a list of same length.

For instance, with the rewrite rule: $[zero] x + 0 \Rightarrow x$ and strategies defined above:

$$\begin{array}{ll} [\text{iterate}(\text{zero}(\text{id}))](a + 0) + 0 & \text{yields } \{(a + 0) + 0, a + 0, a\} \\ [\text{repeat}(\text{zero}(\text{id}))](a + 0) + 0 & \text{yields } \{a\}. \end{array}$$

4 A search space library

Describing deduction processes is very convenient using computational systems which have indeed been created for this purpose. In the earlier version of the ELAN language, strategy definition was only possible using nullary strategy operators. This was already quite powerful and we have used this first version of the system to describe (and execute) several non-trivial computational systems. In particular, we have shown in [KM95] how saturation-based theorem provers like completion can be easily and efficiently implemented using rewrite rules and strategies. We have also implemented a mechanical theorem prover for first-order predicate calculus [BKK96a] and more recently the predicate prover of the B system [CK97].

We are now using the full power of user defined strategies for specifying a strategy library that provides primitives for depth-first search in a computation tree. Two natural applications of this strategy library are examples of a winning strategy for a tic-tac-toe game, and a guided strategy helping to find out a path (or, all paths) towards an exit of a labyrinth.

The strategy library supports four primitives for depth-first search. The current situation of the partially discovered search tree is represented by a path from the initial state (the root of the tree) to the current state (the left-most leaf). It also remembers all possible alternatives (or, choice points) met along this path. Each step of the path is represented by a list of states, where its head is the currently chosen state, and the rest represents all non-explored alternative states. The limitation of this approach is that the number of successive (alternative) states of any state should be finite, i.e. the search tree has a finite degree. Each step of the path is a list of states, thus, the whole path could be naturally represented as $list[list[state]]$. This sort also describes the current situation during the depth-first search in this tree. We define four primitive strategies over the sort $list[list[state]]$, which develop and maintain this search tree. Their intended semantics are the following:

- $call(S)$ applies the strategy S on the current state, where S is a strategy transforming a state into a state, i.e. of the sort $\langle state \mapsto state \rangle$. All possible results of this application, if there are any, are formed into a new step (level) attached to the current path.
- $next$ throws away the current state and continues searching with the next possibility of the previous step (the latest choice-point).
- $exit$ leaves the current state ignoring all alternatives and it returns the control to the previous step of the current path.
- cut eliminates all alternative states of the latest step.

For better understanding the definitions in Figure 1, let us recall that on the left-hand sides, the variable $state$ represents the current state, the variable $level$ represents all alternatives of the current state, i.e. $state.level : list[state]$ is one step of the path, where the others steps are recorded in the variable $space$.

The only non-trivial strategy among the four primitives is the strategy $call(S)$, which uses the function symbol set_of to collect all results of an application of a strategy S .

```

stratop global
  call(@)      : (<state->state>) <list[list[state]]->list[list[state]]>;
  next         : <list[list[state]]->list[list[state]]>;
  exit         : <list[list[state]]->list[list[state]]>;
  cut          : <list[list[state]]->list[list[state]]>;
end
strategies for list[list[state]]
  s            : <state->state>;
  state        : state;
  level,
  new_level    : list[state];
  space        : list[list[state]];
explicit
  [.]          [call(s)]state.level.space => new_level.state.level.space
                                     where new_level:=()setof(s,state)
                                     if new_level != nil
  [.]          [next]state.level.space => level.space
  [.]          [exit]state.level.space => space
  [.]          [cut]state.level.space => state.nil.space
end

```

Figure 1: Search-space library

4.0.1 Tic-tac-toe game

The use of the depth-first search library is illustrated by the definition of a game winning strategy for a simple game with a finite search space. The solution described below is split into game-dependent and independent parts. The game-independent part does not rely on rules of a particular game. These particularities are hidden in the game-dependent kernel.

The particular game chosen for this example is the tic-tac-toe game with the play-board of the size 3×3 . Two players with different signs alternatively put one sign on the board. The winner is the player who succeeds in having 3 signs in a same column, row, or diagonal.

The game-dependent part defines the state (Figure 2) as a pair encoding in its first component the next player

```

operators global
  X      : Sign;      { sign of the 1-st player }
  O      : Sign;      { sign of the 2-nd player }
  -      : Sign;      { empty sign }
  * @,   : Sign;      { next player's sign }
  |@,@,@|,
  |@,@,@|,
  |@,@,@| *
  : ( Sign Sign Sign Sign Sign Sign Sign Sign Sign Sign ) state;
end

```

Figure 2: A state of tic-tac-toe game

and in its second component the play-board. There are three basic strategies *win*, *put(row, column)*, *moves*, all of the sort $\langle state \mapsto state \rangle$. *next* is a function that switches players. Their definitions are given in a straightforward exhaustive way, thus, we show only a small part of them in Figure 3.

The strategy *win* : $\langle state \mapsto state \rangle$ tries to put the last sign and to win the game. It fails, if there is no correct winning move. The deterministic strategy *put(row, column)* puts the sign on the field with coordinates *row* and *column*, if it is possible. The non-deterministic strategy *moves* generates all possible moves (according to the game rules) from the current situation.

The game-independent part specifies the winning strategy *movetowin* using an auxiliary strategy *findloosing* in Figure 4. The strategy *movetowin*, first of all, tries to apply the basic strategy *win* searching a winning move. This application is realised by calling the library primitive *call(win)*. It succeeds if there is a winning move. Otherwise, the strategy *movetowin* produces the set of all possible moves by *call(moves)*. These successive possibilities are filtered by the strategy *findloosing* removing all winning configurations of the adversary. If there is at least one possible move such that the adversary's configuration is not winning, the strategy *findloosing* succeeds, and all alternative

```

strategies for state
a, b, c, d, e, f, g, h, i, Y          : Sign;
explicit
[.] [win] *Y,|-,Y,Y|,|d,e,f|,|g,h,i|* => *nxt(Y),|Y,Y,Y|,|d,e,f|,|g,h,i|* end
[.] [win] *Y,|Y,-,Y|,|d,e,f|,|g,h,i|* => *nxt(Y),|Y,Y,Y|,|d,e,f|,|g,h,i|* end
[.] [win] *Y,|Y,Y,-|,|d,e,f|,|g,h,i|* => *nxt(Y),|Y,Y,Y|,|d,e,f|,|g,h,i|* end
[.] [put(1,1)] *Y,|-,b,c|,|d,e,f|,|g,h,i|* => *nxt(Y),|Y,b,c|,|d,e,f|,|g,h,i|* end
[.] [put(1,2)] *Y,|a,-,c|,|d,e,f|,|g,h,i|* => *nxt(Y),|a,Y,c|,|d,e,f|,|g,h,i|* end
[.] [put(1,3)] *Y,|a,b,-|,|d,e,f|,|g,h,i|* => *nxt(Y),|a,b,Y|,|d,e,f|,|g,h,i|* end
implicit
[.] moves => dk(
    put(1,1), put(1,2), put(1,3),
    put(2,1), put(2,2), put(2,3),
    put(3,1), put(3,2), put(3,3)
)
end
    
```

Figure 3: Basic moves of tic-tac-toe game

```

stratop global
movetowin      : <list[list[state]]->list[list[state]]>;
findloosing    : <list[list[state]]->list[list[state]]>;
guess          : <list[list[state]]->list[list[state]]>;
end
strategies for list[list[state]]
implicit
[.] movetowin => first(call(win), call(moves);findloosing;cut) end
[.] findloosing => if movetowin then exit; next; findloosing
                  or else id
                  fi
[.] guess => first(movetowin, call(moves);heuristic)
end
    
```

Figure 4: A winning strategy

moves are ignored by the primitive *cut*. Otherwise, the strategy *findloosing* fails, and *movetowin* fails also. These two mutually recursive strategies represent the kernel of the game-independent part. The winning strategy *movetowin* may be encapsulated by a strategy *guess*, which either proposes to the player a winning move, or, if there is no winning move, proposes one of the non-winning configuration chosen with an heuristic strategy. The strategy *heuristic* saying what to do if we are not in a winning configuration, depends on the particular game and some defensive tactic. When no specific heuristic is proposed, it can be replaced by the strategy *id*. Figure 5 illustrates the use of these strategies on a small game session. With a similar encoding of the game state and rules, we can also specify other games, such as the nim-game in [CELM96].

4.0.2 Labyrinths

Another application of our strategy library is a definition of a strategy helping to find out an exit from a labyrinth. An alternative re-formulation of this problem could be to find a path (or, all paths) between two nodes of a graph. As in the previous section, we split the solution into a part dependent on the labyrinth and an independent one.

The dependent part contains a specification of a state, which is a pair of coordinates, and a definition of four basic moves in the labyrinth. They are realized as state transforming strategies *left*, *right*, *up*, *down* of sort $\langle state \mapsto state \rangle$ such that these strategies fail, if some movement in a given state is not possible (because of a wall). All exits of the labyrinth are specified by a strategy *exitable*, which fails, if the current state has ‘no exit doors’. Otherwise, it is equivalent to the identity strategy.

The independent part of the strategy (Figure 6) consists of a definition of the strategy *search* of sort

$$\langle list[list[state]] \mapsto list[list[state]] \rangle$$


```

elan's first move from
  [guess] *X,|,-,-,|,-,-,-,|,-,-,-,* to *O,|X,-,-,|,-,-,-,|,-,-,-,*
player's first move from
  [put(2,2)] *O,|X,-,-,|,-,-,-,|,-,-,-,* to *X,|X,-,-,|,-,O,-,|,-,-,-,*
elan's attack from
  [guess] *X,|X,-,-,|,-,O,-,|,-,-,-,* to *O,|X,X,-,|,-,O,-,|,-,-,-,*
player's defense from
  [put(1,3)] *O,|X,X,-,|,-,O,-,|,-,-,-,* to *X,|X,X,O,|,-,O,-,|,-,-,-,*
elan's defense from
  [guess] *X,|X,X,O,|,-,O,-,|,-,-,-,* to *O,|X,X,O,|,-,O,-,|,X,-,-,-,*
player's defense from
  [put(2,1)] *O,|X,X,O,|,-,O,-,|,X,-,-,-,* to *X,|X,X,O,|,O,O,-,|,X,-,-,-,*
elan's defense from
  [guess] *X,|X,X,O,|,O,O,-,|,X,-,-,-,* to *O,|X,X,O,|,O,O,X,|,X,-,-,-,*
no more chance for the player from
  [put(3,2)] *O,|X,X,O,|,O,O,X,|,X,-,-,-,* to *X,|X,X,O,|,O,O,X,|,X,O,-,-,*
no more chance for elan from
  [guess] *X,|X,X,O,|,O,O,X,|,X,O,-,-,* to *O,|X,X,O,|,O,O,X,|,X,O,X,*

game is over, draw game

```

Figure 5: Tic-tac-toe game

```

strategies for state
implicit
  [.]          moves => dk(right, down, left, up)
end
strategies for list[list[state]]
  state       : state;
  level       : list[state];
  space       : list[list[state]];
explicit
  [.]         [search] state.level.space =>
                [first(      call(exitable),
                             if occurs state in space then
                               next;search
                             else
                               first(call(moves);search, next;search)
                             fi
                )] state.level.space
end

```

Figure 6: One path in the labyrinth

and an auxiliary strategy *moves* of sort $\langle state \mapsto state \rangle$. The strategy *search* on an exit-able state (a room with exit doors) succeeds. Otherwise, if the current path contains a loop, it backtracks and it takes the next possible state for continuation. If it does not contain a loop, it either moves to the neighbour state and continues to search, or, if no exit has been found, it ignores the current choice, and backtracks to the next alternative.

Intuitively, if we need to find out all possible paths, we have to replace deterministic constructions (i.e. **first**) in the definition of the strategy *search* by a non-deterministic one (i.e. **dk**). The obtained solution can be consequently optimised thanks to two strategy equivalences $\mathbf{dk}(S_1; S, S_2; S) = \mathbf{dk}(S_1, S_2); S$ and **if cond then** $S_1; S$ **else** $S_2; S$ **fi** = **if cond then** S_1 **else** S_2 **fi** ; S to the form in Figure 7.

5 Conclusion

The definition of the strategy language described in this paper is related to a view of strategies in reflective logics (in particular, rewriting logic) developed in [CM96]. From this point of view, the strategy language described here can be classified as an internal strategy language, whose semantics and implementation are described in the same logic, while the strategy language available in the first distribution of ELAN and described in [Vit94, Vit96] as a built-in one.

Another question which comes with the reflective approach is how to control computation at the meta-level.

```
[.]      [allsearch] state.level.space =>
          [dk(      call(exitable),
                    if occurs state in space then
                      next
                    else
                      dk(call(moves), next)
                    fi;
                    allsearch
          ]] state.level.space
end
```

Figure 7: All paths in the labyrinth

Roughly speaking, there are two levels of rewriting: the object (or first-order term) level, and the meta-level that controls the object-level. The first solution is that the meta-level computation (i.e. the evaluation of strategies) is controlled by a built-in strategy. But computation at the meta-level can also be controlled by meta-strategies. Using a reflective logic allows using the same formalism at all levels, which might be viewed as an advantage of a reflective approach.

From the specification point of view, writing programs in ELAN forces to think about the actions and their control in the same unified way, thus enforcing an equation of the form “Program = Logic + Control $\subseteq$ Logic”. What needs to be done is to develop a methodological way to construct program in this way in order to facilitate the use of these concepts by programmers.

The capability offered by this definition of strategies by rewrite rules leads also to a new approach to strategies compilation. This is a promising direction of our current research.

Acknowledgements: We sincerely thank the members of the Protheo group in Nancy and José Meseguer, Steven Eker, Manuel Clavel at SRI for many interesting discussions on strategies.

References

- [BK97] P. Borovanský and H. Kirchner. Strategies of ELAN: Meta-interpretation and partial evaluation. In A. Sellink, editor, *Proceedings of the Second International Workshop on the Theory and Practice of Algebraic Specifications ASF+SDF’97*, Amsterdam, September 1997. Workshops in Computing, Springer-Verlag.
- [BKK96a] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.
- [BKK⁺96b] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in TCS*, Asilomar (California), September 1996.
- [CELM96] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.
- [CK97] H. Cirstea and C. Kirchner. *Theorem proving using computational systems: The case of the B Predicate prover*. <http://www.loria.fr/~cirstea/Papers/PredicateProver.ps>, 1997.
- [CM96] 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.

- [GKK⁺87] J. A. Goguen, C. Kirchner, H. Kirchner, A. Mgrelis, J. Meseguer, and T. Winkler. An introduction to OBJ-3. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, July 1987. Also as internal report CRIN: 88-R-001.
- [KKV95] 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.
- [Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.
- [KM95] H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, volume 914 of *Lecture Notes in Computer Science*, pages 438–443. Springer-Verlag, 1995.
- [Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
- [MOM93a] N. Mart-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical report, SRI International, 1993.
- [MOM93b] N. Mart-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993. To appear in D. Gabbay, ed., *Handbook of Philosophical Logic*, Oxford University Press.
- [Vit94] M. Vittek. ELAN: *Un cadre logique pour le prototypage de langages de programmation avec contraintes*. Thse de Doctorat d’Universit, Universit Henri Poincar – Nancy 1, October 1994.
- [Vit96] M. Vittek. A compiler for nondeterministic term rewriting systems. In H. 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.