ELAN from the rewriting logic point of view

Peter Borovansky, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau

ELAN from the rewriting logic point of view

Peter Borovanský

Comenius University, Mlynská dolina, 842 15 Bratislava, Slovakia

Claude Kirchner, Hélène Kirchner and Pierre-Etienne Moreau

LORIA – CNRS & INRIA, BP 239 54506 Vandœuvre-lès-Nancy Cedex, France
email: Peter.Borovansky@fmph.uniba.sk,
Claude.Kirchner,Helene.Kirchner,Pierre-Etienne.Moreau@loria.fr

Abstract

ELAN implements computational systems, a concept that combines rewriting logic with the powerful description of rewriting strategies. ELAN can be used either as a logical framework or to describe and execute deterministic as well as non-deterministic rule based processes. With the general goal to make precise the semantics of ELAN, this paper has four contributions: a presentation of the concepts of rules and strategies available in ELAN, an expression of rewrite rules with matching conditions in conditional rewriting logic, an enrichment mechanism of a rewrite theory into a strategy theory in conditional rewriting logic, and eventually a description in conditional rewriting logic of the rules and strategies application mechanism in ELAN.

1 Introduction

The ELAN system provides an environment for specifying and prototyping deduction systems in a language based on rewrite rules controlled by strategies. It offers a natural and simple logical framework for the combination of the computation and deduction paradigms, as it is backed up by the concepts of rewriting logic and computational systems. It permits to support the design of theorem provers, logic programming languages, constraint solvers and decision procedures and to offer a modular framework for studying their combination.

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

ELAN programs define computational systems [KKV95,Vit94,BKK+96b] given by a signature providing the syntax, a set of conditional rewrite rules describing the deduction mechanism, and strategies to guide application of rewrite rules. ELAN programs are structured in modules, possibly parameterised and importing other modules.

ELAN takes from functional programming the concept of abstract data types and the function evaluation principle based on rewriting. In ELAN a rewrite rule may be labelled, may have boolean conditions and matching conditions, three notions that will be explained in this paper. One of the main originality of the language is to provide a strategy language allowing to specify the control on rules application. This is in contrast to many existing rewriting-based languages, where the term reduction strategy is hard-wired and not accessible to the designer of an application. The strategy language offers primitives for sequential composition, iteration, deterministic and non-deterministic choices of primal strategies that are labelled rules. From these primitives, more complex strategies can be expressed, by introducing new strategy operators and defining them again by rewrite rules. The operational semantic of strategies is itself based on rewriting. The evaluation mechanism also involves backtracking, since in ELAN, a computation may have several results.

One of the main features of ELAN is to consider rules and strategies as first-class objects. From the language point of view, this allows writing programs in a very natural way, having rules and strategies available at the same level. From the semantical point of view, strategy application can be expressed in rewriting logic itself, thanks to its reflexive capability. This approach is related to a view of strategies in reflective logics (in particular, rewriting logic) developed in [CM96].

ELAN has a functional semantics described in [BKK98], and logical foundations based on rewriting logic [Mes92,MOM93] and detailed in this paper, which is a revised extension of [BKK96a,KM96]. So the simple and well-known paradigm of term rewriting provides both the logical framework in which deduction systems can be expressed and combined, and the evaluation mechanism of the language. The notion of strategy as set of proof terms of a rewrite theory was proposed in [KKV95,Vit94]. Clearly, an arbitrary set of proof terms may be very complicated or irregular from the computational point of view (for instance, a non-recursive set). This is why we concentrate on languages describing special subclasses of strategies, called elementary and defined strategies. The main difference between these two classes is that elementary strategies are built from basic constructions predefined in ELAN, while defined strategies may be recursive, parameterised and typed as well.

The current version of ELAN includes an interpreter and a compiler written respectively in C++ and Java, a library of standard ELAN modules, a user

manual and examples of applications. Among those, let us mention for instance the design of rules and strategies for constraint satisfaction problems [Cas98], theorem proving tools in first-order logic with equality [KM95,CK98], the combination of unification algorithms and of decision procedures in various equational theories [Rin97,KR98]. More information on the system can be found on the WEB site http://www.loria.fr/ELAN.

This paper presents the logical foundation of ELAN using rewriting logic. After giving a few preliminary concepts and notations in Section 2, we describe, in Section 3, the concepts of rules, strategies, and a powerful notion of rules calling strategies in matching conditions, which is available in ELAN. We also explain the evaluation process for this general form of rules. The goal of the two next sections is to give a logical meaning to this class of rules in the context of rewriting logic. Section 4 shows how to express rewrite rules with matching conditions in conditional rewriting logic. Then, Section 5 is devoted to express strategies in this framework. We show how to enrich a rewrite theory into a strategy theory in conditional rewriting logic. The definition of the application operator of a strategy to a term by a set of rewrite rules with matching conditions, provides in turn a logical description of a strategy evaluator. At this point, the general form of rules presented in Section 3 is a formula in the extended strategy theory. The next step, performed in Section 6, is to formalise the rewriting mechanism as a set of rules controlled by a strategy. This provides a logical description of rule evaluation inside rewriting logic itself. Section 7 explains how these concepts are implemented in ELAN, and Section 8 presents a few applications developed using this system. To conclude, Section 9 draws some perspectives of further extensions of the language.

2 Preliminary concepts and notations

We assume the reader familiar with basic definitions of term rewriting given in particular in [DJ90,JK91,BN98]. We briefly recall and introduce notations for a few concepts that will be used along this paper. The definitions below are given in the many-sorted case. The order-sorted case can be handled in a similar, although more technical, manner.

We consider a set $\mathcal{S}$ of sort symbols, a set $\mathcal{F}$ of ranked function symbols, a set $\mathcal{X}$ of sorted variables and the set of first-order many-sorted terms $\mathcal{T}(\mathcal{F},\mathcal{X})$ built on $\mathcal{F}$ and $\mathcal{X}$. The rank of a function symbol f taking n arguments of respective sorts $s_1,\ldots,s_n$ and giving a result of sort s is denoted as follows: $f : (s_1,\ldots,s_n)\ s$. The arity of f is n. To simplify notation, we denote a sequence of objects $(a_1,\ldots,a_n)$ by $\overline{a}$ or $\overline{a}^n$.

A *substitution* is a sorted assignment from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F},\mathcal{X})$, written $\sigma = \{x_1 \mapsto$

$t_1, \ldots, x_k \mapsto t_k\}$. It uniquely extends to an endomorphism of $\mathcal{T}(\mathcal{F}, \mathcal{X})$. We also use the notations $t\{\overline{x} \mapsto \overline{w}\}$ to express the simultaneous substitution of w_i for x_i in t. Letters $\sigma, \mu, \gamma, \phi, \ldots$ denote substitutions. $\sigma\mu$ denotes the composition of μ and σ.

Positions in a term are represented as sequences of integers. The empty sequence ϵ denotes the position associated to the root symbol, and called the root (or top) position. The subterm of t at position ν is denoted $t_{|\nu}$. The replacement at position ν of the subterm $t_{|\nu}$ by t' is written $t[t']_\nu$. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. If $\mathcal{V}ar(t)$ is empty, t is called a *ground term* and $\mathcal{T}(\mathcal{F})$ is the set of ground terms. A term t is said to be *linear* if no variable occurs more than once in t.

A $\mathcal{T}(\mathcal{F}, \mathcal{X})$-equality is a directed pair of terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$, denoted as usual $(\forall X,\ t = t')$ where $X = \mathcal{V}ar(t) \cup \mathcal{V}ar(t')$. For any set of equalities E, $\mathcal{T}(\mathcal{F}, \mathcal{X})_{/E}$ denotes the free quotient algebra of terms modulo E. The equivalence class of a term t modulo E is denoted $\langle t \rangle_E$ or just $\langle t \rangle$. For details and general results on calculus modulo equational axioms, the reader is invited to consult for example [JK86].

We especially consider associative commutative theories, in which there is at least one binary function symbol f, that satisfies the following set AC of associativity and commutativity axioms:

$$\forall x, y, z,\ \ f(x, f(y, z)) = f(f(x, y), z) \tag{1}$$
$$\forall x, y,\ \ f(x, y) = f(y, x) \tag{2}$$

All such symbols are called AC function symbols. On the other hand, $\mathcal{F}_\emptyset$ is the subset of $\mathcal{F}$ made of function symbols which are not AC, and are called *free* function symbols. A term is said to be *syntactic* if it contains only free function symbols. We write $s =_{AC} t$ to indicate that the two terms s and t are equivalent modulo associativity and commutativity. The reader interested by associative commutative theories, and their interaction with rewriting can refer for instance to [PS81,JK86].

3 Rules and strategies

3.1 The rewrite rules

A *rewrite rule* is an ordered pair of terms denoted $l \to r$ such that $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. The term l is called the *left-hand side* or *pattern* and r is the *right-hand side*. A rewrite rule is said to be *syntactic* if

the left-hand side is a syntactic term. In order to get a better control on the application of the rewrite rules, conditions can be added. We define here an extended notion of condition, called *matching condition*, used in ELAN, but also in ASF+SDF [Kli93].

Definition 3.1 *A* labelled rewrite rule with matching conditions *denoted*

$$[\ell] \quad l \to r \ \textbf{\textit{where}}\ p := c$$

is such that $l, r, p, c \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $\mathcal{V}ar(p) \cap \mathcal{V}ar(l) = \emptyset$, $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p)$ *and* $\mathcal{V}ar(c) \subseteq \mathcal{V}ar(l)$. *Moreover* p *and* c *have the same sort.* ℓ *is called the label of the rule and may be omitted, in which case the rule is said unlabelled. When the term* p *(also called pattern) is the boolean constant true, then* c *must be a boolean term and the condition is usually written "****if*** c*".*

This notion of rule can be generalised with a sequence of conditions, as in

$$[\ell] \quad l \to r \ \textbf{where}\ p_1 := c_1 \ldots \ \textbf{where}\ p_n := c_n$$

with the following restrictions on variables:

- $l, r, p_1, \ldots, p_n, c_1, \ldots, c_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,
- $\forall i, i = 1, \ldots, n$, p_i and c_i have the same sort,
- $\forall i, i = 1, \ldots, n$, $\mathcal{V}ar(p_i) \cap (\mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})) = \emptyset$,
- $\forall i, i = 1, \ldots, n$, $\mathcal{V}ar(c_i) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})$ and
- $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_n)$.

Example 3.1 *The definition of the factorial function can be expressed by the two following unlabelled conditional syntactic rules :*

$$[]\ fact(0) \to 0$$

$$[]\ fact(n) \to n \times fact(n-1)\ \textbf{\textit{if}}\ n > 0$$

Example 3.2 *An operation* $doubleflat$ *on lists, that takes a list (for instance* $((1.2).(3.4))$*) and builds the concatenation of its flattened form with the reverse of its flattened form (in this case* $(1.2.3.4.4.3.2.1)$*) can be defined by the rule with two matching conditions:*

$$[]\ doubleflat(l) \to append(x, y)\ \textbf{\textit{where}}\ x := flatten(l)\ \textbf{\textit{where}}\ y := reverse(x)$$

The interest of this form with respect to the classical rule

$$[] \quad doubleflat(l) \ \to \ append(flatten(l), reverse(flatten(l)))$$

is indeed to factorise the expression of $flatten(l)$ and, with an adequate implementation, to avoid computing twice the flattened form of l.

The advantage of the **where** construction, similar to the *let* statement in ML, is to give to the programmer access to a direct control of sharing during the rewriting mechanism.

3.2 Rule application

To apply a syntactic rule $[\ell]\ \ l \to r$ on a term t at some position ν, one looks for a matching, i.e. a substitution σ satisfying $l\sigma = t_{|\nu}$. Note that t is always considered as a ground term. The algorithm which provides the unique substitution σ, whenever it exists, is called *syntactic matching*. Once a substitution σ is found, the application of the rewrite rule consists of building the *reduced term* $t' = t[r\sigma]_\nu$. Computing a normal form of a term t w.r.t. a rewrite system R consists of successively applying the rewrite rules of R, at every position, until no one applies any more. In order to ensure the existence and unicity of normal forms, the rewrite system R is required to be respectively (weakly) terminating and confluent.

To apply a rule $[\ell]\ \ l \to r$ **where** $p := c$ on a term t (with l and p two syntactic terms), the satisfiability of the condition $p := c$ has to be checked before building the reduced term. Let σ be the matching substitution from l to $t_{|\nu}$. Checking the matching condition $p := c$ consists first of using the rewrite system R to compute a normal form c' of $c\sigma$, when it exists, and then verifying that p matches the ground term c'. If there exists a substitution μ, such that $p\mu = c'$, the composed substitution $\sigma\mu$ is used to build the reduced term $t' = t[r\sigma\mu]_\nu$. Otherwise the application of the rule fails. Note that for usual boolean conditions of the form **if** c, μ is the identity when the normal form of $c\sigma$ is *true*. It may also happen that no normal form is found for $c\sigma$, in which case the rule is said non-terminating.

When the rule is of the form

$$[\ell]\ \ l \to r \ \textbf{where}\ p_1 := c_1 \ldots\ \textbf{where}\ p_n := c_n$$

the matching substitution is successively composed with each matching μ_i from p_i to the normal forms of $c_i\sigma\mu_1 \ldots \mu_{i-1}$, for $i = 1, \ldots, n$. If one of these μ_i does not exist, the application of the rule fails. If no normal form is found at some step i, the rule does not terminate.

When the left-hand side of the rule contains AC function symbols, AC matching is invoked. The term l is said to AC match another term t if there exists a substitution σ such that $l\sigma =_{AC} t$. AC matching has already been extensively

studied, for instance in [Hul80,BKN87,KL91,Dom92,BCR93,LM94,Eke95]. In general, AC matching can return several solutions, which introduces a need for backtracking for conditional rules: as long as there is a solution to the AC matching problem for which the matching condition is not satisfied, another solution has to be extracted. If the pattern p contains AC function symbols, a general one-to-one AC matching procedure is used [Eke95] to find a substitution μ such that $p\mu = c'$. Only when all solutions have been tried unsuccessfully, the application of this conditional rule fails. When the rule contains a sequence of matching conditions, failing to find a match for the i-th condition causes a backtracking to the previous one. So, in practice, conditional rewriting requires AC matching problems to be solved in a specific way: the first solution has to be found as fast as possible, and the others have to be provided one by one on request.

Example 3.3 *A typical case of application for associative and commutative theories is the axiomatisation of multisets. The structure of multisets of integers is described by the following operators:*

$$\begin{aligned} \_ \cup \_ &: (mset \quad mset) \quad mset \\ \emptyset \quad &: mset \\ \_ \quad &: (int) \quad mset \end{aligned}$$

where $\cup$ is AC and the third operator is a (no-named) coercion from the sit int into the sort mset.

A simple rule extracts one element of the multiset:

$$[]\ i \cup m \to i$$

Application of this rule to the term $\emptyset \cup 1 \cup 2 \cup 3 \cup 4 \cup 5$
produces five possible results $1, 2, 3, 4, 5$.

In some cases, one may want to get all solutions of an AC matching problem and build all possible results of rewriting with these different matching substitutions. It is then suitable to consider that application of a rule to a term produces a multiset of results that may be empty, a singleton, a finite, or an infinite multiset of reduced terms. Note that the notion of multiset is useful here since identical results may be produced by different computations.

Example 3.4 *As another example of the use of associative-commutative symbols, let us consider the following unlabelled rule, where the union symbol $\cup$ is*

again an associative-commutative operator, and f is a free function symbol.

$$[]\ f(m_1, m_2) \rightarrow f(p_1, p_2)\ \textbf{\textit{where}}\ x \cup p_1 := m_1$$
$$\textbf{\textit{where}}\ y \cup p_2 := m_2$$
$$\textbf{\textit{if}}\quad x = y$$

This rule has a condition and two matching conditions which involve AC-matching. When applying this rule on two multisets m_1 and m_2, it removes identical elements and returns resulting multisets p_1 and p_2. It is also possible to express the same algorithm in a more compact way:

$$[]\ f(x \cup p_1, x \cup p_2) \rightarrow f(p_1, p_2)$$

Here associative commutative matching alone performs the search for identical elements among the multisets. However the first form is more flexible, since it would allow for instance application of another function on m_1 or m_2.

The last example illustrates the fact that the patterns in **where** statements are not instantiated by the substitutions computed in previous matching conditions. This is merely a design decision, motivated by the wish to keep programs more readable.

The notion of matching condition could be generalised to unification condition, where full unification is used to instantiate variables in the **where** statement. This would enhance the expressive power of the language, but immediately raises the problem of using narrowing at the operational level.

3.3 Strategies

Rules are not only used for function evaluation in Computer Science, and they appear in various areas ranging from expert systems to logical frameworks. In most cases, rules are neither terminating nor confluent, but their application needs to be controlled precisely. Plans in expert systems, or tactics in logical frameworks have been introduced for this purpose. The need for expressing control also appears in deduction systems.

Example 3.5 *Let us consider for instance a sequent calculus, implemented*

using inference rules, among which are the following ones:

$$
\begin{array}{lll}
[\mathsf{negg}] & H, \neg P \vdash Q & \rightarrow H \vdash P, Q \\
[\mathsf{conjg}] & H, (P \wedge Q) \vdash R & \rightarrow H, P, Q \vdash R \\
[\mathsf{disjg}] & H, (P \vee Q) \vdash R & \rightarrow H, P \vdash R :: H, Q \vdash R \\
[\mathsf{impg}] & H, (P \Rightarrow Q) \vdash R & \rightarrow H \vdash P, R :: H, Q \vdash R \\
\ldots & &
\end{array}
$$

We may want to control application of these rules in such a way that repeated application of rules negg *and* conjg *as long as possible is performed before either rule* disjg *or* impg *is applied.*

In order to control rule application and to take into account multisets of results, we introduce the concept of strategy: a strategy is a function which, when applied to an initial term, returns a multiset of results. The strategy fails if the multiset is empty. To precisely define how multisets of results are handled, we introduce the following strategy constructors.

- A labelled rule is a *primal* strategy. Applying a rule labelled ℓ at the root position of a term t results in a multiset of terms. This primal strategy fails if the multiset of resulting terms is empty.
- Two strategies can be concatenated by the symbol ";", i.e. the second strategy is applied on all results of the first one. $S_1; S_2$ denotes the sequential composition of the two strategies. It fails if either S_1 fails or S_2 fails on all results of S_1. Its results are all results of S_1 on which S_2 is successfully applied.
- $\mathsf{dk}(S_1, \ldots, S_n)$ applies all strategies given in the list of arguments and for each of them returns all its results. This multiset of results may be empty, in which case the strategy fails.
- $\mathsf{dc}(S_1, \ldots, S_n)$ chooses one strategy S_i in the list that does not fail, and returns all its results. This strategy fails when all sub-strategies S_i fail.
- $\mathsf{first}(S_1, \ldots, S_n)$ chooses the first strategy S_i in the list that does not fail, and returns all its results. This strategy fails when all sub-strategies S_i fail.
- $\mathsf{dc\_one}(S_1, \ldots, S_n)$ chooses one strategy S_i in the list that does not fail, and returns one of its first results. This strategy returns at most one result or fails if all sub-strategies fail.
- $\mathsf{first\_one}(S_1, \ldots, S_n)$ chooses the first strategy S_i in the list that does not fail, and returns one of its first results. This strategy returns at most one result or fails if all sub-strategies fail.
- The strategy id is the identity that never fails.
- fail is the strategy that always fails and never gives any result.
- $\mathsf{repeat*}(S)$ applies repeatedly the strategy S until it fails and returns the

results of the last unfailing application. This strategy can never fail (zero application of S is always possible) and may return more than one result.
- The strategy iterate*(S) is similar to repeat*(S) but returns all intermediate results of repeated applications.

Let us emphasise that, by definition of primal strategies, any labelled rule is a strategy. In order to precisely describe the meaning of the operators dk, dc, first, dc_one, first_one, it is helpful to introduce four auxiliary operators that allow us to differentiate between two different levels of control:
Controlling the number of results in the result set: given a strategy,
- the one operator builds a strategy that returns at most one result;
- the all operator builds a strategy that returns all possible results of the strategy.

Controlling the choice of the strategy in the list of strategies: given a list of strategies,
- the select_one operator chooses and returns a non-failing strategy among the list of strategies;
- the select_first operator chooses and returns the first (from left to right) non-failing strategy among the list of strategies;
- the select_all operator returns all unfailing strategies.

Using these four primitives, the strategy constructors dk, dc, first, dc_one and first_one can then be defined by the following axioms, where S_i stands for a strategy:

$$
\begin{array}{lcl}
\mathsf{dk}(S_1,\ldots,S_n) & = & \mathsf{select\_all}(\mathsf{all}(S_1),\ldots,\mathsf{all}(S_n)) \\
\mathsf{dc}(S_1,\ldots,S_n) & = & \mathsf{select\_one}(\mathsf{all}(S_1),\ldots,\mathsf{all}(S_n)) \\
\mathsf{first}(S_1,\ldots,S_n) & = & \mathsf{select\_first}(\mathsf{all}(S_1),\ldots,\mathsf{all}(S_n)) \\
\mathsf{dc\_one}(S_1,\ldots,S_n) & = & \mathsf{select\_one}(\mathsf{one}(S_1),\ldots,\mathsf{one}(S_n)) \\
\mathsf{first\_one}(S_1,\ldots,S_n) & = & \mathsf{select\_first}(\mathsf{one}(S_1),\ldots,\mathsf{one}(S_n))
\end{array}
$$

Note that dk, dc and first operators are equivalent if they are applied on a unique argument: $\mathsf{dk}(S) = \mathsf{dc}(S) = \mathsf{first}(S) = S$.

Example 3.6 *Let us come back to Example 3.3 on multisets, with the following rule, which has now a label* R*:*

$$[\mathsf{R}]\ i \cup m \to i$$

Remind that application of this rule to the term $\emptyset \cup 1 \cup 2 \cup 3 \cup 4 \cup 5$ *pro-*

duces five possible results $1, 2, 3, 4, 5$. *The strategies* dk(R), dc(R) *and* first(R) *are all equivalent in this case and enumerate the multiset of results. On the other hand,* dc_one(R) *produces only one among these results, and* first_one(R) *returns the result corresponding to the first found match.*

Let us explain now how to express user-defined strategies with this language.

The easiest way to build a strategy is to use the strategy constructors to build strategy terms and to define a new constant operator that denotes this (more or less complex) strategy expression. This gives rise to a first class of strategies called elementary strategies. Elementary strategies are defined by unlabelled rules of the form $[]\ S \rightarrow strat$, where S is a constant strategy operator and $strat$ a term built on predefined strategy constructors and rule labels, but that does not involve S. The application of a strategy S on a term t is denoted $[S](t)$.

Let us begin with an example of elementary strategies.

Example 3.7 *Two labelled rewrite rules can be defined in order to extract the head and the tail of a list:*

$$[\mathsf{head}]\ elem(e \cdot l) \rightarrow e$$
$$[\mathsf{tail}]\quad elem(e \cdot l) \rightarrow elem(l)$$

These two rules are applied repeatedly according to the following strategy:

$$[]\ listExtract \rightarrow \mathit{iterate}^*(\mathsf{tail}); \mathsf{head}$$

The listExtract strategy controls the application of the two previous rules, as follows: first, tail *is applied 0 time and the head of the list is returned, then,* tail *is applied once and the head of the new tail is returned, and so on. When l is empty, the rule* tail *fails and the strategy listExtract stops. So for a given list, the application of the strategy listExtract allows extracting all its elements one by one "on request".*

However this is not expressive enough to allow recursive and parameterised strategies. This is why the more general notion of defined strategies is introduced. Their definition is given by a strategy operator with a rank, and a set of labelled rewrite rules. In order to illustrate now defined strategies, let us take the example of a map functor on lists.

Example 3.8 *Let us introduce an operator* map *by its rank:*

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

where $\langle s \mapsto s \rangle$ *and* $\langle list[s] \mapsto list[s] \rangle$ *are strategy sorts. The argument of* map *must be a strategy* S *that applies to a term of sort* s *and returns results of sort* s*. The strategy* $\mathsf{map}(S)$ *applies to a term of sort* $list[s]$ *and returns results of sort* $list[s]$*.*

The strategy map *is defined by a rewrite rule:*

$$\mathsf{map}(S) \to dc(\mathsf{nil}, S \cdot \mathsf{map}(S)) \qquad (1)$$

where S *is a variable of sort* $\langle s \mapsto s \rangle$*. The right-hand side of this definition means that whenever the strategy* $\mathsf{map}(S)$ *is applied to a term* t*, either* t *is* nil*, or the strategy* S *is applied to the head of* t *(i.e.* t *should be a non-empty list) and* $\mathsf{map}(S)$ *is further applied to the tail of* t*.*

This strategy definition substantially differs from the traditional functional definition (cf. [MJ95] for instance) of the functor

$$\mathsf{map} \ : \ (s \to s) \to (list[s] \to list[s])$$

$$\begin{aligned} \mathsf{map}\ f\ nil &= nil \qquad (2) \\ \mathsf{map}\ f\ (e \cdot l) &= (f\ e) \cdot (\mathsf{map}\ f\ l) \end{aligned}$$

However, a similar strategy definition can be formulated also using the strategy application symbol $[\_](\_)$:

$$\begin{aligned} [\mathsf{map}(S)](nil) &\to nil \qquad (3) \\ [\mathsf{map}(S)](e \cdot l) &\to [S](e) \cdot [\mathsf{map}(S)](l) \end{aligned}$$

The difference relies on the fact that the list, which the functional map is applied to, is an explicit argument in the two last definitions, while in the first rule (1) of Example 3.8, it is implicit.

3.4 Rules involving strategies in conditions

In our approach up to now, labelled rules and strategies can only be applied at the root position of a given term. Indeed this is not enough and we need to provide the possibility to apply labelled rules and strategies inside expressions. This facility is given by allowing strategy calls in matching conditions.

Let us now consider a more general form of a rule, defined as follows:

$$[\ell] \ \ l \to r \ \textbf{where} \ p_1 := [S_1](c_1) \ \ \ldots \ \ \textbf{where} \ p_n := [S_n](c_n)$$

where

- $l, r, p_1, \ldots, p_n, c_1, \ldots, c_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,
- $\forall i, i = 1, \ldots, n$, p_i and $[S_i](c_i)$ have the same sort.
- $\forall i, i = 1, \ldots, n$, $\mathcal{V}ar(p_i) \cap (\mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})) = \emptyset$,
- $\forall i, i = 1, \ldots, n$, $\mathcal{V}ar(c_i) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})$,
- $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_n)$,
- $S_1, \ldots, S_n$ are strategy terms and $[\_](\_)$ is the application operator of strategies on terms.

The evaluation of a generalised matching condition $p_i := [S_i](c_i)$ involves the evaluation of c_i and S_i first, and then of the application operator $[\_](\_)$. In general, this leads to a multiset of terms. Finally, the pattern p_i is matched with each result in this multiset. If either the multiset is empty, or the matching condition is not satisfied, then the evaluation backtracks to the previous matching condition; otherwise, the evaluation sets a choice point and goes on with one of the returned terms.

Example 3.9 *We could have defined the strategy operator* ***map*** *in a different way using the* ***where*** *construction with strategies:*

$$
\begin{aligned}
&[]\ [\mathsf{map}(S)](nil) \ \rightarrow nil \\
&[]\ [\mathsf{map}(S)](h \cdot t) \rightarrow h_1 \cdot t_1 \ \textbf{\textit{where}}\ h_1 := [S](h) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{\textit{where}}\ t_1 := [\mathsf{map}(S)](t)
\end{aligned}
$$

3.5 Strategies on strategy evaluation

A rule like $\mathsf{map}(S) \rightarrow \mathsf{dc}(\mathsf{nil}, S \cdot \mathsf{map}(S))$ recursively defines the strategy map. But applied without special strategy, it may lead to infinite computations. This justifies the concept of meta-strategies, i.e. strategies that control the rewriting of defined strategies.

Typically such a rule gives rise to a labelled rule

$$[\mathsf{DSTR}] \quad [\mathsf{map}(S)](x) \rightarrow [\mathsf{dc}(\mathsf{nil}, S \cdot \mathsf{map}(S))](x)$$

where x is a variable of sort s. The strategy given for the operator $[\_](\_)$ controls application of this rule, using its label [DSTR].

Once strategy operators are defined, strategy terms are available and may be rewritten too. Rules that define a computation on strategy terms are evaluated exactly like rules on (ordinary) terms according to their labels and strategies

that involve these labels.

Example 3.10 *A few rules can be defined for instance to reduce strategy expressions. Let us consider for instance*

$$\begin{aligned} &[]\ \mathsf{map}(S;S') &&\to \mathsf{map}(S);\mathsf{map}(S')\\ &[]\ \mathsf{map}(\mathit{id}) &&\to \mathit{id}\\ &[]\ \mathsf{map}(\mathit{fail}) &&\to \mathit{fail} \end{aligned}$$

When they are unlabelled as above, the rules are used by the evaluator to eagerly simplify the terms before each application of a labelled rule. When they are labelled as below:

$$\begin{aligned} &[\mathsf{c}]\ \mathsf{map}(S;S') &&\to \mathsf{map}(S);\mathsf{map}(S')\\ &[\mathsf{i}]\ \mathsf{map}(\mathit{id}) &&\to \mathit{id}\\ &[\mathsf{f}]\ \mathsf{map}(\mathit{fail}) &&\to \mathit{fail} \end{aligned}$$

the rule application must be controlled by a user defined strategy such as

$$[]\ \mathit{smap} \to \mathit{repeat}^*(\mathit{dk}(\mathsf{c},\mathsf{i},\mathsf{f}))$$

and they are only applied at the root of terms with map as root symbol.

Another example of simplification rules on strategy terms is given by simplification rules for elementary strategies:

$$\begin{aligned} &[]\ \mathsf{id}\ ;\ \mathsf{S} \to S && \qquad &[]\ \mathsf{S}\ ;\ \mathsf{id} \to S\\ &[]\ \mathsf{dc}(\mathsf{S},\ \mathsf{S}) \to S && &[]\ \mathsf{dk}(\mathsf{S},\ \mathsf{S}) \to S \end{aligned}$$

More sophisticated examples of using the strategy language can be found in in [Bor98].

3.6 Labelled and unlabelled rules

The evaluation process considered here makes an important difference between labelled and unlabelled rules.

– **Labelled rules** are applied under the full control of strategies and only at the root of terms.

– **Unlabelled rules** are intended to perform functional evaluation. Their lack of name means that they are supposed to be controlled by an implicit strategy, and actually they are applied using a leftmost innermost strategy. The set of unlabelled rules is required to be confluent and terminating for the leftmost innermost strategy. Of course, a sufficient condition is to have a confluent and terminating system which can be obtained through standard completion techniques. Few results are known on proving these properties under specific strategies [Gra96,Art97].

During the evaluation process, unlabelled rules are applied eagerly, before each application of labelled rules. Assuming that the set of unlabelled rules E is confluent and terminating, the congruence $=_E$ is decidable and a canonical representant of equivalence classes modulo E is the common E-normal form of all terms in the equivalence class. In presence of associative commutative axioms, E is an equational term rewriting systems and the notions of termination and confluence have to be refined to take into account the axioms. The interested reader can refer to [JK86,Kir95] for more details on this case. Labelled rules are applied on root of terms which are always in E normal form. This kind of rewriting process has been studied in [Mar94,Vir96].

Introducing these two kinds of rules gives the possibility to distinguish between computations, performed by unlabelled rules, and deductions, performed by labelled rules under certain strategies. This is a very convenient approach to implement deduction modulo [DHK98], or what is called in [BB97] the Poincaré Principle. The interest of distinguishing between logical deduction and computation in various areas of automated deduction and program construction is argued in [Dow99]. ELAN can be considered as an implementation of these ideas in a first-order logic based context.

One goal of this paper is now to formalise this powerful kind of rules calling strategies in matching conditions, within conditional rewriting logic. We will proceed in two steps: first we recall conditional rewriting logic, and express sets of rewrite rules with matching conditions as theories of this logic. Second, we propose a canonical way to extend such theories by strategy operators and rules, and by the definition of the application operator $[\_](\_)$. In these extended theories, axioms are now the more general form of rules presented above.

4 Rewriting logic for rewrite rules with matching conditions

In this section, we use conditional rewriting logic [Mes92] to formalise deduction with rewrite rules with matching conditions, and to take into account normalisation with unlabelled rewrite rules, by working in equivalence classes of terms. In order to make the paper self-contained, we recall the main defi-

nitions.

Syntax. The syntax needed for defining a logic is provided by a signature which allows building sentences. In rewriting logic, a signature consists of a 4-tuple $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{L}, E)$, where $\mathcal{S}$ is a set of sort symbols, $\mathcal{L}$ and $\mathcal{F}$ are sets of ranked function symbols and E is a set of $\mathcal{T}(\mathcal{F}, \mathcal{X})$-equalities. In our approach, we consider E as the union of structural axioms A (such as commutativity and associativity) with a set U of unlabelled conditional rewrite rules with matching conditions. The equational rewrite system (A, U) is assumed to be confluent and terminating modulo A. Achieving these properties may involve advanced saturation techniques from automated theorem proving, such as proposed in [Vig94,RV95,BGLS95,NN93]. Equality in the theory E can be decided by rewriting modulo A with U and then checking A equivalence of the results.

Sentences built on a given signature are defined as sequents of the form $\pi : \langle t \rangle \rightarrow \langle t' \rangle$ where $t, t' \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and π is the proof term representing the proof that allows to derive t' from t. So in rewriting logic, proofs are first-order objects identified with proof terms. In order to compose proofs, we introduce the infix binary operator "$\_;\_$" on proof terms. In order to record subproofs corresponding to matching conditions, we introduce the operator "$\_\{\_\}$" whose second argument is a list of subproofs of the first argument. Therefore a proof term is by definition a term built on equivalence classes of $\mathcal{T}(\mathcal{F}, \mathcal{X})_{/E}$, function symbols in $\mathcal{F}$, label symbols in $\mathcal{L}$, the composition operator "$\_;\_$", the subproof operator "$\_\{\_\}$". In other words, the set of proof terms is the domain of the term algebra $\Pi = \mathcal{T}(\mathcal{L} \cup \{\_;\_\ ,\ \_\{\_\}\} \cup \mathcal{F} \cup \mathcal{T}(\mathcal{F}, \mathcal{X})_{/E})$.

In order to be generic, we consider $\mathcal{S}ynt$ a class of pairs (Σ, sen) consisting of a signature Σ together with a mapping sen associating to Σ the set of all legal sentences built on this signature.

Entailment systems. For a given class of syntax $\mathcal{S}ynt$ and (Σ, sen) in $\mathcal{S}ynt$, a theory $\mathcal{RT}$ presented by a set of axioms Φ is the pair $\mathcal{RT} = (\Sigma, \Phi)$ where $\Phi \subseteq sen(\Sigma)$. Given a signature Σ, an entailment system is an abstract description of the provability relation of a sentence ϕ starting from a given set of sentences (also called axioms) Φ and using logical rules.

In rewriting logic, in order to build the entailment system, the notion of rewrite theory is introduced and an appropriate deduction system allows us to inductively define the entailment relation.

A labelled rewrite theory is $\mathcal{RT} = (\Sigma, \mathcal{X}, \mathcal{R})$ where $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{L}, E)$ is a signature in rewriting logic, $\mathcal{X}$ is a given countably infinite set of variables,

and $\mathcal{R}$ is a set of labelled rewrite rules. To indicate that $\{x_1, \ldots, x_n\}$ is the set of variables occurring in a term u, we write $u(x_1, \ldots, x_n)$ or $u(\overline{x})$. In [Mes92], rewrite rules are of the form

$$[\ell(\overline{z})]\ \langle l(\overline{z})\rangle \to \langle r(\overline{z})\rangle$$
$$\textbf{if } \langle u_1(\overline{z})\rangle \to \langle v_1(\overline{z})\rangle \wedge \ldots \wedge \langle u_k(\overline{z})\rangle \to \langle v_k(\overline{z})\rangle$$

Here we consider labelled rewrite rules with matching conditions of the form:

$$[\ell(\overline{x})\{\overline{y_1} \cdot \ldots \cdot \overline{y_n}\}]\ \langle l(\overline{x})\rangle \to \langle r(\overline{x}, \overline{y_1}, \ldots, \overline{y_n})\rangle$$
$$\textbf{where } \langle p_1(\overline{y_1})\rangle := \langle c_1(\overline{x})\rangle \wedge \ldots \wedge \langle p_n(\overline{y_n})\rangle := \langle c_n(\overline{x}, \overline{y_1}, \ldots, \overline{y_{n-1}})\rangle$$

where the matching conditions are gathered in a sequential conjunction, later denoted by C for short, i.e. a term of $\mathcal{T}(\mathcal{F} \cup \{\wedge\}, \mathcal{X})$. Note however that the symbol $\wedge$ is not commutative in C. Let $\mathcal{L}(\mathcal{X})$ be the set of linear terms of the form $\ell(\overline{x})\{\overline{y_1} \cdot \ldots \cdot \overline{y_n}\}$, in which ℓ is a rewrite rule label from $\mathcal{L}$ and $\overline{x}, \overline{y_1}, \overline{y_n}$ are pairwise different variables occurring in this rewrite rule. $\overline{x}$ are variables occurring in the left-hand side and $\overline{y_i}$ are variables introduced in the pattern of the i-th matching condition. $\mathcal{R}$ is a subset of $\mathcal{L}(\mathcal{X}) \times \mathcal{T}(\mathcal{F}, \mathcal{X}) \times \mathcal{T}(\mathcal{F}, \mathcal{X}) \times \mathcal{T}(\mathcal{F} \cup \{\wedge\}, \mathcal{X})$.

A matching condition $p(\overline{y}) := c(\overline{x})$ corresponds to the computation (modulo E-equivalence classes) of a normal form of a ground instance of $c(\overline{x})$ into a term c', followed by a match from $p(\overline{y})$ to the result. If this condition succeeds, the term c' must be of the form (or AC-equivalent to) $p(\overline{v'})$, where $\overline{y} \mapsto \overline{v}$ is the matching substitution, and where $\langle \overline{v}\rangle \to \langle \overline{v'}\rangle$ denote the (simultaneous) derivations in the substitution part.

So the labelled rewrite rule with matching conditions above is a special case of conditional rules allowed in [Mes92], as they can be written:

$$[\ell(\overline{z})]\ \langle l(\overline{z})\rangle \to \langle r(\overline{z})\rangle$$
$$\textbf{if } \langle c_1(\overline{z})\rangle \to \langle p_1(\overline{z})\rangle \wedge \ldots \wedge \langle c_n(\overline{z})\rangle \to \langle p_n(\overline{z})\rangle$$

where $\overline{z}$ is the sequence of all variables $\overline{x}, \overline{y_1}, \ldots, \overline{y_n}$ occurring anywhere in the rule. In the following, we will freely choose the most adequate representation of rules, according to the considered problem. We will also use the more concise form $[\ell(\overline{z})]\ l \to r$ **where** C.

A given labelled rewrite theory $\mathcal{RT}$ entails the sequent $\pi\ :\ \langle t\rangle \to \langle t'\rangle$, which is denoted $\mathcal{RT} \vdash \pi\ :\ \langle t\rangle \to \langle t'\rangle$, if $\pi\ :\ \langle t\rangle \to \langle t'\rangle$ is obtained by finite application of the deduction rules in Figure 1. The notation $\overline{\alpha}\ :\ \langle \overline{w}\rangle \to \langle \overline{w'}\rangle$ is used to abbreviate k sequents $\alpha_j\ :\ \langle w_j\rangle \to \langle w'_j\rangle$ for $j = 1, \ldots, k$.

Reflexivity. For all $t \in \mathcal{T}(\mathcal{F})$

$$\langle t \rangle \ : \ \langle t \rangle \to \langle t \rangle$$

Congruence

$$\frac{\pi_i \ : \ \langle t_i \rangle \to \langle t_i' \rangle, \ i = 1, \ldots, n}{f(\pi_1, \ldots, \pi_n) \ : \ \langle f(t_1, \ldots, t_n) \rangle \to \langle f(t_1', \ldots, t_n') \rangle}$$

$$[\ell(\overline{x})\{\overline{y_1} \cdot \ldots \cdot \overline{y_n}\}] \ \langle l(\overline{x}) \rangle \to \langle r(\overline{x}, \overline{y_1}, \ldots, \overline{y_n}) \rangle$$

where $\langle p_1(\overline{y_1}) \rangle := \langle c_1(\overline{x}) \rangle \wedge \ldots \wedge \langle p_n(\overline{y_n}) \rangle := \langle c_n(\overline{x}, \overline{y_1}, \ldots, \overline{y_{n-1}}) \rangle$

Replacement. For each rewrite rule

$[\ell(\overline{x})\{\overline{y_1} \cdot \ldots \cdot \overline{y_n}\}] \quad \langle l(\overline{x}) \rangle \to \langle r(\overline{x}, \overline{y_1}, \ldots, \overline{y_n}) \rangle$

where $\langle p_1(\overline{y_1}) \rangle := \langle c_1(\overline{x}) \rangle \wedge \ldots \wedge \langle p_n(\overline{y_n}) \rangle := \langle c_n(\overline{x}, \overline{y_1}, \ldots, \overline{y_{n-1}}) \rangle$

$$\frac{\begin{array}{r} \overline{\alpha} \ : \ \langle \overline{w} \rangle \to \langle \overline{w'} \rangle \\ \overline{\beta_i} \ : \ \langle \overline{v_i} \rangle \to \langle \overline{v_i'} \rangle, \ i = 1, \ldots, n \\ \gamma_i(\overline{\alpha}, \overline{\beta_1}, \ldots, \overline{\beta_i}) \ : \ \langle c_i(\overline{w}, \overline{v_1}, \ldots, \overline{v_{i-1}}) \rangle \to \langle p_i(\overline{v_i}) \rangle \ i = 1, \ldots, n \end{array}}{\ell(\overline{\alpha})\{\gamma_1(\overline{\alpha}) \cdot \ldots \cdot \gamma_n(\overline{\alpha}, \overline{\beta_1}, \ldots, \overline{\beta_n})\} \ : \ \langle l(\overline{w}) \rangle \to \langle r(\overline{w'}, \overline{v_1'}, \ldots, \overline{v_n'}) \rangle}$$

Transitivity

$$\frac{\pi_1 \ : \ \langle t_1 \rangle \to \langle t_2 \rangle \quad \pi_2 \ : \ \langle t_2 \rangle \to \langle t_3 \rangle}{(\pi_1; \pi_2) \ : \ \langle t_1 \rangle \to \langle t_3 \rangle}$$

Fig. 1. Deduction rules for Rewriting Logic

It is worth emphasizing the slight differences with conditional rewriting logic. in addition to the fact that we consider here a more general class of theories for E, there are two differences concerning labelled rewrite rules: first, the conjunction of conditions is not commutative, since the order of matching conditions is relevant in our approach; second, the instantiated conditions c_i are normalised, whenever possible, into instances of p_i; finally, concerning the deduction rules, one should notice that the **Replacement** rule is similar to its version in [Mes92], except for the structuration of proof terms. Its hypotheses can be split into two parts: the first one (with proof terms α_i and β_i) for the deductions in the substitution part, the second one (with proof terms γ_i) for the deductions in the matching conditions part, which appear as concatenated subproofs in the final proof term. The structuration is actually just a syntactical facility that could be handled in the same way as conditions in [Mes92]. Especially the proof term $\ell(\overline{\alpha})\{\gamma_1(\overline{\alpha}) \cdot \ldots \cdot \gamma_n(\overline{\alpha}, \overline{\beta_1}, \ldots, \overline{\beta_n})\}$ is actually another

notation for a proof term $\ell(\overline{\delta}, \gamma_1, \ldots, \gamma_n)$ in the notation of [Mes92], where $\overline{\delta}$ stands for $\overline{\alpha}, \overline{\beta_1}, \ldots, \overline{\beta_n}$. The former notation is better for displaying the proof term in a more structured way. This correspondence between the two notations allows directly reusing the results of [Mes92] on equivalent proof terms. An equivalence on proof terms is defined by E and a set E_R of equational axioms described in Figure 2. This equivalence relation is important to relate different derivations with the same result, but different proofs.

$\forall \pi_1, \pi_2, \pi_3 \in \Pi \quad \pi_1; (\pi_2; \pi_3) = (\pi_1; \pi_2); \pi_3$	**Associativity**
$\forall \pi : \langle t \rangle \to \langle t' \rangle, \quad \pi; \langle t' \rangle = \pi, \quad \text{and} \quad \langle t \rangle; \pi = \pi$	**Local Identities**
$\forall f \in \mathcal{F}\ of\ arity\ n, \quad \forall \pi_1, \ldots, \pi_n, \pi'_1, \ldots, \pi'_n$: $f(\pi_1; \pi'_1, \ldots, \pi_n; \pi'_n) = f(\pi_1, \ldots, \pi_n); f(\pi'_1, \ldots, \pi'_n)$	**Independence**
$\forall [\ell] : g \to d \in {}_{,}R, \forall \pi_1 : \langle t_1 \rangle \to \langle t'_1 \rangle, \ldots, \pi_n : \langle t_n \rangle \to \langle t'_n \rangle$ $\ell(\pi_1, \ldots, \pi_n) = \ell(\langle t_1 \rangle, \ldots, \langle t_n \rangle); d(\pi_1, \ldots, \pi_n)$ and $\ell(\pi_1, \ldots, \pi_n) = g(\pi_1, \ldots, \pi_n); \ell(\langle t'_1 \rangle, \ldots, \langle t'_n \rangle)$	**Parallel Move Lemma**

Fig. 2. E_R: Equivalence of proof terms

5 Rewrite theories of strategies

In this section, we show how to extend a theory of rewriting logic in order to define a theory for elementary and defined strategies. In this extended theory, it is then possible to write the generalised form of rewrite rules with strategies in matching conditions. In rewriting logic, each rewrite theory $\mathcal{RT}$ is enriched by new strategy sorts, new strategy operators and new rules, to obtain a strategy theory $\mathcal{RT}_{\mathcal{ES}}$.

This section provides an axiomatisation via many-sorted rewrite theories of strategies, with the assumption that rewrite rules in the user's rewrite theory have their left and right-hand sides of the same sort. This also means that any derivation preserves the sort of terms. This has been extended to non-sort preserving rules in [Bor98].

The rewrite theory for strategies is an extension of the user's many-sorted rewrite theory $\mathcal{RT}$, given by

$$\mathcal{RT} = ((\mathcal{S}, \mathcal{F}, \mathcal{L}, E), \mathcal{X}, \mathcal{R}).$$

The rewrite theory of strategies

$$\mathcal{RT}_{\mathcal{ES}} = ((\mathcal{S}_{\mathcal{ES}}, \mathcal{F}_{\mathcal{ES}}, \mathcal{L}_{\mathcal{ES}}, E_{\mathcal{ES}}), \mathcal{X}_{\mathcal{ES}}, \mathcal{R}_{\mathcal{ES}})$$

extends the theory $\mathcal{RT}$ as defined below. We adopt here the flat and abbreviated representation of rules $[\ell(\overline{z})]\ l \to r$ **where** C where $\overline{z}$ is split into $\mathcal{V}ar(l) = \{z_1, \ldots, z_k\}$ and $\mathcal{V}ar(p_1) \cup \ldots \cup \mathcal{V}ar(p_n) = \{z_{k+1}, \ldots, z_m\}$.

Sorts: $\mathcal{S}_{\mathcal{ES}}$ is the disjoint union of $\mathcal{S} \cup \mathcal{S}_{\mathcal{E}} \cup \mathcal{S}_{\mathcal{I}}$ defined as follows:

- $\mathcal{S}_{\mathcal{E}}$ is the set of strategy sorts, recursively defined by:
  $\mathcal{S}_{\mathcal{E}} = \{\langle s \mapsto s \rangle \mid s \in \mathcal{S} \cup \mathcal{S}_{\mathcal{E}}\}$

- $\mathcal{S}_{\mathcal{I}}$ consists of all sorts $\mathbf{Set}(s)$, for all $s \in \mathcal{S} \cup \mathcal{S}_{\mathcal{E}}$. They are used to type multisets of results.

Variables: Variables are taken from a set $\mathcal{X}_{\mathcal{ES}} = \mathcal{X} \cup \mathcal{X}_{\mathcal{S}_{\mathcal{E}}} \cup \mathcal{X}_{\mathcal{S}_{\mathcal{I}}}$, where $\mathcal{X}$ is the set of *object variables* that range over sorts $s_i \in \mathcal{S}$, and where $\mathcal{X}_{\mathcal{S}_{\mathcal{I}}}$ and $\mathcal{X}_{\mathcal{S}_{\mathcal{E}}}$ range respectively over sorts in $\mathcal{S}_{\mathcal{I}}$ and $\mathcal{S}_{\mathcal{E}}$.

Functions: $\mathcal{F}_{\mathcal{ES}}$ is the disjoint union of $\mathcal{F} \cup \mathcal{F}_{\mathcal{I}} \cup \mathcal{F}_{\mathcal{E}} \cup \mathcal{F}_{\mathcal{D}}$ defined as follows.

- $\mathcal{F}_{\mathcal{I}}$ contains the function symbol for application of a strategy to a term of sort $s \in \mathcal{S} \cup \mathcal{S}_{\mathcal{E}}$ which has the following rank:

$$[\_](\_) \ : \ (\langle s \mapsto s \rangle \ s) \ \ \mathbf{Set}(s)$$

  Since the result of rules application is a multiset of terms, we use an abstract data type defining multisets as lists, with concatenation denoted by ., the empty list denoted by $\emptyset$, and an append operation denoted by $\cup$. A membership function is also provided.

  Additional operators are also defined over multisets, such as an application of a function f to multiset arguments $w_1, \ldots, w_n$, or a replacement in a term t of variables x_i by multisets of terms w_i:

$$\begin{aligned} \bowtie f(w_1, \ldots, w_n) &= \{f(e_1, \ldots, e_n) \mid e_i \in w_i,\ i = 1 \ldots n\}, \text{ and} \\ \bowtie t(w_1, \ldots, w_n) &= \{t\{x_i \mapsto e_i\} \mid x_i \in Var(t),\ e_i \in w_i,\ i = 1, \ldots, n\}, \end{aligned}$$

  The operator $[[\_]](\_)$ is the generalisation of $[\_](\_)$ on multisets.
- The set of elementary strategy symbols $\mathcal{F}_{\mathcal{E}}$ consists of the following symbols

defined on sorts $s, s_1, \ldots, s_n \in \mathcal{S} \cup \mathcal{S}_\mathcal{E}$

$$
\begin{array}{lcl}
\mathbf{f} & : & (\langle s_1 \mapsto s_1\rangle \ldots \langle s_n \mapsto s_n\rangle)\ \ \langle s \mapsto s\rangle \\
 & & \text{for any } f : (s_1 \ldots s_n)\ s \in \mathcal{F}, \\
\mathbf{l} & : & (\langle s_1 \mapsto s_1\rangle \ldots \langle s_k \mapsto s_k\rangle \mathbf{Set}(s_{k+1}) \ldots \mathbf{Set}(s_m))\ \ \langle s \mapsto s\rangle \\
 & & \text{for each } ([\ell(\overline{z})]l \to r \ \mathbf{where}\ C) \in \mathcal{R} \\
 & & \text{if } z_i : s_i,\ l : s,\ r : s \\
\mathsf{id},\ \mathsf{fail} & : & \langle s \mapsto s\rangle \\
\mathsf{one},\ \mathsf{all} & : & (\langle s \mapsto s\rangle)\ \ \langle s \mapsto s\rangle \\
\mathsf{repeat*},\ \mathsf{iterate*} & : & (\langle s \mapsto s\rangle)\ \ \langle s \mapsto s\rangle \\
\mathsf{select\_one}, \mathsf{select\_all}, \mathsf{select\_first} & : & (\langle s \mapsto s\rangle, \ldots, \langle s \mapsto s\rangle)\ \ \langle s \mapsto s\rangle \\
\mathsf{dk},\ \mathsf{dc},\ \mathsf{first},\ ; & : & (\langle s \mapsto s\rangle, \ldots, \langle s \mapsto s\rangle)\ \ \langle s \mapsto s\rangle \\
\mathsf{dc\_one},\ \mathsf{first\_one} & : & (\langle s \mapsto s\rangle, \ldots, \langle s \mapsto s\rangle)\ \ \langle s \mapsto s\rangle
\end{array}
$$

- The set of defined strategy symbols $\mathcal{F}_\mathcal{D}$ with signatures:

$$\mathsf{d} : (s_1, \ldots, s_n)\langle s \mapsto s\rangle$$

  where $s_1, \ldots, s_n \in \mathcal{S} \cup \mathcal{S}_\mathcal{E}$ are sorts of the arguments, and $\langle s \mapsto s\rangle$ is the sort of the result.

We call elementary strategy terms the terms of

$$\mathcal{ES} = \mathcal{T}(\mathcal{F} \cup \mathcal{L} \cup \{\ \mathsf{id}, \mathsf{fail}, ;, \mathsf{one}, \mathsf{all}, \mathsf{select\_one}, \mathsf{select\_all}, \mathsf{select\_first}, \mathsf{dk}, \mathsf{dc}, \mathsf{first}, \mathsf{dc\_one}, \mathsf{first\_one}, \mathsf{repeat*}, \mathsf{iterate*}\})$$

Unlabelled rewrite rules: $E_{\mathcal{ES}}$ axiomatise operations on multisets of terms. Rules on multisets of terms are partly given in Figure 3 and in Figure 4. A part of them provide an axiomatisation for the membership relation of a term to the domain $dom(S)$ of a strategy S, that is the set of terms t such that $[S](t)$ is a non-empty multiset.

Additional operators defined over multisets, such as $\bowtie\ f$ and $\bowtie\ t$, and the operator $[[\_]](\_)$ on multisets are formally defined by rewrite rules too.

Rewrite rules: $\mathcal{R}_{\mathcal{ES}}$ is the disjoint union of several subsets:

- Rules defining the congruence property. For any symbol $\mathbf{f}$ such that there exists an $f \in \mathcal{F}_{\mathcal{ES}}$ with rank $f : (s_1 \ldots s_n)\ s$, with $s, s_1, \ldots, s_n \in \mathcal{S}_{\mathcal{ES}}$, a

$$
\begin{array}{lll}
t \in dom(\mathsf{id}) & \rightarrow true & \\
t \in dom(\mathsf{one}(S)) & \rightarrow true & \textbf{if } t \in dom(S) \\
t \in dom(\mathsf{all}(S)) & \rightarrow true & \textbf{if } t \in dom(S) \\
t \in dom(\mathsf{select\_all}(S_1, \ldots, S_n)) & \rightarrow true & \\
\quad \textbf{if } t \in dom(S_1) \ \vee \ \ldots \ \vee \ t \in dom(S_n) & & \\
t \in dom(\mathsf{select\_one}(S_1, \ldots, S_n)) & \rightarrow true & \\
\quad \textbf{if } t \in dom(S_1) \ \vee \ \ldots \ \vee \ t \in dom(S_n) & & \\
t \in dom(\mathsf{select\_first}(S_1, \ldots, S_n)) & \rightarrow true & \\
\quad \textbf{if } t \in dom(S_1) \ \vee \ \ldots \ \vee \ t \in dom(S_n) & & \\
f(t_1, \ldots, t_n) \in dom(\mathbf{f}(S_1, \ldots, S_n)) & \rightarrow true & \\
\quad \textbf{if } \wedge_{i=1,\ldots,n} t_i \in dom(S_i) & & \\
l(z_1, \ldots, z_m) \in dom(\mathbf{l}(S_1, \ldots, S_k, X_{k+1}, \ldots, X_n)) & \rightarrow true & \\
\quad \textbf{if } \wedge_{i=1,\ldots,k} z_i \in dom(S_i) \wedge_{j=k+1,\ldots,m} z_j \in X_j & & \\
t \in dom(S_1; S_2) & \rightarrow true & \\
\quad \textbf{if } t \in dom(S_1) \textbf{ where } res := [S_1](t) \textbf{ if } at\_least\_one(res, S_2) & &
\end{array}
$$

Fig. 3. Unlabelled Rules for *dom*

rewrite rule over the sort $\mathbf{Set}(s)$ is generated:

$$[\mathbf{f}(S_1, \ldots, S_n)](f(x_1, \ldots, x_n)) \rightarrow\!\!\bowtie f([S_1](x_1), \ldots, [S_n](x_n))$$

where variables have types: $x_i : s_i$ and $S_i : \langle s_i \mapsto s_i \rangle$. The label [FSYM] of this rule refers to the set of all rules generated by this schema (see Figure 5).

- Rules defining label application. For any rewrite rule from $\mathcal{R}$ on sort s and of the form:

$$[\ell(\overline{z})] \quad l \rightarrow r \textbf{ where } C$$

where $z_i : s_i \in \overline{z}$, and $l : s$, $r : s$ with $s \in \mathcal{S}_{\mathcal{ES}}$, a rewrite rule over the sort $\mathbf{Set}(s)$ is embedded:

$$
\begin{array}{l}
\mathsf{l}(S_1, \ldots, S_k, X_{k+1}, \ldots, X_m)\ l(z_1, \ldots, z_k) \rightarrow \\
\qquad\qquad \bowtie r([S_1](z_1), \ldots, [S_k](z_k), X_{k+1}, \ldots, X_m) \\
\qquad\qquad \textbf{where } C([S_1](z_1), \ldots, [S_k](z_k), X_{k+1}, \ldots, X_m)
\end{array}
$$

where variables have type $z_i : s_i$ for $i = 1, \ldots, k$, $S_i : \langle s_i \mapsto s_i \rangle$ for $i =$

• at_least_one is defined by:

$$at\_least\_one(e.w, S) \to true \textbf{ if } e \in dom(S)$$
$$at\_least\_one(e.w, S) \to true \textbf{ if } at\_least\_one(w, S)$$

• $\bowtie u'$ is defined for any fixed $u' \in \mathcal{T}(\mathcal{F}_{\mathcal{ES}} \cup \mathcal{X}_{\mathcal{ES}})$ with n variables by $2n+1$ rewriting rules:

$$\begin{array}{ll} For\ i = 1, \ldots, n: & \bowtie u'(w_1, \ldots, e_i.w_i, \ldots, w_n) \to \\ & \bowtie u'(w_1, \ldots, e_i, \ldots, w_n) \cup \bowtie u'(w_1, \ldots, w_i, \ldots, w_n) \\ For\ i = 1, \ldots, n: & \bowtie u'(w_1, \ldots, \emptyset, \ldots, w_n) \to \emptyset \\ & \bowtie u'(e_1, \ldots, e_n) \to u'(e_1, \ldots, e_n) \end{array}$$

• $\bowtie f$ is defined for any $f \in \mathcal{F}_{\mathcal{ES}}$ as follows:

$$\bowtie f(w_1, \ldots, w_n) \to (\bowtie f(x_1, \ldots, x_n))(w_1, \ldots, w_n)$$

• $[[\_]](\_)$ is defined by:

$$\begin{array}{ll} [[S]](e.w) & \to [S](e) \ \cup \ [[S]](w) \\ [[S]](\emptyset) & \to \emptyset \end{array}$$

Fig. 4. Unlabelled Rules (Cont.)

$1, \ldots, k$ and $X_j : \mathbf{Set}(s_j)$ for $j = k+1, \ldots, m$.
Again the label [RLAB] refers to the set of all rules generated by this schema (see Figure 5).

- Rules defining identity, failure and concatenation, i.e. strategy operators id, fail, ; , defined in Figure 5.
- Rules defining the semantics of strategy constructors one, all, select_one, select_all, select_first, id, fail given in Figure 6.
- Rules defining dk, dc, first, dc_one, first_one, repeat*, iterate* with respect to previous ones, given in Figure 7.
- Rules for defined strategies are built from rewrite rules on strategies. For each rule on sort s, either in an implicit form $\mathsf{d}(S_1, \ldots, S_n) \to S$, or in an explicit form $[\mathsf{d}(S_1, \ldots, S_n)](x) \to [S](x)$, where $\mathsf{d} \in \mathcal{F}_{\mathcal{D}}$, S_i for $i = 1, \ldots, n$ and S are strategy terms, $x : s$ is a new variable in $\mathcal{X}$, the following rewrite rule, labelled with [DSTR], is added to the theory:

$$[\mathsf{DSTR}] \quad [\mathsf{d}(S_1, \ldots, S_n)](x) \to [S](x)$$

Other rules on strategy terms are also added with their own labels.

Labels: $\mathcal{L}_{\mathcal{ES}}$ is the disjoint union of all labels defined in the sets of rules $\mathcal{R}_{\mathcal{ES}}$.

Example 5.1 *With the traditional definition of the sort list[s] with two constructors (nil, ·) and a rewrite rule with variables $x_1 : s, x_2 : s$:*

$$
\begin{array}{lll}
[\mathsf{ID}] & [\mathsf{id}(S)](t) & \to \{t\} \\
[\mathsf{FAIL}] & [\mathsf{fail}(S)](t) & \to \emptyset \\
[\mathsf{CONC}] & [(S_1; S_2)](t) & \to [[S_2]](\ [S_1](t)) \\
[\mathsf{FSYM}]_1 & [\mathbf{f}(S_1, \ldots, S_n)](f(x_1, \ldots, x_n)) \to \bowtie f([S_1](x_1), \ldots, [S_n](x_n)) & \\
[\mathsf{FSYM}]_2 & [\mathbf{f}(S_1, \ldots, S_n)](g(x_1, \ldots, x_m)) \to \emptyset \ \textbf{if}\ f \neq g & \\
[\mathsf{RLAB}]_1 & [\mathbf{l}(S_1, \ldots, S_k, X_{k+1}, \ldots, X_m)](l(z_1, \ldots, z_k)) \to & \\
 & \bowtie r([S_1](z_1), \ldots, [S_k](z_k), X_{k+1}, \ldots, X_m) & \\
 & \textbf{where}\ C([S_1](z_1), \ldots, [S_k](z_k), X_{k+1}, \ldots, X_m) & \\
 & \text{for each } ([\ell(\overline{z})]l \to r\ \textbf{where}\ C) \in \mathcal{R} & \\
[\mathsf{RLAB}]_2 & [\mathbf{l}(S_1, \ldots, S_k, X_{k+1}, \ldots, X_m)](t) & \to \emptyset \\
 & \textbf{if}\ l\ \textit{does not match}\ t & \\
 & \text{for each } ([\ell(\overline{z})]l \to r\ \textbf{where}\ C) \in \mathcal{R} &
\end{array}
$$

Fig. 5. Labelled rules for elementary strategies

$[\ell(x_1, x_2)] \quad x_1 + 0 \to x_1 + x_2 \quad \textbf{\textit{where}}\ x_2 := f(x_1)$,
the previous construction generates the following strategy symbols:

$$
\begin{array}{lll}
\mathsf{nil} & : & \langle list[s] \mapsto list[s] \rangle \\
\_ \cdot \_ : & & (\langle s \mapsto s \rangle\ \langle list[s] \mapsto list[s] \rangle) \quad \langle list[s] \mapsto list[s] \rangle \\
\mathbf{l}(\_, \_) : & & (\langle s \mapsto s \rangle\ \mathbf{Set}(s))\ \ \langle s \mapsto s \rangle
\end{array}
$$

Assuming that the signature of the symbol $+$ *is* $(s\ s)\ s$*, one* [RLAB] *rewrite rule is generated for the rewrite rule* ℓ

$$[\mathbf{l}(S_1, X_2)](x_1 + 0) \to \bowtie +([S_1](x_1), X_2) \quad \textbf{\textit{where}}\ X_2 :=\bowtie f([S_1](x_1))$$

where $S_1 : \langle s \mapsto s \rangle$, $X_2 : \mathbf{Set}(s)$, $x_1 : s$.

Example 5.2 *The definition of the strategy* map *is for instance given by the definition of the strategy symbol*

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

and the rule

$$\mathsf{map}(S) \to \mathit{dc}(\mathsf{nil}, S \cdot \mathsf{map}(S))$$

$$
\begin{array}{lll}
[\mathsf{SA}] & [\mathsf{select\_all}(S_1,\ldots,S_n)](t) & \rightarrow [S_1](t) \cup \ldots \cup [S_n](t) \\
[\mathsf{SO}]_1 & [\mathsf{select\_one}(S_1,\ldots,S_n)](t) & \rightarrow res_1 \\
 & & \textbf{if } t \in dom(S_1) \\
 & & \textbf{where } res_1 := [S_1](t) \\
[\mathsf{SO}]_n & [\mathsf{select\_one}(S_1,\ldots,S_n)](t) & \rightarrow res_n \\
 & & \textbf{if } t \in dom(S_n) \\
 & & \textbf{where } res_n := [S_n](t) \\
[\mathsf{SO}]_0 & [\mathsf{select\_one}(S_1,\ldots,S_n)](t) & \rightarrow \emptyset \\
 & & \textbf{if } t \notin dom(S_1) \wedge \ldots \wedge t \notin dom(S_n) \\
[\mathsf{SF}]_1 & [\mathsf{select\_first}(S_1,\ldots,S_n)](t) & \rightarrow res_1 \\
 & & \textbf{if } t \in dom(S_1) \\
 & & \textbf{where } res_1 := [S_1](t) \\
[\mathsf{SF}]_n & [\mathsf{select\_first}(S_1,\ldots,S_n)](t) & \rightarrow res_n \\
 & & \textbf{if } t \notin dom(S_1) \wedge \ldots \wedge \\
 & & t \notin dom(S_{n-1}) \wedge t \in dom(S_n) \\
 & & \textbf{where } res_n := [S_n](t) \\
[\mathsf{SF}]_0 & [\mathsf{select\_first}(S_1,\ldots,S_n)](t) & \rightarrow \emptyset \\
 & & \textbf{if } t \notin dom(S_1) \wedge \ldots \wedge t \notin dom(S_n) \\
[\mathsf{ONE}]_1 & [\mathsf{one}(S)](t) & \rightarrow \{r\} \\
 & & \textbf{if } t \in dom(S) \wedge r \in res \\
 & & \textbf{where } res := [S](t) \\
[\mathsf{ONE}]_2 & [\mathsf{one}(S)](t) & \rightarrow \emptyset \\
 & & \textbf{if } t \notin dom(S) \\
[\mathsf{ALL}] & [\mathsf{all}(S)](t) & \rightarrow [S](t)
\end{array}
$$

Fig. 6. Labelled rules for elementary strategies (Cont.)

where $S : \langle s \mapsto s \rangle$ *and* $s \in \mathcal{S}$. *The following rule over the sort* $\mathbf{Set}(list[s])$ *with the label* [DSTR] *is added:*

$$[\mathsf{map}(S)](x) \rightarrow [dc(\mathsf{nil}, S \cdot \mathsf{map}(S))](x)$$

[DK]	$\mathsf{dk}(S_1, \ldots, S_n)$	$\rightarrow \mathsf{select\_all}(\mathsf{all}(S_1), \ldots, \mathsf{all}(S_n))$
[DC]	$\mathsf{dc}(S_1, \ldots, S_n)$	$\rightarrow \mathsf{select\_one}(\mathsf{all}(S_1), \ldots, \mathsf{all}(S_n))$
[FIRST]	$\mathsf{first}(S_1, \ldots, S_n)$	$\rightarrow \mathsf{select\_first}(\mathsf{all}(S_1), \ldots, \mathsf{all}(S_n))$
[DCONE]	$\mathsf{dc\_one}(S_1, \ldots, S_n)$	$\rightarrow \mathsf{select\_one}(\mathsf{one}(S_1), \ldots, \mathsf{one}(S_n))$
[FIRSTONE]	$\mathsf{first\_one}(S_1, \ldots, S_n)$	$\rightarrow \mathsf{select\_first}(\mathsf{one}(S_1), \ldots, \mathsf{one}(S_n))$
[REP]	$\mathsf{repeat*}(S)$	$\rightarrow \mathsf{first}(S; \mathsf{repeat*}(S), id)$
[ITE]	$\mathsf{iterate*}(S)$	$\rightarrow \mathsf{dk}(S; \mathsf{iterate*}(S), id)$

Fig. 7. Definition of other elementary strategies

where $x : list[s]$.

We have now reached the point where general rules calling strategies in matching conditions are rewrite rules in the rewrite theory of strategies $\mathcal{RT}_{\mathcal{ES}}$. Looking now at the whole set of rewrite rules with matching conditions constructed in $\mathcal{R}_{\mathcal{ES}}$, we can state the following properties.

Theorem 5.1 *The rewrite theory of strategies* $\mathcal{RT}_{\mathcal{ES}}$ *has the following properties:*

- $E_{\mathcal{ES}}$ *is confluent and terminating.*
- $\mathcal{R}_{\mathcal{ES}}$ *is not terminating in general. If* S *is a deterministic strategy, i.e.* $card(dom(S)) = 1$, $\mathsf{repeat*}(S)$ *and* $\mathsf{iterate*}(S)$ *do not terminate.*
- *The application function* $[\_](\_)$ *is completely defined over* $\mathcal{ES}$.
- $\mathcal{R}_{\mathcal{ES}}$ *is not confluent. However using ordered rewriting makes this problem disappear.*

Proof: The proofs are just sketched here. More details can be found in [Bor98].

- $E_{\mathcal{ES}}$ is confluent and terminating because rules in this system define functions by structural induction on their arguments.
- $\mathcal{R}_{\mathcal{ES}}$ is not terminating, due in particular to rules for $\mathsf{repeat*}$ and $\mathsf{iterate*}$. According to their definition, the evaluation of $[\mathsf{repeat*}(S)](t)$ stops only if their exists a step n where $([S](t))^n$ fails. So if S is a deterministic strategy that never fails and always returns one result, i.e. $card(dom(S)) = 1$, $\mathsf{repeat*}(S)$ and $\mathsf{iterate*}(S)$ do not terminate.
- The fact that the application function $[\_](\_)$ is completely defined over $\mathcal{ES}$ is easy to prove by structural induction on elementary strategy terms.
- $\mathcal{RES}$ is not confluent, due to the rules $[\mathsf{SO}]_1, \ldots, [\mathsf{SO}]_n, [\mathsf{SO}]_0$ that may apply concurrently on a same expression. However using a strategy that applies these rules in the given order makes this problem disappear. Note that in this case, there is no more difference between $\mathsf{select\_one}$ and

select_first, which actually corresponds to the current implementation in ELAN of the strategy language.
□

6 Reflective definition of rewriting with matching conditions

In Section 5, we have described the strategy evaluation mechanism using rewrite rules with matching conditions. In this section, we take advantage of the reflexive power of rewriting logic and we describe the mechanism of rewriting with matching conditions in the formalism presented in Section 5 and using rewrite rules with matching conditions and strategies.

To encode rewriting, it is needed to describe all steps of matching, substituting and replacement performed in an elementary rewrite step and how to iterate them. The problem has been already addressed in higher-order logic in [Pau83], and in the rewriting setting by [Vit94,Cla98] for the case of non-conditional rewrite rules. Here we deal with rewrite rules with matching conditions. However we do not fully axiomatise matching, substituting and replacement, that have been addressed in previous works [Vit94,Cla98,BKK98,BKKR99]. We rather focus on the construction of proof terms associated to the rewriting mechanism to explain their structuration presented in Section 4.

The rewriting mechanism is described with rewrite rules, applying on tuples $\pi : (t, \omega, \ell, \sigma, \mu)$ where π is a proof-term, t is the term to be rewritten, ω a position in t (ϵ is used when no position is specified), ℓ a rewrite rule, denoted by its label, in a set $\mathcal{R}$ of rewrite rules, σ and μ are substitutions.

Deduction of a sequent in a rewrite theory, $\mathcal{RT} \vdash \pi \ : \ \langle t \rangle \rightarrow \langle t' \rangle$, corresponds to rewriting a term $\mathcal{RT} \bullet \pi : t$ built with a ternary operator $\_ \bullet \_ : \_$ taking as arguments a rewrite theory $\mathcal{RT}$, a proof term π and a term. We introduce a **one_step** rewrite rule, applying on such triple:

$$\begin{array}{lll} \textbf{[one\_step]}\ \mathcal{RT} \bullet \pi : t & \rightarrow & \mathcal{RT} \bullet (\pi ; \pi') : t' \\ \qquad\qquad \textbf{where} & & \ell \in \mathcal{R} \text{ and } \pi', t' \text{ are such that} \\ & & \pi : (t, \epsilon, \ell, Id) \rightarrow \pi ; \pi' : (t', \epsilon, \ell, Id) \end{array}$$

This rule chooses a candidate rule ℓ in $\mathcal{R}$, performs an elementary step to rewrite the term t into t' and produces the associated proof term π.

We have now to explain how to build the derivation

$$\pi : (t, \epsilon, \ell, Id) \rightarrow \pi ; \pi' : (t', \epsilon, \ell, Id).$$

Let ℓ be the candidate rule to rewrite the term t. Its left and right-hand sides are denoted respectively $lhs(\ell)$ and $rhs(\ell)$. There are three phases in the elementary rewrite step: find a position ω in t and a substitution σ, such that $lhs(\ell)\sigma = t_{|\omega}$, then eliminate conditional parts by normalising conditions, comparing the results to the patterns p_i, and recording them in the substitution σ, and finally apply the substitution and the replacement to obtain the resulting term $t[rhs(\ell)\sigma]_\omega$. Before applying the replacement, elimination of conditional parts are iterated until no more matching condition is left. We can express this algorithm with three rewrite rules **match**, **where_elim** and **replace** detailed below and a simple strategy:

TopRewrite $\rightarrow$ dc(**match**) ; repeat*(dc(**where_elim**)) ; dc(**replace**).

The associated proof term is built during the computation. This provides a description of the proof close to the actual execution in ELAN, since the proof term corresponds exactly now to the computation trace provided by the system.

Lists of proof terms are built from the empty list "$\Box_{pt}$" and the concatenation operator "$\_\cdot\_$". The notation $proof(t \rightarrow t')$ is used to compute the proof term in the sequent $\pi \ : \ \langle t \rangle \rightarrow \langle t' \rangle$. The partial function nf applied to a term t computes a normal form of t with respect to the whole set of rewrite rules R. Below we describe the rules that allow computing an elementary rewrite step.

$$\begin{array}{lll} [\textbf{match}] & \pi \ : \ (t, \epsilon, \ell, Id) \quad \rightarrow \quad \pi; (\ell(\sigma)\{\Box_{pt}\}) \ : \ (t, \omega, \ell, \sigma) \\ & \textbf{where} \qquad \omega, \sigma \text{ such that } lhs(\ell)\sigma = t_{|\omega} \end{array}$$

Starting with a term t and a rewrite rule ℓ, the **match** phase records a position ω in t and a substitution σ such that $lhs(\ell)\sigma = t_{|\omega}$.

How to build by rewriting the matching substitution is described in [Vit94,Cla98,BKK98,BKKR99]. Note however that in presence of AC theories, the encoding is more complex and we do not describe it here.

The rule first builds the proof term $\ell(\sigma)$ corresponding to the application of the rewrite rule ℓ and instantiation of variables occurring in its left-hand side by values given by σ. $\ell(\sigma)\{\Box_{pt}\}$ is the proof term under construction. The list of subproofs issued from matching conditions in this matching phase is initialised by the empty list denoted by $\Box_{pt}$.

$$\begin{array}{ll} [\textbf{where\_elim}] \ \pi; (\pi'\{\Lambda\}) \ : \ (t, \omega, \ell \ where \ p := c, \sigma) \quad \rightarrow \\ \qquad \pi; (\pi'\{\Lambda \cdot proof(c\sigma \rightarrow c')\}) : (t, \omega, \ell, \sigma\{p \mapsto c'\}) \\ \qquad \textbf{where } c' = nf(c\sigma) \qquad \textbf{if } p \mapsto c' \text{ exists} \end{array}$$

This rule eliminates *one* matching condition. The rewrite rule ℓ is transformed by removing the matching condition $p := c$. The substitution σ records the matching related to the matching condition. This will be useful to instantiate the right-hand side of ℓ in the last **replace** phase. To achieve the construction of the proof term associated to the application of ℓ, the proof term associated to the sequent $c\sigma \to c'$ (denoted by $proof(c\sigma \to c')$) is recorded in the list of subproof terms Λ. The rule applies if there exists a match from p to c', which is expressed by the condition $p \mapsto c'$ exists.

$$\begin{aligned}[\textbf{replace}]\ \pi; (\pi'\{\Lambda\})\ :\ (t, \omega, r, \sigma) \quad &\to \\ \pi; t[\pi'\{\Lambda\}]_\omega\ :\ &(t[rhs(\ell)\sigma]_\omega, \epsilon, \ell, Id)\end{aligned}$$

At this stage of the calculus, the substitution σ is able to instantiate each variable of the right-hand side of ℓ by a ground term. The replacement is performed and produces the result term, as well as the resulting proof term. Several applications of **Congruence** are simulated by building the proof term $t[\pi'\{\Lambda\}]_\omega$ describing the *replacement* under the appropriate context t.

Soundness and completeness of this encoding is expressed in the two next propositions:

Proposition 6.1 *For any application of strategy* **TopRewrite**, *there exists a finite number of applications of rules* **Reflexivity**, **Congruence**, **Replacement** *and* **Transitivity** *building the same sequent and associated proof term.*

Proof: Let us prove Proposition 6.1 by induction on the depth n of the proof term resulting from the application of the strategy **TopRewrite**. The *depth* of a proof term is defined by:

- $\text{depth}(\pi_1; \pi_2) = \max(\text{depth}(\pi_1), \text{depth}(\pi_2))$
- $\text{depth}(t) = 0$
- $\text{depth}(f(\pi_1, \ldots, \pi_n)) = \max_{i=1,\ldots,n}(\text{depth}(\pi_i))$
- $\text{depth}(\ell(\overline{\alpha})\{\beta_1 \ldots \beta_k.\gamma_1 \ldots \gamma_j\}) =$
  $1 + \max(\max_{i=1,\ldots,k}(\text{depth}(\beta_i)), \max_{i=1,\ldots,j}(\text{depth}(\gamma_i)))$

▶ **case** $n = 1$**:** Then there is no condition or local affectation in r, $(j + k = 0)$. Applying rules **match** and **replace** on the tuple: $t : (t, \epsilon, u(\overline{x}) \to u'(\overline{x}), Id, Id)$ yields:

$$t : (t, \epsilon, u(\overline{x}) \to u'(\overline{x}), Id, Id).$$

$$\xrightarrow{\textbf{match}} t; \ell(\sigma)\{\Box_{pt}\} : (t, \omega, u(\overline{x}) \to u'(\overline{x}), \sigma, \sigma)$$

$$\xrightarrow{\textbf{replace}} t[\ell(\sigma)\{\Box_{pt}\}]_\omega : (t[u'(\overline{x})\sigma]_\omega, \epsilon, u(\overline{x}) \to u'(\overline{x}), Id, Id)$$

Ground terms appearing in σ and their associated proofs terms $\overline{\alpha}$ are build by **Reflexivity**; an application of **Replacement** builds the se-

quent: $\ell(\overline{\alpha}) \ : \ t_{|\omega} \rightarrow u'(\{\overline{w} \mapsto \overline{x}\})$. Several applications of **Congruence** transform the sequent $\ell(\overline{\alpha}) \ : \ t_{|\omega} \rightarrow u'(\{\overline{w} \mapsto \overline{x}\})$ into $t[\ell(\overline{\alpha})]_\omega \ : \ t \rightarrow t[u'(\{\overline{w} \mapsto \overline{x}\})]_\omega$. The composition of **match** and **replace** corresponds to an application of the **Transitivity** deduction rule.

▶ **case** $n+1$**:** Then there are n embedded applications of one-step rewriting in the evaluation of conditions and local affectations in r. Let us assume that we apply the rules **match**, m times ($m = j + k \geq 1$) **where_elim** or **if_elim** and then **replace** on the tuple: $t : (t, \epsilon, r, Id, Id)$ and get the result:

$$t; t[\ell(\sigma)\{\lambda\}]_\omega : (t[u'(\{\overline{a} \cup \overline{w} \mapsto \overline{x} \cup \overline{y}\})]_\omega, \epsilon, u(\overline{x}) \rightarrow u'(\overline{x} \cup \overline{y}), Id, Id)$$

where $\lambda = \beta_1 \ldots \beta_k.\gamma_1 \ldots \gamma_j$ is a list of proof terms β_i and γ_i corresponding to evaluation of conditions and local affectations in r and of depth at most n.

By induction hypothesis applied to the proof terms in λ and the associated reductions, there is a finite number of applications of rules of rewriting logic that build these proof terms and sequents. An application of **Replacement** followed by several applications of **Congruence** then build the resulting sequent and proof term.

□

Proposition 6.2 *For any application of rules* **Reflexivity**, **Congruence**, **Replacement** *and* **Transitivity**, *there exists a finite number of applications of the strategy* **TopRewrite**, *building the same sequent and an* equivalent *associated proof term.*

Proof: Let us prove by structural induction on proof terms the expected result. Suppose that sequents involved in premises of rules **Reflexivity**, **Transitivity**, **Congruence** and **Replacement** can be obtained in an *equivalent* form by applying **TopRewrite**. We want to prove that for each sequent produced by one application of those deduction rules, there exists a given sequence of applications of **TopRewrite** that produce an equivalent sequent.

▶ **Reflexivity:** This is trivially obtained by 0 application of the strategy **TopRewrite**.

▶ **Transitivity:** Let $\pi_1 \ : \ t_1 \rightarrow t_2$ and $\pi_2 \ : \ t_2 \rightarrow t_3$ be two sequents. By hypothesis, there exist a finite number of applications of **TopRewrite** that transforms the term t_1 (resp. t_2) into t_2 (resp. t_3) and produces the proof term π_1' resp. π_2' equivalent to π_1 resp. π_2. Successive applications of those two sequences of **TopRewrite** produce the proof term: $\pi_1'; \pi_2'$ which is equivalent to $\pi_1; \pi_2$.

▶ **Congruence:** Given $\pi_i \ : \ t_i \rightarrow t_i'$, $i = 1 \ldots n$. An application of **Congruence** produces the sequent

$$f(\pi_1, \ldots, \pi_n) \ : \ f(t_1, \ldots, t_n) \rightarrow f(t_1', \ldots, t_n')$$

By induction hypothesis, equivalent sequents $\pi'_i \ : \ t_i \to t'_i, \ i = 1 \ldots n$ can be produced by **TopRewrite**.

Starting from the term $f(t_1, \ldots, t_n)$, successive applications of **TopRewrite** produce:

$$f(\pi'_1, t_2 \ldots, t_n); f(t'_1, \pi'_2, t_3 \ldots, t_n); \ldots; \ f(t'_1, \ldots, t'_{n-1}, \pi'_n) \to f(t_1, \ldots, t_n) \ : \ f(t'_1, \ldots, t'_n)$$

which is a proof term equivalent to $f(\pi'_1, \ldots, \pi'_n)$ so also to $f(\pi_1, \ldots, \pi_n)$. This equivalence uses repeatedly axioms of Independence and Local Identities.

▶ **Replacement:** Given $\alpha_i, i = 1 \ldots n$, $\beta_i, i = 1 \ldots k$ and $\gamma_i, i = 1 \ldots j$. By induction hypothesis, the list λ of subproof terms is equivalent to the list $\overline{\beta}^k.\overline{\gamma}^j$. An application of the (**match**, ..., **replace**) sequence at the root position ϵ produces the proof term $t[\ell(\sigma)\{\lambda\}]_\epsilon = \ell(\sigma)\{\lambda\}$ which is equivalent to $\ell(\overline{w}^n)\{\overline{\beta}^k.\overline{\gamma}^j\}$. Successive applications of **TopRewrite** will produce:

$$\ell(\overline{w}^n)\{\lambda\}; u'(\alpha'_1, w_2, \ldots, w_n); u'(w'_1, \alpha'_2, w_3, \ldots, w_n); \ldots u'(w'_1, \ldots, w'_{n-1}, \alpha'_n)$$

which is a proof term equivalent to $\ell(w_1, \ldots, w_n)\{\lambda\}; u'(\alpha'_1, \ldots, \alpha'_n)$ (thanks to Independence and Local Identities of Figure 2).

Applying the axiom Parallel Move Lemma of Figure 2, this proof term is equivalent to $\ell(\alpha'_1, \ldots, \alpha'_n)\{\lambda\}$ where the $\alpha'_i, i = 1 \ldots n$ are equivalent to $\alpha_i, i = 1 \ldots n$. The resulting proof term is equivalent to $\ell(\overline{\alpha}^n)\{\overline{\beta}^k.\overline{\gamma}^j\}$.

□

The following example illustrates why equivalent proof terms have sometimes to be considered.

Example 6.1 *Let us consider different proofs of* $g(f(a)) \to f(b)$ *using the following rules:* $\ell_1 \ : \ a \to b$, *and* $\ell_2 \ : \ g(x) \to x$. *With rules* **Reflexivity**, **Congruence**, **Replacement** *and* **Transitivity**, *applied in different orders, one can derive three sequents with different proof terms:*

$$\begin{aligned} \ell_2(f(r_1)) \ &: \ g(f(a)) \to f(b) \\ \ell_2(f(a)); f(\ell_1) \ &: \ g(f(a)) \to f(b) \\ g(f(\ell_1)); \ell_2(f(b)) \ &: \ g(f(a)) \to f(b) \end{aligned}$$

But only the two last ones can be derived using the strategy dc(**match**) ; repeat*(dc(**where_elim**)) ; dc(**replace**). *Indeed this is not a problem, since according to the* **ParallelMoveLemma** *on proof terms, the first one is equivalent to the two others.*

The rewrite rules **match**, **where_elim** and **replace**, as well as the strategy **TopRewrite** can easily be implemented in ELAN, leading to a definition of the ELAN interpreter in ELAN.

7 Implementation in ELAN

The evaluation mechanism of the ELAN language relies on the concepts described in Sections 4 and 5. There are very few points that deviate from what has been presented, and that relate to implementation choices.

First, we distinguish two classes of strategies in the language, but this distinction is historical and only relies at the implementation level, in the way their evaluation is implemented. More precisely, application of an elementary strategy S on a term t, denoted $(S)t$, is performed by a C function generated by ELAN. On the contrary, when S is a defined strategy, the application of S on a term t, denoted $[S]t$, is performed by an ELAN program, called the strategy meta-interpreter.

Rules that define the result of the application of a defined strategy to a term and involve implicitly or explicitly the application operator are used by the strategy meta-interpreter, and applied with a strategy `eval` defined in the meta-interpreter. They are labelled by a special label `[.]`. Indeed the meta-interpreter, designed as an ELAN program, can be modified or enriched by another evaluation strategy.

In ELAN, once strategy operators are defined, strategy terms are available and may be rewritten too. Rules that define a computation on strategy terms are evaluated exactly like rules on (ordinary) terms:

- If the rule is unlabelled, the leftmost innermost predefined strategy of the interpreter is applied.
- When the rule is labelled by $[\ell]$, it is used by the ELAN interpreter as any other labelled rule governed by a strategy. So the user has to provide a strategy involving ℓ.

More details on the language are available in the ELAN manual [BCD+98].

8 Applications

Many computational processes in automated deduction can be expressed as instances of a general schema that consists of applying transformation rules

on formulas with some strategy, until reaching specific normal forms. Such processes are naturally expressed in ELAN and several applications have been designed.

Programming: One of the first application was to prototype the fundamental mechanisms of logic and functional programming languages like first-order resolution and λ-calculus. The general framework of Constraint Logic Programming [JM94] can be easily designed in the ELAN framework [KR98], since its operational semantics is clearly formalised as rewrite rules, although the application strategy is often defined in an informal way. Some implementations [Bor95] related to a calculus of explicit substitutions (the first-order rewrite system $\lambda\sigma$ that mimics λ-calculus) open the way of implementing higher-order logic programming languages via a first-order setting. Another calculus of explicit substitutions based on the π-calculus is used to provide a formal specification of Input/Output for ELAN [Vir96].

Proving: ELAN was used to implement a predicate prover based on the rules proposed by J.-R. Abrial, and implemented in the B-tools [Abr96]. We developed also a propositional sequent calculus, completion procedures for rewrite systems [KM95], and sufficient conditions for the termination problem. In particular criteria for termination have been designed thanks to automata techniques [Gen98]. In addition, a library for automata construction and manipulation has been designed. Approximation automata are used to check conditions for reachability, sufficient completeness, absence of conflicts in systems described by non-conditional rewrite rules [Gen98].

Solving: The notion of rewriting controlled by strategies is used in [KR98] to describe in a unified way the constraint solving mechanism as well as the meta-language needed to manipulate the constraints. This provides programs that are very close to the proof theoretical setting used now to describe constraint manipulations like unification or numerical constraint solving. ELAN offers a constraint programming environment where the formal description of a constraint solver is directly executable. ELAN has been tested on several examples of constraint solvers for various computation domains and combinations like abstract domains [KR98,Rin97] (term algebras) and more concrete ones (booleans, integers, reals). In [Cas96,Cas98], it is shown how to use computational systems as a general framework for handling Constraint Satisfaction Problems (CSP for short). The approach leads to the design in ELAN of COLETTE, a solver for constraints over integers and finite domains. A generalisation of the ELAN strategy language is proposed in [BC98] for programming the cooperation of constraint solvers.

9 Conclusion

Another way to give semantics to ELAN is presented in [BKK98] using a functional approach, and is extended in [CK99] using a rewriting calculus, called ρ-calculus. This is a simple and powerful calculus that allows dealing with explicit rule application, explicit handling of result sets, and is paramaterised by a matching theory T. The ρ-calculus allows a uniform combination of first-order and higher-order computations, and application of strategies (high-order objects) to terms is a special instance of this calculus. This more general setting provides capability to extend the framework of the ELAN system. Other possible extensions are to consider rewrite rules with constraints, and to provide in the language structured objects, in the line of object-oriented languages.

Acknowledgement: The Protheo team is currently composed by Peter Borovanský, Horatiu Cirstea, Hubert Dubois, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen, Marian Vittek. We thank them all for their interaction in the design and the implementation of ELAN. We also thank José Meseguer for our long collaboration on many topics concerning the theoretical background as well as the implementation of ELAN.

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[Art97] T. Arts. *Automatically proving termination and innermost normalisation of term rewriting systems.* PhD thesis, Universiteit Utrecht, Utrecht, 1997.

[BB97] H. Barendregt and E. Barendsen. Autartik computations in formal proofs. `ftp://ftp.cs.kun.nl/pub/CompMath.Found/computations.ps.Z`, 1997.

[BC98] 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 Second International Workshop on Rewriting Logic and its Applications, RWLW'98*, volume 15, pages 379–398, Pont-à-Mousson, France, September 1998. Electronic Notes in Theoretical Computer Science.

[BCD+98] P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. ELAN *V 3.3 User Manual.* LORIA, Nancy (France), third edition, December 1998.

[BCR93] L. Bachmair, T. Chen, and I. V. Ramakrishnan. Associative-commutative discrimination nets. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT '93: Theory and Practice of Software Development, 4th International Joint Conference CAAP/FASE*, LNCS 668, pages 61–74, Orsay, France, April 13–17, 1993. Springer-Verlag.

[BGLS95] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation. *Information and Computation*, 121(2):172–192, 1995.

[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.

[BKK98] P. Borovanský, C. Kirchner, and H. Kirchner. A functional view of rewriting and strategies for a semantics of ELAN. In M. Sato and Y. Toyama, editors, *The Third Fuji International Symposium on Functional and Logic Programming*, pages 143–167, Kyoto, April 1998. World Scientific. Also report LORIA 98-R-165.

[BKKR99] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in elan: a functional semantics. *International Journal of Foundations of Computer Science*, 1999.

[BKN87] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3(1 & 2):203–216, April 1987.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.

[Bor95] P. Borovanský. Implementation of higher-order unification based on calculus of explicit substitutions. In M. Bartošek, J. Staudek, and J. Wiedermann, editors, *Proceedings of the SOFSEM'95: Theory and Practice of Informatics*, volume 1012 of *Lecture Notes in Computer Science*, pages 363–368. Springer-Verlag, 1995.

[Bor98] 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, October 1998. also TR CRIN 98-T-326.

[Cas96] C. Castro. Solving Binary CSP using Computational Systems. 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.

[Cas98] C. Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, 34:263–293, September 1998.

[CK98] H. Cirstea and C. Kirchner. Using rewriting and strategies for describing the B predicate prover. In *Proceedings of the Workshop on Strategies in Automated Deduction, CADE-15*, Lindau, Germany, July 1998.

[CK99] H. Cirstea and C. Kirchner. Combining higher-order and first-order computation using ρ-calculus: Towards a semantics of ELAN. In D. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems*. Research Studies Press (Wiley), 1999. Presented at the Frocos'98 workshop, Amsterdam.

[Cla98] M. Clavel. *Reflection in general logics, rewriting logic, and Maude*. PhD thesis, University of Navarre, Spain, 1998.

[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.

[DHK98] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. Rapport de Recherche 3400, Institut National de Recherche en Informatique et en Automatique, April 1998. `ftp://ftp.inria.fr/INRIA/publication/RR/RR-3400.ps.gz`.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990.

[Dom92] E. Domenjoud. A technical note on AC-unification. the number of minimal unifiers of the equation $\alpha x_1 + \cdots + \alpha x_p \doteq_{AC} \beta y_1 + \cdots + \beta y_q$. *Journal of Automated Reasoning*, 8:39–44, 1992. Also as research report CRIN 89-R-2.

[Dow99] G. Dowek. *La part du Calcul*. PhD thesis, Mémoire d'habilitation de l'Université de Paris 7, 1999.

[Eke95] S. Eker. Associative-commutative matching via bipartite graph matching. *Computer Journal*, 38(5):381–399, 1995.

[Gen98] T. Genet. *Contraintes d'ordre et automates d'arbres pour les preuves de terminaison*. PhD thesis, Universié Henri Poincaré - Nancy I, 1998.

[Gra96] B. Gramlich. On proving termination by innermost termination. 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 93–107. Springer-Verlag, July 1996.

[Hul80] J.-M. Hullot. *Compilation de Formes Canoniques dans les Théories équationelles*. Thèse de Doctorat de Troisième Cycle, Université de Paris Sud, Orsay (France), 1980.

[JK86] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986. Preliminary version in Proceedings 11th ACM Symposium on Principles of Programming Languages, Salt Lake City (USA), 1984.

[JK91] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.

[JM94] J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19-20:503–582, 1994.

[Kir95] H. Kirchner. Some extensions of rewriting. In H. Comon and J.-P. Jouannaud, editors, *Term Rewriting*, volume 909 of *Lecture Notes in Computer Science*, pages 54–73. Springer-Verlag, 1995.

[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.

[KL91] E. Kounalis and D. Lugiez. Compilation of pattern matching with associative commutative functions. In *16th Colloquium on Trees in Algebra and Programming*, volume 493 of *Lecture Notes in Computer Science*, pages 57–73. Springer-Verlag, 1991.

[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.

[KM96] H. Kirchner and P.-E. Moreau. A reflective extension of Elan. 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.

[KR98] C. Kirchner and C. Ringeissen. Rule-Based Constraint Programming. *Fundamenta Informaticae*, 34(3):225–262, September 1998.

[LM94] D. Lugiez and J.-L. Moysset. Tree automata help one to solve equational formulae in AC-theories. *Journal of Symbolic Computation*, 18(4):297–318, 1994.

[Mar94] C. Marché. Normalised rewriting and normalised completion. In S. Abramsky, editor, *Proceedings 9th IEEE Symposium on Logic in Computer Science, Paris (France)*, pages 394–403, 1994.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[MJ95] E. Meijer and J. Johan. Merging Monads and Folds for Functional Programming. In E. Meijer and J. Johan, editors, *Proceedings of Advanced Fonctional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 228–266. Springer-Verlag, 1995.

[MOM93] N. Martí-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.

[NN93] P. Nivela and R. Nieuwenhuis. Saturation of first-order (constrained) clauses with the *saturate* system. In C. Kirchner, editor, *Proceedings 5th Conference on Rewriting Techniques and Applications, Montreal (Canada)*, volume 690 of *Lecture Notes in Computer Science*, pages 436–440. Springer-Verlag, 1993.

[Pau83] L. C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[PS81] G. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28:233–264, 1981.

[Rin97] C. Ringeissen. Prototyping Combination of Unification Algorithms with the ELAN Rule-Based Programming Language. In *Proceedings 8th Conference on Rewriting Techniques and Applications, Sitges (Spain)*, volume 1232 of *Lecture Notes in Computer Science*, pages 323–326. Springer-Verlag, 1997.

[RV95] M. Rusinowitch and L. Vigneron. Automated Deduction with Associative-Commutative Operators. *Applicable Algebra in Engineering, Communication and Computation*, 6(1):23–56, January 1995.

[Vig94] L. Vigneron. *Déduction automatique avec contraintes symboliques dans les théories équationnelles.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, November 1994.

[Vir96] P. Viry. Input/Output for ELAN. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in TCS*, Asilomar (California), September 1996.

[Vit94] 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.