To appear in: International Journal of Foundations of Computer Science
© World Scientific Publishing Company

REWRITING WITH STRATEGIES IN ELAN: A FUNCTIONAL SEMANTICS[*]

PETER BOROVANSKÝ
Comenius University, Bratislava, Slovakia
`Peter.Borovansky@loria.fr`

and

CLAUDE KIRCHNER and HÉLÈNE KIRCHNER and CHRISTOPHE RINGEISSEN
`Claude.Kirchner,Helene.Kirchner,Christophe.Ringeissen@loria.fr`
LORIA: CNRS & INRIA
615, rue du Jardin Botanique
54600 Villers-lès-Nancy, France

Received (January 1999)
Revised (May 1999)
Communicated by M. Sato & Y. Toyama

ABSTRACT

In this work, we consider term rewriting from a functional point of view. A rewrite rule is a function that can be applied to a term using an explicit application function. From this starting point, we show how to build more elaborated functions, describing first rewrite derivations, then sets of derivations. These functions, that we call strategies, can themselves be defined by rewrite rules and the construction can be iterated leading to higher-order strategies. Furthermore, the application function is itself defined using rewriting in the same spirit. We present this calculus and study its properties. Its implementation in the ELAN language is used to motivate and exemplify the whole approach. The expressiveness of ELAN is illustrated by examples of polymorphic functions and strategies.

Keywords: Rewriting Calculus, Rewriting Logic, Strategy, Rewrite Based Language, Term Rewriting, Strategy, Matching.

1. Introduction

Rule-based reasoning is present in many domains of computer science. In algebraic specification methods, rewriting is used for prototyping specifications and computing the specified functions; in theorem proving, for dealing with equality, simplifying the formulas and pruning the search space; in programming languages, the rewriting concept can be explicit like in OBJ or ML, or hidden in the operational semantics; expert systems and knowledge representation systems also use rewriting

[*]This is a revised version of [6].

to describe actions to perform; in constraint solving, rewrite rules have been first used in unification theory [22] to express transformation of equational systems or constraint systems [10], and are now becoming available at the programming language level in systems like Eclipse, via constraint handling rules [17], Claire [9] and OZ [37]. In the more general setting of logical frameworks, rules are also used for describing inference systems and logic of computations as in LCF [18], Isabelle [34, 35] or HOL [19].

In most cases, a rewrite rule is applied to a term but this application is not an object accessible to the user. As a consequence, the use of term rewriting is always submitted to hidden strategies and the straightforward standard question is: "what kind of strategy is your system using? left-most inner-most? outside-in? needed? random? lazy?..."

The importance of strategies as such has been recognized since a long time. For example in LCF, *"The discovery of the operators THEN, ORELSE, and REPEAT, for combining tactics, was a breakthrough in the development of Edinburgh LCF"* [33]. In the context of rewriting, user-defined strategies have been first introduced in the ELAN language [23]. Now, in this work, we consider a rewrite rule as a function called a primal strategy, and the application of a rule as an explicit user-accessible function. We thus give to the user the possibility to express when and where a rule must be applied. Of course this opens many new possibilities, but also raises several problems addressed in this paper.

After Section 2 where the main basic notations used in this work are summarized, we give a formal definition of strategies and the semantics of strategy application to a term. We introduce strategies step by step, distinguishing between primal strategies (basically rewrite rules and two constants expressing identity and failure, composed by concatenation and congruence) in Section 3, elementary strategies (with choice operators to deal with sets of possible results) in Section 4, and finally defined strategies (with user-defined operators and rewrite rules) in Section 5. Then we explain in Section 6 how this can be iterated and used to define functions of higher level, again by rewriting. Such functions are also called strategies, since they express the (possibly very complex) way in which basic rewrite rules are applied to a term. ELAN is a language that makes these concepts available and in which first class objects are terms, rewrite rules and strategies. We give in Section 7 and Section 8 several examples to illustrate the language, with a strong emphasis on its functional features. The formalism developed here gives to the basic constructions of ELAN a formal functional semantics, which is our primary goal. But it also opens new insights and questions, sketched in the conclusion.

2. Notations

We first restrict ourselves in Sections 3, 4, 5 to unsorted non-conditional rules. We then consider in Section 6 the many-sorted case. The order-sorted and conditional cases can be handled in a similar, although more technical, manner, not presented here. Our definitions are consistent with [16, 22, 2] to which the reader is referred for detailed considerations on universal algebra and term rewriting systems.

We consider a set $\mathcal{F}$ of ranked function symbols, a set $\mathcal{X}$ of variables and the set of first-order terms $\mathcal{T}(\mathcal{F},\mathcal{X})$ built from $\mathcal{F}$ and $\mathcal{X}$. The set of variables of a term t is denoted by $\mathcal{V}ar(t)$. $\mathcal{T}(\mathcal{F})$ is the set of ground terms, i.e. terms without variables. Substitutions are term endomorphisms written $\sigma = \{(y \mapsto u)\ldots\}$, when the variable y is substituted by the term u. $\sigma(t)$ or σt is the result of applying σ to the term t. To simplify notation, we sometimes denote a sequence of objects $(a_1,\ldots,a_n)$ by $\overline{a}$. A rewrite rule is an object $l \to r$, where l and r are terms in $\mathcal{T}(\mathcal{F},\mathcal{X})$ respectively called left and right-hand side and where $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. A set of rewrite rules is called a rewrite system or a rewrite program. The set of variables $\mathcal{V}ar(l \to r)$ is $\mathcal{V}ar(l)$.

In Sections 3, 4, 5, we define the syntax and the semantics of our strategy language and specify how each construction is applied to a term. This amounts to define a strategy application operator $@$, where $@$ is a place-holder for '@'rguments. This operator takes as arguments a strategy ζ and a term t, and produces the term $[\zeta](t)$. The evaluation of this operator provides the definition of the interpreter of our strategy language, which is described via labeled rewrite rules of the form ($\mathsf{Label}\quad L \mapsto\!\!\!\twoheadrightarrow R$) to distinguish them, at least in a first step, from the rewrite rules given by the user.

3. Primal strategies

In this section, we build in a progressive way the most elementary notion of strategy, called primal strategy, and we make precise the associated semantics.

3.1. Applying a rewrite rule

Given a rewrite rule $l \to r$, we consider it first as an untyped function that can be applied to a term t, leading to the object $[l \to r](t)$. This function returns the empty set when l does not match t. Otherwise, it evaluates to the result of the standard rewriting operation at the top position of t, which consists in replacing t by r whose variables have been substituted by their corresponding instances coming from the matching of l to t.

The function on terms associated to a rewrite rule is called a *primal strategy*. The semantics of application of a rewrite rule to a term is defined by the rules given in Figure 1. Their intuitive meaning is the search for a match between l and t; this is done by decomposing the matching problem thanks to the Decompose rule, and instantiating the variables in r when they are associated to a subterm in t; this is performed by the Instantiate rule. The Clash and Var-Clash rules detect matching failure, and the Success rule returns the result when the match is found. All these rules involve the syntactic construction of tuples of terms separated by commas, and denoted by $(t_1,\ldots,t_n)$ or simply $t_1,\ldots,t_n$. The strategy application operator $@$ takes as first argument a syntactic object of the form $(t_1,\ldots,t_n) \to r$ where $(t_1,\ldots,t_n)$ is a tuple of terms and r is a term. In the following, as in Figure 1, parentheses are often omitted and we write $[t_1,\ldots,t_n \to r](u_1,\ldots,u_n)$ or $[l \to r](t)$, instead of $[(t_1,\ldots,t_n) \to r]((u_1,\ldots,u_n))$ or $[(l) \to r]((t))$.

Decompose $[v_1, \ldots, v_p, f(\bar{u}), v_{p+2}, \ldots, v_n \to r](w_1, \ldots, w_p, f(\bar{t}), w_{p+2}, \ldots, w_n)$
$\mapsto\!\!\!\!\twoheadrightarrow$
$[v_1, \ldots, v_p, \bar{u}, v_{p+2}, \ldots, v_n \to r](w_1, \ldots, w_p, \bar{t}, w_{p+2}, \ldots, w_n)$
if $f(\bar{u}) \neq f(\bar{t})$

Clash $[v_1, \ldots, v_p, f(\bar{u}), v_{p+2}, \ldots, v_n \to r](w_1, \ldots, w_p, g(\bar{t}), w_{p+2}, \ldots, w_n)$
$\mapsto\!\!\!\!\twoheadrightarrow$
$\emptyset$
if $f \neq g$

Instantiate $[v_1, \ldots, v_p, y, v_{p+2}, \ldots, v_n \to r](w_1, \ldots, w_p, u, w_{p+2}, \ldots, w_n)$
$\mapsto\!\!\!\!\twoheadrightarrow$
$[\sigma v_1, \ldots, \sigma v_p, u, \sigma v_{p+2}, \ldots, \sigma v_n \to \sigma r](w_1, \ldots, w_p, u, w_{p+2}, \ldots, w_n)$
if $y \in \mathcal{X} \backslash \mathcal{V}ar((w_1, \ldots, w_p, u, w_{p+2}, \ldots, w_n))$
where $\sigma = \{(y \mapsto u)\}$

Var-Clash $[v_1, \ldots, v_p, y, v_{p+2}, \ldots, v_n \to r](w_1, \ldots, w_p, u, w_{p+2}, \ldots, w_n)$
$\mapsto\!\!\!\!\twoheadrightarrow$
$\emptyset$
if $y \in \mathcal{V}ar((w_1, \ldots, w_p, u, w_{p+2}, \ldots, w_n)) \wedge y \neq u$

Success $[w_1, \ldots, w_n \to r](w_1, \ldots, w_n)$
$\mapsto\!\!\!\!\twoheadrightarrow$
$\{r\}$

Figure 1: PSA: Primal Strategy Application.

The next example shows how the PSA calculus can be used to compute the term obtained by rewriting t at the top with a rule $l \to r$. Later on, we will formally prove that this term is the normal form w.r.t. PSA of $[l \to r](t)$. The PSA calculus closely corresponds to the implementation of a rewriting step into an interpreter or a compiler, where the substitution of the matching problem is not explicitly built, but where variables are instead progressively instantiated.

Example 3.1

$$\begin{array}{ll}
\multicolumn{2}{l}{[f(x, h(x)) \to h(x)](f(h(x'), h(h(x'))))} \\
\mapsto\!\!\!\!\twoheadrightarrow_{\mathsf{Decompose}} & [x, h(x) \ \to h(x)](h(x'), h(h(x'))) \\
\mapsto\!\!\!\!\twoheadrightarrow_{\mathsf{Instantiate}} & [h(x'), h(h(x')) \ \to h(h(x'))](h(x'), h(h(x'))) \\
\mapsto\!\!\!\!\twoheadrightarrow_{\mathsf{Success}} & \{h(h(x'))\}
\end{array}$$

With the same rewrite rule, a simple case of failure is

$$\begin{array}{lll}
[f(x, h(x)) \ \to h(x)](f(x', h(y'))) & \mapsto\!\!\!\!\twoheadrightarrow_{\mathsf{Decompose}} & [x, h(x) \ \to h(x)](x', h(y')) \\
& \mapsto\!\!\!\!\twoheadrightarrow_{\mathsf{Instantiate}} & [x', h(x') \ \to h(x')](x', h(y')) \\
& \mapsto\!\!\!\!\twoheadrightarrow_{\mathsf{Decompose}} & [x', x' \ \to h(x')](x', y') \\
& \mapsto\!\!\!\!\twoheadrightarrow_{\mathsf{Var-Clash}} & \emptyset
\end{array}$$

In this case, Var-Clash forbids the instantiations of variables occurring in $f(x', h(y'))$. These variables must be considered as skolemized.

In the previous example, the rewrite rule $f(x, h(x)) \rightarrow h(x)$ is not left-linear. In case where all rewrite rules $l \rightarrow r$ are left-linear, the Instantiate rule can be specialized as follows:

$$\begin{array}{ll} \textsf{Instantiate-LL} & [v_1, \ldots, v_p, y, v_{p+2}, \ldots, v_n \rightarrow r](w_1, \ldots, w_p, u, w_{p+2}, \ldots, w_n) \\ & \mapsto\!\!\!\twoheadrightarrow \\ & [v_1, \ldots, v_p, v_{p+2}, \ldots, v_n \rightarrow \{y \mapsto u\}r](w_1, \ldots, w_p, w_{p+2}, \ldots, w_n) \end{array}$$

which is fully adapted to the left linearity hypothesis and allows furthermore to get the intuition of the calculus more easily.

3.2. Properties of the PSA *calculus*

Proposition 3.1 The PSA calculus for primal strategy application is terminating.

Proof. The complexity measure is obtained by a lexicographic combination of two complexity measures (N_1, N_2) defined as follows:

1. $N_1([v_1, \ldots, v_n \rightarrow r](w_1, \ldots, w_n)) = \sharp(\mathcal{V}ar((v_1, \ldots, v_n)) \backslash \mathcal{V}ar((w_1, \ldots, w_n)))$, where $\mathcal{V}ar((t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} \mathcal{V}ar(t_i)$, and $N_1(\{r\}) = N_1(\emptyset) = -1$.
2. $N_2([v_1, \ldots, v_n \rightarrow r](w_1, \ldots, w_n)) = \sum_{i=1}^{n} |v_i|$, where $|t|$ denotes the size of t, i.e. the number of function symbols occurring in t, and $N_2(\{r\}) = N_2(\emptyset) = -1$.

It is easy to check that:

- Decompose preserves N_1 and strictly decreases N_2,
- Instantiate strictly decreases N_1,
- Clash, Var-Clash and Success strictly decrease N_1.

□

Proposition 3.2 The PSA calculus for primal strategy application is locally confluent.

Proof. By the classical critical pair lemma [2], we have just to check that all critical pairs are confluent. Because of the form of the rules and since there is no recursive strategy call, we have only to check that there is no non-confluent top critical pair.

Decompose does not superpose with Clash since $f \neq g$, and due to its conditional application, the rule Decompose does not applies if Success applies.

We should notice that in the Instantiate and Var-Clash rules, the variable y is assumed to be of sort variable, thus it cannot be an object of the form $f(\bar{u})$. Consequently there is no overlap between Decompose and Instantiate or Var-Clash.

The same argument allows us to state that there is no overlap between Clash and Instantiate or Var-Clash. Since $f \neq g$ Clash and Success can not overlap.

There is no overlap between Instantiate and Var-Clash because their respective conditions are mutually exclusive.

Finally because of the syntactic limitation put on the variable y in the rules Instantiate and Var-Clash, there is no overlap between these rules and Success.

This shows that there is no overlap between the rules of PSA. □

Corollary 1 The PSA calculus for primal strategy application is terminating and Church–Rosser.

Proposition 3.3 The normal form of a term $[v_1, \dots, v_n \to r](w_1, \dots, w_n)$ with respect to PSA is either a singleton or the empty set.

Proof. Ad absurdum. If no rule among Decompose, Clash, Instantiate, Var-Clash applies anymore on $[v_1, \dots, v_n \to r](w_1, \dots, w_n)$, then $(v_1, \dots, v_n) = (w_1, \dots, w_n)$ and Success applies. □

In order to formalize a correctness result for PSA calculus, we need to introduce a notion of invariant on expressions reduced in this calculus. Let us define RWT as follows:

$$\begin{aligned} RWT([v_1, \dots, v_n \to r](w_1, \dots, w_n)) = \ & \{\sigma r \mid \textstyle\bigwedge_{i=1}^{n} \sigma v_i = w_i \text{ and } \\ & \quad \mathcal{D}om(\sigma) \cap \mathcal{V}ar((w_1, \dots, w_n)) = \emptyset\} \\ RWT(\{r\}) = \ & \{r\} \\ RWT(\emptyset) = \ & \emptyset \end{aligned}$$

Lemma 1 For any rule r in PSA and any expressions e, e', if $e \mapsto\!\!\!\!\twoheadrightarrow e'$, then

$$RWT(e) = RWT(e').$$

Proof. Let us check this property for each rule in PSA:

- Decompose: $\sigma f(u_1, \dots, u_n) = f(t_1, \dots, t_n)$ implies $\sigma u_1 = t_1, \dots, \sigma u_n = t_n$.
- Clash: there is no substitution σ such that $\sigma f(\bar{u}) = g(\bar{t})$ if $f \neq g$.
- Var-Clash: if $y \in \mathcal{V}ar((w_1, \dots, w_n))$, there is no substitution σ such that $\sigma y = u$ and $\mathcal{D}om(\sigma) \cap \mathcal{V}ar((w_1, \dots, w_n)) = \emptyset$, provided that u is not identical to y.
- Instantiate: if $y \notin \mathcal{V}ar((w_1, \dots, w_n))$, $\sigma y = u$ and $\mathcal{D}om(\sigma) \cap \mathcal{V}ar((w_1, \dots, w_n)) = \emptyset$ implies that the image of y by σ is necessarily u.
- Success: $\sigma w_1 = w_1, \dots, \sigma w_n = w_n$ implies that σ is the identity substitution satisfying $\mathcal{D}om(\sigma) \cap \mathcal{V}ar((w_1, \dots, w_n)) = \emptyset$ since $\mathcal{D}om(\sigma) = \emptyset$.

□

Corollary 2 (PSA correctness) For a rewrite rule $l \to r$ and a term t such that $\mathcal{V}ar(l \to r) \cap \mathcal{V}ar(t) = \emptyset$, we have $[l \to r](t) \downarrow_{\mathsf{PSA}} = \{s\}$ if and only if t is rewritten on top into s with the rewrite rule $l \to r$.

Proof. This follows from Lemma 1 and from standard definition of rewriting: $RWT([l \to r](t)) = s$, where s is obtained by rewriting t at the top with the rule $l \to r$. □

When $[l \to r](t) \downarrow_{\mathsf{PSA}} = \emptyset$, we say that the rule $l \to r$ does not apply to t and that the rule application fails. Otherwise, if $s \in [l \to r](t) \downarrow_{\mathsf{PSA}}$, we call s a result of the rule application to t, which is said successful. Note that currently, the normal form can only be a singleton or the empty set.

3.3. Comments on the PSA *calculus*

In contrast to the standard notion of rewriting, the result of a rule application is a set of terms. This is useful for several purposes.

To avoid partial functions. It is convenient to deal with functions computing a possibly empty set of results. Otherwise, the rule application should be considered as a partial function.

To extend to rewriting modulo. The framework extends smoothly to rewriting modulo equational theories, in which case the result of the application of a rule may be not unique. For example, if the operator g is assumed to be commutative, then Decompose and Success are replaced by the following rules:

$$\begin{array}{ll} \mathsf{DecomposeC} & [\ldots, g(u_1,u_2), \ldots \to r](\ldots, g(t_1,t_2), \ldots) \\ & \mapsto\!\!\!\twoheadrightarrow \\ & [\ldots, u_1, u_2, \ldots \to r](\ldots, t_1, t_2, \ldots) \\ & \cup \\ & [\ldots, u_1, u_2, \ldots \to r](\ldots, t_2, t_1, \ldots) \\ & \text{if } g(u_1,u_2) \neq_C g(t_1,t_2) \\ \\ \mathsf{SuccessC} & [v_1, \ldots, v_n \to r](w_1, \ldots, w_n) \\ & \mapsto\!\!\!\twoheadrightarrow \\ & \{r\} \\ & \text{if } v_1 =_C w_1, \ldots, v_n =_C w_n \end{array}$$

where $=_C$ denotes the equality modulo the commutativity of g, and $\cup$ is the union operator. In order to take into account the union operator $\cup$, considered here without any equational property, a few rules are added:

$$\begin{array}{llll} \mathsf{Merge} & \{r_1, \ldots, r_n\} \cup \{r_{n+1}, \ldots, r_m\} & \mapsto\!\!\!\twoheadrightarrow & \{r_1, \ldots, r_n, r_{n+1}, \ldots, r_m\} \\ \mathsf{IdR} & \{r_1, \ldots, r_n\} \cup \emptyset & \mapsto\!\!\!\twoheadrightarrow & \{r_1, \ldots, r_n\} \\ \mathsf{IdL} & \emptyset \cup \{r_1, \ldots, r_n\} & \mapsto\!\!\!\twoheadrightarrow & \{r_1, \ldots, r_n\} \\ \mathsf{IdLR} & \emptyset \cup \emptyset & \mapsto\!\!\!\twoheadrightarrow & \emptyset \end{array}$$

The bottom-up application of the previous rules leads to an expression that encodes a set of results, not necessarily reduced to one element.

Example 3.2 Let g be a commutative operator, and a, b two constants.

$$\begin{array}{ll} \multicolumn{2}{l}{[g(x,y) \to x](g(a,b))} \\ \mapsto\!\!\!\twoheadrightarrow_{\mathsf{DecomposeC}} & [x, y \ \to x](a,b) \cup [x, y \ \to x](b,a) \\ \mapsto\!\!\!\twoheadrightarrow^*_{\mathsf{Instantiate}} & [a, b \ \to a](a,b) \cup [b, a \ \to b](b,a) \\ \mapsto\!\!\!\twoheadrightarrow^*_{\mathsf{Success}} & \{a\} \cup \{b\} \\ \mapsto\!\!\!\twoheadrightarrow_{\mathsf{Merge}} & \{a, b\} \end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{[g(x,a) \to x](g(a,b))} \\
\quad \mapsto\!\!\!\!\rightarrow_{\mathsf{DecomposeC}} & [x, a \ \to x](a,b) \cup [x, a \ \to x](b,a) \\
\quad \mapsto\!\!\!\!\rightarrow_{\mathsf{Clash}} & \emptyset \cup [x, a \ \to b](b,a) \\
\quad \mapsto\!\!\!\!\rightarrow_{\mathsf{Instantiate}} & \emptyset \cup [b, a \ \to b](b,a) \\
\quad \mapsto\!\!\!\!\rightarrow_{\mathsf{SuccessC}} & \emptyset \cup \{b\} \\
\quad \mapsto\!\!\!\!\rightarrow_{\mathsf{IdL}} & \{b\}
\end{array}$$

More sophisticated PSA-like calculi should be designed to implement rewriting modulo other equational theories, like for instance modulo the associativity-commutativity axioms, or more generally modulo syntactic equational theories, where decomposition rules can be used to simplify unification and matching problems [24, 21, 26].

To deal with more elaborated strategies. After this first step in the definition of strategies, more elaborate ones will produce sets containing in general more than a single element. Later on, we will use the same syntax for strategy application on sets of terms. Then, the result of such an application will be obtained as a union of results.

Instead of choosing sets as the basic structure for representing the result of strategies application, other possible choices can be lists or multisets. Similar calculi could be developed in these cases, some of them sticking closely to implementation considerations. We have chosen here to use the set representation because of its simplicity.

3.4. Building more primal strategies

Now, to apply consecutively two rules, the corresponding primal strategies are composed using the concatenation operator ";" that applies as follows:

$$\mathsf{Compose} \quad [l_1 \to r_1 \ ; \ l_2 \to r_2](t) \quad \mapsto\!\!\!\!\rightarrow \quad [l_2 \to r_2]([l_1 \to r_1](t))$$

and more generally, for two strategies ζ_1 and ζ_2, we have

$$\mathsf{Compose} \quad [\zeta_1 \ ; \ \zeta_2](t) \quad \mapsto\!\!\!\!\rightarrow \quad [\zeta_2]([\zeta_1](t)).$$

The composition of two primal strategies is again called a primal strategy. Note that this strategy corresponds to the THEN tactical in LCF.

It is natural to add as primal strategies the identity (associated to the rewrite rule $x \to x$) and the failure function, denoted respectively by **id** and **fail**. They apply to terms in the expected way:

$$\begin{array}{lll}
\mathsf{Identity} & [\mathbf{id}](t) & \mapsto\!\!\!\!\rightarrow \quad \{t\} \\
\mathsf{Fail} & [\mathbf{fail}](t) & \mapsto\!\!\!\!\rightarrow \quad \emptyset.
\end{array}$$

The next step consists of applying a rule deeper into a term. For this purpose, function operators (i.e. elements of $\mathcal{F}$) can take as arguments primal strategies ζ and yield again a primal strategy, that applies to terms as follows:

Congruence	$[f(\zeta_1,\ldots,\zeta_n)](f(t_1,\ldots,t_n))$	$\mapsto\!\!\!\!\twoheadrightarrow$	$f([\zeta_1](t_1),\ldots,[\zeta_n](t_n))$
Congruence-Fail	$[f(\zeta_1,\ldots,\zeta_n)](g(t_1,\ldots,t_n))$	$\mapsto\!\!\!\!\twoheadrightarrow$	$\emptyset$.

In this definition, the interpretation of f is strongly overloaded: on the left-hand side of the Congruence rule, the first f operates on strategies and the second on terms; on the right-hand side, f operates on sets, where it is defined as usual by $f(A_1,\ldots,A_n) = \{f(a_1,\ldots,a_n)|a_i \in A_i\}$. When f is a nullary symbol, these definitions simplify in an straightforward way.

To summarize, primal strategies are built from rewrite rules and the two constants **fail** and **id**, using concatenation and congruence w.r.t. the function operators. The application of a primal strategy to a term is defined by the rules of PSA, and the set of rules BP composed of Compose, Identity, Fail, Congruence and Congruence-Fail.

Proposition 3.4 For a rewrite system $\mathcal{R}$, the application of a primal strategy to a term t using PSA and BP always results in a set of terms (which, except for rewriting modulo an equational theory, is either empty or a singleton). When u belongs to the set of results, there exists a sequence of rewriting steps from t to u using $\mathcal{R}$, and conversely.

Proof. This follows from Corollary 2 and from the definition of the concatenation, congruence, identity and failure strategies. □

In order to simplify the notations, and to iterate the construction of functions, a name is given to rewrite rules. So the basic objects are now *labeled rewrite rules* $[\ell]\ l \to r$, where the label ℓ belongs to a set $\mathcal{L}$ of ranked label symbols, and whose left and right-hand sides are terms of $\mathcal{T}(\mathcal{F},\mathcal{X})$. When the rule has n distinct variables, then the label ℓ is assumed to be of arity n and the rule is denoted $[\ell(\bar{x})]\ l(\bar{x}) \to r(\bar{x})$. When variables are irrelevant, we just omit them from the notation.

Example 3.3 Given the rewrite system $\{[\ell_1(x)]\ f(x) \to x, [\ell_2]\ g(a) \to a\}$, the strategy $g(f(\ell_1)); g(\ell_1); \ell_2$ is the function that, when applied to the term $g(f(f(a)))$, yields $\{a\}$ and the empty set when applied to any other term.

We can notice that in rewriting logic [30], proofs are first-order objects and are represented by proof terms. Proof terms are terms built from elements of $\mathcal{T}(\mathcal{F},\mathcal{X})$, combined with function symbols in $\mathcal{F}$, label symbols in $\mathcal{L}$ and the concatenation operator ";". So a proof term denotes a primal strategy in our approach.

4. Elementary strategies

How to handle sets of results and to encode search are the main questions we are facing now. As the last example illustrates, the functions called *primal strategies*, are just application of rules at some fixed position. Now we need to express the search for specific reductions. For instance, we would like to express the leftmost innermost rewriting strategy as a function that, given a term rewrite system R and a term t, returns, when it terminates, the set of all its leftmost innermost normal forms.

For this purpose, we enrich the strategy language by operators allowing the description of the computation space in a convenient way.

First, we introduce two selectors, specifying in which way the set of results of a strategy is used: one may be interested in the whole set or just in one of its elements. For that purpose, we introduce now **one** and **all**, two elementary *unary* operators on strategies, whose semantics is defined by the following rules:

$$\begin{array}{lllll} \mathsf{One} & [\mathbf{one}(\zeta)](t) & \mapsto\!\!\!\rightarrow & \{t'\} \\ & & & \text{if } t' \in [\zeta](t) \\ \mathsf{All} & [\mathbf{all}(\zeta)](t) & \mapsto\!\!\!\rightarrow & [\zeta](t). \end{array}$$

The result selectors can be made more precise when a different representation for results is chosen. For example, when using lists, the **one** operator could be completed with selectors for the "nth" result.

Then we also introduce selection operators **select-one**, **select-first**, **select-all** on strategy tuples. Their semantics is defined by the rules:

$$\begin{array}{llll} \mathsf{Select\text{-}first} & [\mathbf{select\text{-}first}(\zeta_1,\ldots,\zeta_n)](t) & \mapsto\!\!\!\rightarrow & [\zeta_j](t) \\ & & & \text{if } \bigcup_{i=1}^{j-1}[\zeta_i](t) = \emptyset \text{ and } [\zeta_j](t) \neq \emptyset \\ \mathsf{Select\text{-}one} & [\mathbf{select\text{-}one}(\zeta_1,\ldots,\zeta_n)](t) & \mapsto\!\!\!\rightarrow & [\zeta_i](t) \\ & & & \text{if } [\zeta_i](t) \neq \emptyset \\ \mathsf{Select\text{-}all} & [\mathbf{select\text{-}all}(\zeta_1,\ldots,\zeta_n)](t) & \mapsto\!\!\!\rightarrow & \bigcup_{i=1}^{n}[\zeta_i](t) \end{array}$$

This allows us to introduce operators like **dk** ("dont know"), **first** and **dc** ("dont care"). **first** is indeed the same operator as the "ORELSE" used in LCF [18].

The semantics of these operators is defined w.r.t. the previously introduced selectors, using rewriting on strategy terms (notice the arrow change) as follows:

$$\mathsf{Dk} \quad \mathbf{dk}(\zeta_1,\ldots,\zeta_n) \;\rightarrowtail\; \mathbf{select\text{-}all}(\mathbf{all}(\zeta_1),\ldots,\mathbf{all}(\zeta_n)).$$

where $\zeta_1,\ldots\zeta_n$ are any strategies.

This rule has the same meaning, for all term t, as the following rule:

$$\mathsf{Dk} \quad [\mathbf{dk}(\zeta_1,\ldots,\zeta_n)](t) \;\mapsto\!\!\!\rightarrow\; [\mathbf{select\text{-}all}(\mathbf{all}(\zeta_1),\ldots,\mathbf{all}(\zeta_n))](t).$$

For example, $[\mathbf{dk}(a \to b, a \to c)](a) \mapsto\!\!\!\rightarrow^* \{b,c\}$.

The dual of the **dk** strategy is the "dont care" one, **dc**, defined by:

$$\mathsf{Dc} \quad \mathbf{dc}(\zeta_1,\ldots,\zeta_n) \;\rightarrowtail\; \mathbf{select\text{-}one}(\mathbf{all}(\zeta_1),\ldots,\mathbf{all}(\zeta_n)).$$

This last operator, in its version that simply returns the *first* non-empty set of results given by its arguments, is called **first** and is defined by:

$$\mathsf{First} \quad \mathbf{first}(\zeta_1,\ldots,\zeta_n) \;\rightarrowtail\; \mathbf{select\text{-}first}(\mathbf{all}(\zeta_1),\ldots,\mathbf{all}(\zeta_n)).$$

For example, $[\mathbf{first}(a \to b, a \to c)](a) \mapsto\!\!\!\rightarrow^* \{b\}$ and $[\mathbf{first}(a \to c, a \to b)](a) \mapsto\!\!\!\rightarrow^* \{c\}$, while $[\mathbf{first}(a \to c, a \to b)](b) \mapsto\!\!\!\rightarrow^* \emptyset$.

We can also define the **dc-one** strategy which returns only singletons:

$$\textsf{Dc-one} \quad \mathbf{dc\text{-}one}(\zeta_1, \ldots, \zeta_n) \;\;\rightarrowtail\;\; \mathbf{select\text{-}one}(\mathbf{one}(\zeta_1), \ldots, \mathbf{one}(\zeta_n)).$$

Of course, **first-one** can be defined in the same way:

$$\textsf{First-one} \quad \mathbf{first\text{-}one}(\zeta_1, \ldots, \zeta_n) \;\;\rightarrowtail\;\; \mathbf{select\text{-}first}(\mathbf{one}(\zeta_1), \ldots, \mathbf{one}(\zeta_n)).$$

The elementary strategies, and in particular the operators **dk** and **first**, in conjunction with primal strategies, allow us to express, in a very convenient way, search functions which are particularly useful in mechanized theorem proving.

To summarize, elementary strategies are built using two kinds of selectors. **one** and **all** select one or all elements in the set of results of a strategy, while **select-one**, **select-all** and **select-first** select one, all or the first element in a sequence of strategies. The next table presents all elementary strategies previously introduced. In addition, an elementary strategy **dk-one**, that chooses one result in each set of results provided by a sequence of strategies, can also be defined. This strategy may be useful in the context of concurrent rewriting and strategies, as explored in [4].

	result selector	
strategy selector	**one**	**all**
select-one	**dc-one**	**dc**
select-first	**first-one**	**first**
select-all	**dk-one**	**dk**

To conclude this section on elementary strategies, let us mention that constant strategies can also be introduced. For all terms t and s, these constant strategies denoted $\tilde{s}$ are defined by

$$\textsf{Constant} \quad [\tilde{s}](t) \;\;\mapsto\!\!\!\rightarrow\;\; \{s\}.$$

In the following, we denote by ES the set of rules performing elementary evaluation strategies, i.e., the rules One, All, Select-first, Select-one, Select-all, Dk, Dc, First, Dc-one and First-one.

5. Defined strategies

We have seen how the user can define primal strategies and specify some search strategies using elementary strategies. The goal is now to give the possibility to define more complex strategies from elementary ones. This is done as previously, by defining new strategy operators and rewrite rules to specify the strategy semantics.

For example, the function **map** can be defined by a strategy rewrite rule in the following way:

$$\mathbf{map}(\zeta) \rightarrowtail \mathbf{first}(nil, cons(\zeta, \mathbf{map}(\zeta))) \qquad (1)$$

The right-hand side of this definition means that whenever the strategy **map** with an argument ζ (i.e. $\mathbf{map}(\zeta)$) is applied to a list t, either t should be nil, or the strategy ζ is applied to the head of t (in which case t should be a non-empty list) and $\mathbf{map}(\zeta)$ is further applied to the tail of t.

This strategy definition differs from the traditional functional definition of the **map** functor. To get a more standard definition, we can also formulate the definition of **map** using the strategy application symbol @ (and the infix notation "." for the cons operator):

$$\begin{array}{lcl} [\mathbf{map}(\zeta)](nil) & \mapsto\!\!\!\twoheadrightarrow & nil \\ [\mathbf{map}(\zeta)](a.l) & \mapsto\!\!\!\twoheadrightarrow & [\zeta](a).[\mathbf{map}(\zeta)](l) \end{array} \qquad (2)$$

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

In rule (1), $\mathbf{map}(\zeta)$ is a strategy which is defined recursively. When applied without special control, it may lead to infinite computations. This can of course be controlled using the concept of meta-strategies, i.e. strategies controlling the execution of defined strategies. Indeed other rules involving **map** can be written, to express in particular some equivalence on defined strategies, for instance the distributivity of **map** on the concatenation ";"

$$\begin{array}{lcl} \mathbf{map}(\zeta_1) \ ; \ \mathbf{map}(\zeta_2) & \rightarrowtail & \mathbf{map}(\zeta_1 \ ; \ \zeta_2) \\ \mathbf{map}(\mathbf{id}) & \rightarrowtail & \mathbf{id} \end{array}$$

Other examples of rules for elementary strategies are:

$$\begin{array}{lclclcl} \mathbf{id} \ ; \ \zeta & \rightarrowtail & \zeta & \qquad & \zeta \ ; \ \mathbf{id} & \rightarrowtail & \zeta \\ \mathbf{first}(\zeta, \ \zeta) & \rightarrowtail & \zeta & & \mathbf{dk}(\zeta, \ \zeta) & \rightarrowtail & \zeta \end{array}$$

Given a set of strategy symbols $\mathcal{F}_\mathcal{D}$, called defined strategy symbols, rewrite rules on defined strategies are of the form $[\ell] \ \zeta_1 \rightarrowtail \zeta_2$ where ℓ is a label, ζ_1 and ζ_2 are strategy terms built from all previously introduced strategy symbols and $\mathcal{F}_\mathcal{D}$. The defined strategy application is then expressed by rules of the form

$$\mathsf{Dstr} \quad [\zeta_1](t) \ \mapsto\!\!\!\twoheadrightarrow \ [\zeta_2](t)$$

and the labels Dstr can be used to control the application of defined strategies.

Example 5.1 Several examples of basic strategy definitions are the following ones:

$$\begin{array}{lcl} \mathbf{iterate}(\zeta_1) & \rightarrowtail & \mathbf{dk}(\zeta_1; \mathbf{iterate}(\zeta_1), \ \mathbf{id}) \\ \mathbf{repeat}(\zeta_1) & \rightarrowtail & \mathbf{first}(\zeta_1; \mathbf{repeat}(\zeta_1), \ \mathbf{id}) \\ \mathbf{map}2(nil) & \rightarrowtail & nil \\ \mathbf{map}2(\zeta_1.\zeta_2) & \rightarrowtail & \zeta_1.\mathbf{map}2(\zeta_2) \end{array}$$

The **iterate** strategy differs from **repeat** by returning all intermediate forms encountered during the evaluation of the term by the strategy ζ_1, while **repeat** returns only the last one. The strategy **map**2 is driven by a list of strategies which are respectively applied to elements of a list of the same length.

With a rewrite rule: $[\ell] \quad x + 0 \to x$ and all the strategies defined above, we get the following application examples:

$$\begin{array}{lcl} [\mathbf{iterate}(\ell)]((a+0)+0) & \mapsto\!\!\!\twoheadrightarrow^* & \{(a+0)+0, a+0, a\} \\ [\mathbf{repeat}(\ell)]((a+0)+0) & \mapsto\!\!\!\twoheadrightarrow^* & \{a\} \\ [\mathbf{map}2(\mathbf{iterate}(\ell).\mathbf{repeat}(\ell).nil)](a+0.b+0.nil) & \mapsto\!\!\!\twoheadrightarrow^* & \{a+0.b.nil, a.b.nil\} \end{array}$$

Strategy rewrite rules of the form $[\ell]\ \zeta_1 \rightarrowtail \zeta_2$ are called *implicit* since they do not involve explicitly the application operator $@$. However, as illustrated by the **map** example, it is sometimes useful to express a strategy rule depending on the argument on which it is applied. Such rules are also allowed and are called *explicit* strategy rules. They are of the form:

$$\mathsf{Dstr} \quad [\zeta_1](t) \quad \mapsto\!\!\!\twoheadrightarrow \quad R$$

where ζ_1 is a strategy term as before, t is a term and R is build on terms, strategy symbols and the strategy application operator. From the evaluation point of view, these rules are just added to the strategy interpreter.

Example 5.2 To illustrate the definition of explicit strategy rules, let us consider for instance an if-then-else strategy constructor. **if** ζ_1 **then** ζ_2 **else** ζ_3 applied to a term t, returns $[\zeta_1;\zeta_2](t)$, if $[\zeta_1](t)$ is not empty (does not fail), otherwise, it returns $[\zeta_3](t)$. The semantics is defined using an auxiliary symbol **ite** as follows:

$$\begin{array}{lll}
\mathsf{Ite1} & [\mathbf{ite}(\emptyset,\zeta_2,\zeta_3)](t) & \mapsto\!\!\!\twoheadrightarrow \ [\zeta_3](t) \\
\mathsf{Ite2} & [\mathbf{ite}(\{s\}\cup E),\zeta_2,\zeta_3](t) & \mapsto\!\!\!\twoheadrightarrow \ [\zeta_2](\{s\}\cup E) \\
\mathsf{Ite} & [\mathbf{if}\ \zeta_1\ \mathbf{then}\ \zeta_2\ \mathbf{else}\ \zeta_3](t) & \mapsto\!\!\!\twoheadrightarrow \ [\mathbf{ite}([\zeta_1](t),\zeta_2,\zeta_3)](t)
\end{array}$$

6. The rewriting tower and the typing of strategies

Let us now show how to iterate (ad infinitum) the construction of primal, elementary and defined strategies, and how this iteration is related to a typed hierarchy of strategies.

We assume first-order terms classified by a set of sorts $\mathcal{S}_0$ whose elements are denoted by s. These terms are built on a set of function symbols $\mathcal{F}_0$ and used to build a set of labeled rules $\mathcal{R}_0$. The 3-tuple $(\mathcal{S}_0,\mathcal{F}_0,\mathcal{R}_0)$ defines the initial user's theory $\mathcal{RT}_0$. Strategies representing functions over these terms do not belong to this theory. To build the theory $\mathcal{RT}_1$ for strategies, $\mathcal{F}_0$ is imported, rules in $\mathcal{R}_0$ (or equivalently their labels) become strategy operators, other operators are added (**id**, **fail**, **dc**, **dk**, **first**, . . .), and defined functions (like **map**) are introduced. In $\mathcal{RT}_1$, it is natural to understand the type of a primal strategy $l \to r$ as $\langle s \mapsto s' \rangle$ where (s, s') is the pair of sorts of respectively l and r. Moreover, it is reasonable to assume that the user's rewrite rules are sort-preserving, i.e. that s and s' are the same. Then, strategies in $\mathcal{RT}_1$ are typed on a set of sorts $\mathcal{S}_1$ whose elements are $\langle s \mapsto s \rangle$ with s in $\mathcal{S}_0$. In $\mathcal{RT}_1$, rules are also involved to define application of strategies (objects of $\mathcal{RT}_1$) to terms (objects of $\mathcal{RT}_0$). These rules can become in turn primal strategies at an upper level of theory.

Example 6.1 In Example 5.2, the strategy constructor **if** @ **then** @ **else** @ has rank:

$$\mathbf{if}\ \zeta_1\ \mathbf{then}\ \zeta_2\ \mathbf{else}\ \zeta_3 : (\langle s \mapsto s \rangle\ \langle s \mapsto s \rangle\ \langle s \mapsto s \rangle)\ \langle s \mapsto s \rangle,$$

meaning that the first, second and third arguments as well as the co-arity of the operator are all of strategy sort $\langle s \mapsto s \rangle$.

The first-order strategies in the theory $\mathcal{RT}_1$ are objects (or terms) of this theory. Then, we can construct second-order strategies representing functions over objects of $\mathcal{RT}_1$. To deal with these new objects, we extend in the same way the theory $\mathcal{RT}_1$ into $\mathcal{RT}_2$. Clearly this construction can be repeated as much as needed.

Let us detail the construction of the theory $\mathcal{RT}_{i+1}$ from $\mathcal{RT}_i$. In particular, the case $i = 0$ covers all already presented strategies.

1. If the theory $\mathcal{RT}_i$ contains sorts $s \in \mathcal{S}_i$, the sorts of $\mathcal{RT}_{i+1}$ are $\{\langle s \mapsto s\rangle \mid s \in \mathcal{S}_i\} \cup \mathcal{S}_i$. $\langle s \mapsto s\rangle$ is the sort of a sort-preserving strategy over s. The general case of sort-changing strategies (i.e. $\langle s \mapsto s'\rangle$) is a bit more technical, and is developed in [3].

2. If the theory $\mathcal{RT}_i$ contains a symbol $f \in \mathcal{F}$ with rank $f : (s_1, \ldots, s_n)\ s$, or a labeled rewrite rule $[\ell]\ l \rightarrowtail r$, where $l, r : s$ and $x_i : s_i$ are variables of ℓ, the following symbols of the theory $\mathcal{TR}_{i+1}$ are primal strategies inherited from the i-th level:

$$\begin{array}{lll} \mathbf{f} & : & (\langle s_1 \mapsto s_1\rangle \ldots \langle s_n \mapsto s_n\rangle)\ \ \langle s \mapsto s\rangle \\ \ell & : & (\langle s_1 \mapsto s_1\rangle \ldots \langle s_n \mapsto s_n\rangle)\ \ \langle s \mapsto s\rangle \\ & & \text{where } \{x_1 : s_1, ..., x_n : s_n\} = \mathcal{V}ar(l) \cup \mathcal{V}ar(r) \end{array}$$

3. Primal strategies are made of strategy operators inherited from the previous step, of two constant strategies **id** and **fail**, and of the concatenation operator ";" which are overloaded with the following type declarations:

$$\begin{array}{lll} ; & : & (\langle s \mapsto s\rangle\ \langle s \mapsto s\rangle)\ \ \langle s \mapsto s\rangle \\ \mathbf{id} & : & \langle s \mapsto s\rangle \\ \mathbf{fail} & : & \langle s \mapsto s\rangle \end{array}$$

4. Elementary strategies are obtained by adding strategy constructors to handle sets of results. The binary versions of these constructors are overloaded with the following type declarations:

$$\begin{array}{lll} \mathbf{one},\ \mathbf{all} & : & (\langle s \mapsto s\rangle)\ \ \langle s \mapsto s\rangle \\ \mathbf{select\text{-}one},\ \mathbf{select\text{-}first},\ \mathbf{select\text{-}all} & : & (\langle s \mapsto s\rangle\langle s \mapsto s\rangle)\ \ \langle s \mapsto s\rangle \\ \mathbf{dk},\ \mathbf{dc},\ \mathbf{first},\ \mathbf{dc\text{-}one},\ \mathbf{first\text{-}one} & : & (\langle s \mapsto s\rangle\langle s \mapsto s\rangle)\ \ \langle s \mapsto s\rangle \end{array}$$

5. Defined strategies over objects of the theory $\mathcal{TR}_i$ become new strategy symbols of the theory $\mathcal{TR}_{i+1}$, e.g.

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

6. The symbol of the strategy application $@ : (\langle s \mapsto s\rangle\ s)\ s$ is overloaded with this new type declaration.

Up to this point, we have described the syntax (i.e. the signature) of the theory $\mathcal{TR}_{i+1}$. The semantics of strategies (i.e. objects from $\mathcal{TR}_{i+1} - \mathcal{TR}_i$) are *also* defined by a set of rewrite rules specifying a *default* strategy interpreter. These rules have been described in Sections 3, 4 for the primal and elementary strategies, and in Section 5 for defined strategies.

Proposition 6.1 The application of rules in BP and ES is sort-preserving at each level of the hierarchy.

Proof. (Sketch) Both left and right-hand sides of all rules of the strategy interpreter are well-typed terms in the theory $\mathcal{TR}_{i+1}$, and sorts of both sides of each rule are equal. $\square$

We have sketched the construction of strategy symbols with their interpretations in the theory $\mathcal{TR}_{i+1}$. When used in a programming environment and from the user's point of view, this construction should come in general for free. However, if needed, the user can redefine or complete the behavior of the strategy interpreter; he can extend the set of defined strategy constructors, or describe equalities among objects of the theory $\mathcal{TR}_{i+1}$ by rewrite rules in this theory.

For example, in the theory $\mathcal{TR}_{i+1}$, we can also define labeled rewrite rules over strategies, e.g.

$$\begin{array}{lllll} [\mathbf{Dm}] & \mathbf{map}(\zeta_1)\ ;\ \mathbf{map}(\zeta_2) & \rightarrowtail & \mathbf{map}(\zeta_1\ ;\ \zeta_2) \\ [\mathbf{Im}] & \mathbf{map}(\mathbf{id}) & \rightarrowtail & \mathbf{id} \end{array}$$

transforming strategies into simpler ones. The symbols **Dm** and **Im** represent labels of rewrite rules in the theory $\mathcal{TR}_{i+1}$, but in $\mathcal{TR}_{i+2}$, they are primal strategies, thus they can be used for the construction of a reduction meta-strategy like

$$\mathbf{redmap} \rightarrowtail \mathbf{first}(\mathbf{Im}, \mathbf{Dm}).$$

7. Strategies in ELAN

ELAN is an environment for specifying and prototyping deduction systems in a language based on *computational systems*, i.e. sets of rewrite rules with strategies. The language has been developed first in order to specify in a uniform framework and efficiently execute computational systems describing constraint solvers, theorem provers and kernels of programming languages [38, 23, 7]. The language semantics is based on the strategy notion presented above and we now briefly describe some of the strategy features offered by the system.

7.1. Rewriting with Strategies

ELAN provides a kernel that implements the leftmost innermost standard rewriting strategy, the elementary strategies, and which allows iterating the construction on defined strategies.

An ELAN program consists of two parts: a set of rewrite rules and a set of strategies describing the rule application. A rewrite rule in ELAN is of the form:

$$\begin{array}{lll} [\texttt{label}]\ l & => & r \\ & & \texttt{where}\ p_1 := \ll S_1 \gg t_1 \\ & & \ldots \\ & & \texttt{where}\ p_n := \ll S_n \gg t_n \end{array}$$

in which

- $l, r, p_1, \ldots, p_n, t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,
- $\mathcal{V}ar(p_i) \cap (\mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})) = \emptyset$,
- $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_n)$,
- $\mathcal{V}ar(t_i) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \cdots \cup \mathcal{V}ar(p_{i-1})$,
- $S_1, \ldots, S_n$ are strategy terms built from the strategy constructors or defined by the user,
- and $\ll @ \gg @$ is either the application operator —denoted in ELAN (@)@— of elementary strategies on terms, or the application operator —denoted in ELAN [@]@— of defined strategies on terms.

For each matching condition `where` $p :=\ll S \gg t$, the term p is matched successively to each result in $\ll S \gg\ t$, where S denotes a strategy and t a term. This matching process provides values for the variables in p. A rule may have an empty label [] and a matching condition may be of the form `where` $p := ()t$. A matching condition `where true` $:= ()\ c$, where c is a boolean term, is written as a boolean condition `if` c.

When programming in ELAN, it is important to realize the difference between:

– **labeled rules** whose evaluation is fully controlled by the user strategies and,

– **unlabeled rules** which are intended to perform deterministic computations and are applied using a built-in leftmost innermost strategy.

To make the system easier to use, three basic evaluation mechanisms for strategies are available.

7.1.1. Leftmost Innermost Strategy

The first evaluation mechanism available in ELAN is the leftmost innermost standard rewriting strategy. It is used for applying the set of all unlabelled rules. It is also called in matching conditions of the form `where` $p := ()t$.

Example 7.1 A typical example of such a rule set, given in the ELAN syntax, is the specification of the Fibonacci function:

```
rules for int
  n : int ;
  global
   [] fib(0) => 1 end
   [] fib(1) => 1 end
   [] fib(n) => fib(n - 1) + fib(n - 2) if n > 1 end
end
```

The program is executed by giving to the evaluator top level the query `fib(33)`, which indeed means

“evaluates `[built-in-leftmost-innermost-strategy](fib(33))`”.

The set of unlabeled rules is assumed to be terminating and Church-Rosser. For instance, we can implement the PSA calculus (see Figure 1) as a set of unlabeled rules.

7.1.2. Built-in Primal and Elementary Strategies

The second kind of built-in strategy provided by the system consists in applying primal and elementary strategies. The rules in PSA are implemented as a built-in rewriting operation, and all rules in BP and ES are implemented as built-ins.

Example 7.2 A simple example, involving associative commutative rewriting and the built-in strategies **dk** or **first**, is a program that extracts all the elements of a multiset. Assuming that an element is automatically coerced to a singleton when needed, the main rule has the following form:

```
rules for elem
  S : set;  e : elem;
  global
   [extractrule]   element(S U e) => e   end
end
```

where the operator `U` (the union operation on multisets) is assumed to be associative and commutative. The `element` operator returns one of the elements in its argument. In this context, if the user calls the strategy `dk(extractrule)` on a term of sort `set`, he gets all elements of the multiset. The strategy `first(extractrule)` on the contrary, returns only one element of the multiset. The request

$$[\texttt{dk}(\texttt{extractrule})](\texttt{1 U 2 U 1})$$

is evaluated by the system into `{1,2,1}`.

7.1.3. User-Defined Strategies

As explained in the previous sections, the application of strategies on terms is also defined by rewrite rules in BP, ES and the set of rules for defined strategies, as described before using (for clarity) the $\mapsto\!\!\!\!\twoheadrightarrow$ symbol. Indeed these rules can also be defined as ELAN rules. This provides a rewrite system describing the meta-interpreter of strategies. The third evaluation mechanism, for the meta-interpreter, is given through a built-in strategy *eval*, which is applied to strategy rules using their labels. By convention, for defined strategy rewrite rules, when the label is a single dot (`[.]`), the system automatically transforms the rule
`[.]  L => R` into the meta-interpreter rule `[Dstr] [L](t) => [R](t)` where `t` is a variable of the right sort. The strategy *eval* has been implemented in ELAN by using the rules described in this paper, and built-in primal and elementary strategies provided by ELAN.

7.2. The ELAN *Environment*

The ELAN system provides a library of built-ins and standard modules, an interpreter and a compiler [39, 32]. In compiled mode, ELAN can apply more than 15 millions of unlabeled rewrite rules per second on alpha-powered machines (300/500).

For instance, it can evaluate `fib(33)` in 0.695 seconds. For theorem proving applications, where strategies and labeled rules are heavily used, it rewrites at the average speed of 800 000 labeled rules per second. As an example, the completion of non-Abelian groups is performed in less than 1 second.

More informations and access to the system as well as web demos are available at `http://www.loria.fr/ELAN`.

7.3. Examples of more complex strategies

In order to illustrate the previous concepts and how they can be used in ELAN (these examples are actually running), we introduce now two examples of increasing difficulty. The first one illustrates the strategy language describing the normalization of λ-terms using the leftmost innermost and leftmost outermost strategies. λ-terms are represented using de Bruijn notation, and thus there are two primitive operations: $free(v,t)$ meaning that a variable v is free in a λ-term t, and $replace(v,a,t)$ standing for $\{(v \mapsto t)\}(a)$. The definition of λ-terms and *beta* and *eta* evaluation rules is straightforward in ELAN:

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

The definition of the innermost or outermost normalization strategies $lis(s)$ and $los(s)$ is done in a natural way by writing:

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

The strategies $lis(S)$ and $los(S)$ fail if the input λ-term does not contain any S-redex reducible with the substrategy S (later instantiated to $\mathbf{first}(beta, eta)$). The strategy $lis(S)$ tries to apply the substrategy $(lis(S)\ id)$, which succeeds if the input term has a form $(M\ N)$ and $lis(S)$ is applicable to M, i.e. if M contains an S-redex. Otherwise, it continues on the right λ-subterm N by the application of $(id\ lis(S))$. If the input term has the form $\lambda\ M$, the strategy $lis(S)$ is propagated towards M by the application of $la\ lis(S)$. If none of the three cases above succeeds, the strategy S is applied to the top of this λ-term. The fact that S is applied in this order, makes the crucial difference between strategies $lis(S)$ and $los(S)$.

The second example shows higher-order strategies on the classical example of the Church's numbers that can be defined, using the standard notations of lambda calculus, by two functions *zero* and *succ*:

$$\begin{aligned}
&zero : (X \to X) \to (X \to X)\\
&zero = \lambda f.\lambda x.x\\
&succ : ((X \to X) \to (X \to X)) \to (X \to X) \to (X \to X)\\
&succ = \lambda n.\lambda f.\lambda x.(f\ ((n\ f)\ x))
\end{aligned}$$

These functions can be represented as strategies in several ways, because of the absence of currying for defined strategies. The function *zero* can be defined as a binary function, a unary defined strategy, or a nullary defined strategy where `X` denotes a parameter sort:

```
  zero2(@,@) : (<X->X> X) X;
  zero1 @    : (<X->X>) <X->X>;
  zero0      : < <X->X>-><X ->X> >;
```

The semantics of these symbols can be easily defined by:

```
rules for X
global
[]   zero2(f,x) => x              end
end
strategies for X
explicit
[.]  [zero1 f] x => x             end
end
strategies for <X->X>
explicit
[.]  [zero0] f => id              end
end
```

where $\texttt{f} : \langle \texttt{X} \mapsto \texttt{X}\rangle, \texttt{x} : \texttt{X}$ are variables. The explicit definition of $zero1$ is equivalent to the following implicit one:

```
strategies for X
implicit
[.]  zero1 f => id                end
end
```

but not to the third one. Similarly, there are several possibilities for defining *succ*:

```
succ3(@,@,@): (<<X->X>-><X->X>> <X->X> X) X;
succ2(@,@)  : (<<X->X>-><X->X>> <X->X>) <X->X>;
succ1 @     : (<<X->X>-><X->X>>) <<X->X>-><X->X>>;
succ0       : <<<X->X>-><X->X>>-><<X->X>-><X->X>>>;
```

The definition of *succ*3 is quite natural from the functional point of view:

```
rules for X
global
[]   succ3(n,f,x) => [f]([[n](f)](x))     end
```

where $\mathtt{n} : \langle\langle \mathtt{X} \mapsto \mathtt{X}\rangle \mapsto \langle \mathtt{X} \mapsto \mathtt{X}\rangle\rangle$, $\mathtt{f} : \langle \mathtt{X} \mapsto \mathtt{X}\rangle$, and $\mathtt{x} : \mathtt{X}$ are rule variables. The right-hand side of the rule is not far from the original functional definition. The term $[\mathtt{f}]([[\mathtt{n}](\mathtt{f})](\mathtt{x}))$ is an abbreviation of the term `x2` which is defined by the following sequence of `where` local affectations: $\mathtt{f1} := [\mathtt{n}](\mathtt{f}), \mathtt{x1} := [\mathtt{f1}](\mathtt{x}), \mathtt{x2} := [\mathtt{f}](\mathtt{x1})$. For a better understanding, the first application symbol in the expression $\mathtt{f1} := [\mathtt{n}](\mathtt{f})$ has the type $(\langle\langle \mathtt{X} \mapsto \mathtt{X}\rangle \mapsto \langle \mathtt{X} \mapsto \mathtt{X}\rangle\rangle\ \langle \mathtt{X} \mapsto \mathtt{X}\rangle)\ \langle \mathtt{X} \mapsto \mathtt{X}\rangle$, while those in the expressions for `x1` and `x2` have the following type $(\langle \mathtt{X} \mapsto \mathtt{X}\rangle\ \mathtt{X})\ \mathtt{X}$.
The definition of *succ*2 is similar:

```
strategies for X
explicit
[.]  [succ2(n,f)] x => [f]([[n](f)](x))             end
```

The explicit definition of `succ1` introduces primal strategies $l \to r$ with the syntax `[l => r]`:

```
strategies for <X->X>
explicit
[.]  [succ1 n] f => [ x => [f]([[n](f)](x)) ]        end
```

where `[ x => [f]([[n](f)](x)) ]` stands for the primal strategy referencing the rewrite rule $x \to [f]([[n](f)](x))$. A more complex example is the definition of *succ*0:

```
strategies for <<X->X>-><X->X>>
explicit
[.]  [succ0] n => [f => [x => [f]([[n](f)](x)) ] ]   end
```

8. Dynamic types and ad-hoc polymorphism

Polymorphism is an important feature of functional programming languages. We explain in this section how dynamic types and ad-hoc polymorphism can be provided in ELAN, thanks to a module `any` available in the library.

8.1. Dynamic types

Let us first explore an idea based on a simple sort construction called dynamics [27], or dynamic typing [1, 20], which allows to define functions that essentially require run-time type-checking in statically typed language. A weakness of static typing can be eliminated by the construction of objects with dynamic types (or dynamics, for short) [28], or by the introduction of an advanced polymorphic system [31]. Both methods were combined in [8], where the concept of dynamics is generalized into dependent data types. Another approach combining dynamics and the parametric polymorphism is studied in [27] from the point of the type-inference.

The definition *type Any* $= (t : \mathit{Type},\ x : t)$ introduces a sort *Any* (called *dyn* in [27, 1]), which is the simplest example of a dependent data type. The first part of the pair *Any* indicates the sort of the second part. This means that the type t ranges over the domain of all types, and the type of x depends on the value of t. There are different approaches with respect to the construction of the set of all types [28, 31, 8], but the situation we consider here is much simpler, since we restrict to first-order rewrite rules and to finite sets of sorts $\mathcal{S}$.

The sort *Any* is defined in the parametric module `any[X]`:

```
module any[X]
sort Any; end
operators global
  (X)@   : (X) Any;
end
```

where the argument `X` of this module is supposed to be (the name of) a sort. This module defines also one *embedding* coercion wrapping up terms of sort `X` into terms of sort `Any`, in such a way, that if `t:X`, then `(X)t : Any`. A projection $(\mathtt{X}^{-1})$@ : `(Any) X` from the sort `Any` into `X` can be also defined by: $(\mathtt{X}^{-1})(\mathtt{X})\mathtt{t} \rightarrow \mathtt{t}$.

This simple construction of dynamics gives the possibility to define functions in an 'ad-hoc' polymorphic style, to introduce a set of general term constructors and destructors, and to write polymorphic strategies.

8.2. Ad-hoc polymorphic functions

A simple example of an 'ad-hoc' polymorphic function is the function `export` converting terms into a list of lexems. The 'ad-hoc' polymorphism of `export` means that it works for terms of different types, and terms of different type are treated in different ways. Lexems such as integers and strings, can be represented as `Any`-terms (i.e. terms of sort `Any`), namely `(int)n` and `(string)s`, where `n:int` and `s:string`. The function `export : (Any)  list[Any]` is defined on these two elementary sorts:

```
  []    export((int)n) => (int)n . nil                  end
  []    export((string)s) => (string)s . nil            end
```

Then on other sorts, for instance `list[X]`, `export` is defined as follows:

```
[] export((list[X])nil) => (string)"nil" . nil     end
[] export((list[X])a.as) => export((X)a)+(string)"." . export((list[X])as) end
```

where the symbol `+` stands for the *append* and `.` for *cons* over `list[Any]`. For terms of an arbitrary type `s`, there are rules for each constructor `f` : $(\mathtt{s}_1 \ldots \mathtt{s}_\mathtt{n})$`s` where `ti` are variables of sort `si`:

```
  []    export((s)f(t1,...,tn)) =>
                (string)"f(" . export((s1)t1)+(string)"," .
                . . . . . . . . . . . . . .
                export((sn)tn)+(string)")" . nil                end
```

So the sort `Any` allows defining 'ad-hoc' polymorphic functions dealing with terms of an arbitrary sort. However, in general, for each function and each constructor, a rule defining the behavior of the function over that constructor has to be provided. The first possible solution is to generate these rules with a pre-processor that automatically enriches programs. We explore another solution, conceptually simpler, in the next section.

8.3. Term constructors and destructors

Basic operations with `Any`-terms are composition and decomposition, defined by two function symbols `explode` and `implode`. The symbol `explode` transforms an `Any`-term into a list of `Any`-subterms. To be able to construct back the initial term, we introduce a symbol `implode`, that takes a list of `Any`-subterms `ti` and a function symbol `f`, and constructs the `Any`-term `f(t1,...,tn)`. If `f` is the top-most function symbol of a term `t`, then `t = implode(f,explode(t))` holds. In this situation, we have to consider function symbol as constants. Therefore, we introduce a new sort `Functor`, which collects all function symbols: if `f(@,...,@) : (s1...sn) s` is in the signature, then the following constant is introduced in the sort `Functor`: `f(s1,...,sn) : Functor`.

The module `Any` introduces the sort `Functor`, a function `functor` which extracts a function symbol from an `Any`-term, and the functions `explode` and `implode`:

```
module Any
import global list[Any]; end
sort Any Functor; end
operators global
  functor(@)    : (Any) Functor;
  explode(@)    : (Any) list[Any];
  implode(@,@)  : (Functor list[Any]) Any;
end
```

Example 8.1 Let us illustrate these functions on a small example. In a module importing `any[int]` and `any[tree[int]]`, the sort `tree[int]` is defined as follows:

```
operators global
  empty         : tree[int];
  leaf(@)       : (int) tree[int];
  node(@,@,@)   : (tree[int] int tree[int]) tree[int];
end
```

An `Any` term `(tree[int])node(leaf(3),4,node(leaf(5),6,empty))` is destructed with `explode` into a list of `Any`-terms:

```
  explode((tree[int])node(leaf(3),4,node(leaf(5),6,empty))) =>
      (tree[int])leaf(3).(int)4.(tree[int])node(leaf(5),6,empty).nil
```

To get back, we use the symbol `implode` applied on this list with a function symbol name `node(tree[int],int,tree[int]):Functor`, and we obtain:

```
  implode(node(tree[int],int,tree[int]),
        (tree[int])leaf(3).(int)4.(tree[int])node(leaf(5),6,empty).nil) =>
                (tree[int])node(leaf(3),4,node(leaf(5),6,empty))
```

The functions `explode`, `implode` and `functor` are defined by three sets of rewrite rules, generated and imported automatically, whenever the user imports the module `any[X]`. For instance, for the symbol `node(@,@,@)`, the following rules are automatically generated:

```
[] explode((Any)node(x,y,z)) => (tree[int])x.(int)y.(tree[int])z.nil   end
[] implode(node(tree[int],int,tree[int]),
           (tree[int])x.(int)y.(tree[int])z.nil)
   =>  ((Any)node(x,y,z))                                              end
[] functor((Any)node(x,y,z)) => node(tree[int],int,tree[int])          end
```

where `x,z:tree[int]` and `y:int` are variables. These rules can be correctly parsed if the following function symbol name is declared:
`node(tree[int],int,tree[int]):Functor`.
Similar rewrite rules and symbol definitions are also automatically produced for other constructors of the sort `tree[int]`, and in general, for any function symbol `f:(s1...sn)s` in the signature, provided the user has already imported modules `any[si]` and `any[s]`, for any $i = 1..n$.

This feature allows us to include typed terms using type flags (also called type decorations) into mono-sorted `Any`-terms. We can also make projections of `Any`-terms into a particular sort `X`, which is a run-time test of a type flag (it can fail). We can define various symbols and strategies over this sort `Any`, which can be viewed (from the point of view of many-sorted systems) as polymorphic symbols or strategies. This is explored in the next section.

8.4. Polymorphic strategies

Let us concentrate now on the definition of 'polymorphic' strategies, i.e. strategies working over an arbitrary user-defined sort. Our goal is to get a polymorphic version of the 'leftmost innermost' strategy, which runs over any sort. The module presented in Figure 2 defines a strategy `lis(S):(<X->X>) <Any->Any>` (an abbreviation for 'leftmost innermost').
This definition introduces two auxiliary strategies, `map` and `mis`. The strategy `mis(S)` assigns to the variable `as` the list of subterms of `a`, on which the strategy `map(S)` is applied. The final result is constructed by `implode(functor(a),as)` (it is an `Any`-term). If `mis(S)` fails, the strategy $(\texttt{X})\texttt{S} : \langle\texttt{Any}\rangle$, is applied. The operational meaning of this strategy expression is as follows: when applied to an `Any`-term, it destructs it into a term of sort `X` (which is, in fact, a test of the type flag) and then, it applies the strategy `S` on the obtained term. Finally, it constructs back the result term from the result of the strategy application (which is of sort `X`) as an `Any`-term.

The second auxiliary strategy $\texttt{map(S)} : \langle\texttt{list[Any]}\rangle$ tries to apply the strategy `lis(S)` to the input list. Actually, it applies `lis(S)` on the head, and if it fails, `map(S)` continues on the tail of this list. If the input list does not contain any element, on which the strategy `lis(S)` is applicable, `map(S)` fails too.

By swapping two strategy expressions in `lis(S)`, we obtain a 'left-most outermost' polymorphic strategy.

```
module lis[X]
import global Any list[Any] strat[X] strat[Any] strat[list[Any]];
end
stratop global
  lis(@)        : (<X->X>) <Any->Any>;
local
  map(@)        : (<X->X>) <list[Any]->list[Any]>;
  mis(@)        : (<X->X>) <Any->Any>;
end
strategies for Any
S   : <X->X>;   a   : Any;
                as  : list[Any];
implicit
  [.] lis(S) => first(mis(S),(X)S)                      end
explicit
  [.] [mis(S)]a => implode(functor(a),as)
                       where as:=[map(S)]explode(a)    end
end
strategies for list[Any]
S  : <X->X>;
implicit
  [.] map(S) => first(cons(lis(S),id), cons(id,map(S))) end
end
```

Figure 2: Left-most inner-most strategy

9. Conclusion

In this paper, we have presented rewriting from a functional point of view. This allows giving a functional semantics to rewrite based environments like ELAN [7], Maude [13] and Claire [9] where strategies play a crucial role. Applications of such languages range from knowledge representation to constraint solving, theorem proving and language executable descriptions. Based on many non-trivial examples run with ELAN, we believe that the "rewriting tower" approach induced by the formalism developed in this paper is extremely attractive in terms of expressiveness and readability for such applications.

Moreover, in addition to the functional semantics developed here, such languages have a logical semantics based on rewriting logic [29] which allows us in particular to understand strategies as sets of proof terms of a specific rewrite theory, as introduced in [5]. This latter approach provides a reflective view of languages like ELAN [25] or Maude [14], in the lineage of Scheme [36], and Nuprl [15].

Based on the ideas developed here, we are now working on the definition of a *rewriting calculus*, called the ρ-calculus, which is a higher-order calculus fully integrating the notion of strategies, and thus of rule application at the object level [11, 12].

Acknowledgements

We thank Pierre-Etienne Moreau, Marian Vittek and Horatiu Cirstea for all our interactions on ELAN from the design to the implementation, José Meseguer and Manuel Clavel for many valuable discussions on strategies, Christopher Lynch for its comments on an earlier draft of this paper and the anonymous referees for their careful reading and suggestions.

The second author has been partially supported by the Grant-in-Aid for Scientific Research on Priority Areas "Research on the Principles for Constructing Software with Evolutionary Mechanisms", from Ministry of Education, Science and Culture of Japan.

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

References

1. M. Abadi, C. Luca, P. Benjamin, and P. Gordon. Dynamic typing in a statically-typed language. In *POPL'89*, pages 213–227, Israel, 1989. Hebrew University.
2. F. Baader and T. Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.
3. 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 LORIA 98-T-326.
4. 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.
5. 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 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), September 1996.
6. 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.
7. 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.
8. P. Borovanský and P. Voda. Implementation of type as a value polymorphism. In *Proceeding SOFSEM '93 (SOFtware SEMinar), 1993, Part II.*, 1993.
9. Y. Caseau and F. Laburthe. Combining objects and rules for problem solving. In *In JICSLP'96 Workshop on Multi-paradigm Programming*, Bonn, Germany, 1996.
10. C. Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, 34:263–293, September 1998.
11. H. Cirstea and C. Kirchner. The rewriting calculus as a semantics of ELAN. In J. Hsiang and A. Ohori, editors, *4th Asian Computing Science Conference*, volume

1538 of *Lecture Notes in Computer Science*, pages 8–10, Manila, The Philippines, December 1998. Springer-Verlag.

12. 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 2*, Research Studies, ISBN 0863802524, pages 95–120. Wiley, 1999.
13. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.
14. 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.
15. R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, Inc., Englewood Cliffs, New Jersey 07632, 1986.
16. 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.
17. T. Frühwirth and P. Hanschke. Terminological reasoning with constraint handling rules. In P. V. Hentenryck and V. J. Saraswat, editors, *Principles and Practice of Constraint Programming*. The MIT press, April 1995. (Revised version of Technical Report ECRC-94-6).
18. M. Gordon, A. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, New York (NY, USA), 1979.
19. M.-J.-C. Gordon and T.-F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993. ISBN 0-521-44189-7.
20. F. Henglein. Dynamic typing. In B. Krieg-Brückner, editor, *Proceedings of ESOP'92*, volume 582 of *Lecture Notes in Computer Science*, pages 233–253. Springer-Verlag, 1992.
21. J.-P. Jouannaud. Syntactic theories. In *Proceedings Mathematical Foundations of Computer Science*, volume 452 of *Lecture Notes in Computer Science*, pages 15–25, Banska Bystrica (Czechoslovakia), 1990. Springer-Verlag.
22. 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.
23. 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.
24. C. Kirchner and F. Klay. Syntactic theories and unification. In *Proceedings 5th IEEE Symposium on Logic in Computer Science, Philadelphia (Pa., USA)*, pages 270–277, June 1990.
25. 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.

26. F. Klay. Undecidable properties in syntactic theories. In R. V. Book, editor, *Proceedings 4th Conference on Rewriting Techniques and Applications, Como (Italy)*, volume 488 of *Lecture Notes in Computer Science*, pages 136–149. Springer-Verlag, April 1991.

27. X. Leroy and M. Mauny. Dynamics in ml. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, Cambridge, MA, USA, august 1991. Springer-Verlag.

28. C. Luca, D. Jim, J. Mick, K. Bill, and N. Greg. The module-3 type system. In *POPL'89*, pages 202–212, Israel, 1989. Hebrew University.

29. 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.

30. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

31. R. Milner and M. Tofte. *The definition of Standard ML*. The MIT press, Cambridge, Massachusetts and London, England, 1991.

32. P.-E. Moreau and H. Kirchner. A compiler for rewrite programs in associative-commutative theories. In *"Principles of Declarative Programming"*, number 1490 in Lecture Notes in Computer Science, pages 230–249. Springer-Verlag, September 1998. Report LORIA 98-R-226.

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

34. L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

35. L. C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic in Computer Science*. Academic Press, 1990.

36. B. Smith. Reflection and Semantics in Lisp. In *Proceedings 11th ACM Symposium on Principles of Programming Languages, Salt Lake City*. ACM Press, 1984.

37. G. Smolka. A foundation for higher-order concurrent constraint programming. In J.-P. Jouannaud, editor, *Proceedings of the 1st International Conference on Constraints in Computational Logics, Munich (Germany)*, volume 845 of *Lecture Notes in Computer Science*, pages 50–72. Springer-Verlag, 1994.

38. 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.

39. M. Vittek. A compiler for nondeterministic term rewriting systems. In H. Ganzinger, editor, *Proceedings of RTA'96*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–168, New Brunswick (New Jersey), July 1996. Springer-Verlag.