Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 174 (2007) 3–25
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Deduction, Strategies, and Rewriting

Steven Eker[a], Narciso Martí-Oliet[b], José Meseguer[c], and Alberto Verdejo[b]

[a] *Computer Science Laboratory, SRI International, Menlo Park, CA, USA*

[b] *Facultad de Informática, Universidad Complutense de Madrid, Spain*

[c] *Department of Computer Science, University of Illinois at Urbana-Champaign, IL, USA*

Abstract

Automated deduction methods should be specified not *procedurally*, but *declaratively*, as inference systems which are proved correct regardless of implementation details. Then, different algorithms to implement a given inference system should be specified as *strategies* to apply the inference rules. The inference rules themselves can be naturally specified as (possibly conditional) *rewrite rules*. Using a high-performance rewriting language implementation and a strategy language to guide rewriting computations, we can obtain in a modular way implementations of both the inference rules of automated deduction procedures and of algorithms controling their application. This paper presents the design of a strategy language for the Maude rewriting language that supports this modular decomposition: inference systems are specified in *system modules*, and strategies in *strategy modules*. We give a set-theoretic semantics for this strategy language, present its different combinators, illustrate its main ideas with several examples, and describe both a reflective prototype in Maude and an ongoing C++ implementation.

Keywords: Rewriting logic, Maude, rewriting strategies, congruence closure.

1 Introduction

Automated deduction methods (e.g. congruence closure, resolution, etc.) should be specified, not *procedurally*, as low level algorithms (based on pointer manipulation, etc.), but *declaratively* as inference systems, which are proved correct regardless of implementation details. Then, specific algorithms to implement a given inference system should be specified at a high level as *strategies* to apply the inference rules. For example, Shostak's [24] and Downey-Sethi-Tarjan's [13] congruence closure algorithms can be viewed as different strategies to apply the same congruence closure inference rules [25,2].

This allows a modular separation between the inference rules themselves and their control through strategies. In this way, one can reason, for example, about

⋆ Research partially supported by Spanish MEC project MIDAS (TIC2003–01000), by CAM program PROMESAS (S–0505/TIC–0407), and by ONR grant N00014-02-1-0715.

doi:10.1016/j.entcs.2006.03.017

key correctness properties of an inference system while leaving open various control issues on how to apply the inference rules in an optimal way. This is a much better way to specify an inference system than the traditional algorithmic descriptions of deductive procedures in which the logical and control aspects are merged and confused together.

This abstract methodology can be and has been applied not just to isolated examples, but to a very wide range of automated deduction methods. It is by now customary as a way to specify at an abstract level many logical inference systems. Without in any way trying to be exhaustive, which would be quite impossible, we can mention by way of examples:

- Unification (Martelli & Montanari [18], Jouannaud & Kirchner [15]).
- Congruence closure (Kapur [16], Tiwari [2,25]).
- Nelson-Oppen combination of decision procedures (Tiwari [25], Conchon & Krstic [12], Nieuwenhuis, Oliveras & Tinelli [22]).
- Knuth-Bendix completion (Bachmair & Dershowitz [1], Lescanne [17]).
- Inductive theorem proving and other Maude tools (Clavel, Durán & Meseguer [11,8]).

Rewriting logic is a simple logical framework in which the inference rules of a logic or of any deductive procedure can be easily expressed as conditional rewrite rules [21,19]. For example, a rule for orienting equalities present in inference systems for completion-like procedures

$$\text{orient}\ \frac{(K, E \cup \{t \approx c\}, R)}{(K, E, R \cup \{t \to c\})} \qquad \text{if } t \succ c$$

corresponds to a conditional rewrite rule

$$\text{orient} : (K, E \cup \{t \approx c\}, R) \longrightarrow (K, E, R \cup \{t \to c\}) \quad \text{if} \quad t \succ c$$

Using a high-performance rewriting language implementation and a strategy language to guide rewriting computations, the above high-level distinction between inference rules and strategies can be directly implemented, providing a way to faithfully represent the high-level specification as an executable prototype or even as an implementation.

An interesting feature of rewriting logic is that, thanks to its reflective capabilities [5,10], strategies themselves can also be specified in a declarative way by means of rewrite rules *at the metalevel*. That is, by rewrite rules that guide one level up how the inference rules of the system we are interested in are applied at the "object level." This reflective approach supports reasoning about metalevel features by means of reflective reasoning techniques [3].

The Maude language [6,7] maintains this modular distinction between rewrite rules and strategies to execute them: a Maude system module is a rewrite theory with no strategy information. In general, the same rewrite theory describing an inference system may be executed with different strategies, which may each have specific advantages depending on the purpose at hand. Therefore, in Maude strate-

gies can be defined by rules at the metalevel in its `META-LEVEL` module. There is great freedom to define in this way different *strategy languages* [9], which can then be used to specify and execute strategies for any object theory of interest. The semantics of the strategy language in question is used to ensure that all computations allowed are correct deductions in the object theory.

However, pragmatic considerations are important to guide strategy language designs that can deal well with relevant applications. Therefore, we have undertaken the project of providing a basic strategy language for Maude [20]. To make the language easier to use we have made it available *at the object level*, rather than at the metalevel. Our strategy language allows the definition of strategy expressions that control the way a term is rewritten. We have benefitted from our own previous experience designing strategy languages in Maude, and also from the experience of other languages like ELAN [4] and Stratego [27]. Our design is based on a strict separation between the rewrite rules in *system modules* and the strategy expressions, that are specified in separate *strategy modules*. Thus, in our proposal it is not possible to use strategy expressions in the rewrite rules of a system module: they can only be specified in a separate strategy module. In fact, this separation makes possible defining different strategy modules to control in different ways the rewrites of a single system module.

A strategy is described as an operation that, when applied to a given term, produces a *set* of terms as a result, given that the process is nondeterministic in general. The basic strategies consist of the application of a rule (identified by the corresponding rule label) to a given term, and allowing variables in a rule to be instantiated before its application by means of a substitution. For conditional rules—which may contain rewrite conditions—such rewrite conditions can also be controlled by means of strategies. Basic strategies are combined by means of several combinators, including: regular expression constructions (concatenation, union, and iteration), if-then-else, combinators to control the way subterms of a given term are rewritten, and recursion [20]. As a new contribution of this paper, we develop the notion of *generic strategies* (e.g., backtracking, map, etc.), which are applicable not to a single rewrite theory, but to a wide range of rewrite theories satisfying some parametric requirements.

In order to validate our strategy language design, we have mainly focused on automated deduction and programming language semantics applications. Besides the short examples presented in [20], the language has been successfully used in the implementation of the operational semantics of the ambient calculus [23], the two-level operational semantics of the parallel functional programming language Eden [14], and basic completion algorithms [26]. Moreover, as a further contribution of this paper, we apply here the strategy language to congruence closure algorithms.

Our first prototype implementation defined the language at the metalevel in the usual reflective way, while keeping the user interface at the object level for ease and convenience. After validating our language design experimentally and reaching a mature language design, we are currently developing a direct implementation of our strategy language at the C++ level, at which the Maude system itself is

implemented, to make the language a stable new feature of Maude, and to allow a more efficient execution. A third contribution of this paper is a description of the main design ideas used in such ongoing implementation.

2 Rewriting logic and Maude

A *rewrite theory* [21] (Σ, E, R) consists of a signature Σ, a set E of equations, and a set R of rules. The static part of a system or logic is specified in an equational sublogic of rewriting logic (membership equational logic) by means of the equations E. The system dynamics (transitions or inferences) is specified by means of possibly conditional rules R that rewrite terms, representing parts of the system, into other terms.

Maude [6,7] is an efficient implementation of rewriting logic. Maude syntax is user-definable and operators can have equational attributes like *associativity* (`assoc`), *commutativity* (`comm`), and *identity* (`id:`), so that rewriting can take place *modulo* such equational axioms.

The general form of a rewrite rule with label l is the following:

$$l : t \longrightarrow t' \;\; \mathit{if} \;\; (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \rightarrow q_k)$$

where t, t', u_i, v_i, w_j, p_k and q_k are terms and s_j are sort identifiers.

In order to apply such a rule to a subject term u, we have to find a subterm of u which is an instance of t modulo the equational axioms of the theory (e.g., associativity, commutativity, identity) with a substitution θ. But before rewriting u by replacing $\theta(t)$ by $\theta(t')$, the rule's condition has to be checked. This means checking that the equalities $\bigwedge_i \theta(u_i) = \theta(v_i)$ hold, the memberships $\bigwedge_j \theta(w_j) : s_j$, stating that each term $\theta(w_j)$ has sort s_j, hold, and that for each k we can rewrite by zero, one, or more steps each term $\theta(p_k)$ to a substitution instance of $\theta(q_k)$ (q_k is typically a pattern that can have extra variables). Assuming that the equations E in the rewrite theory are Church-Rosser and terminating, checking the equations and memberships is a decidable problem. Instead, checking the rewrite conditions requires performing breadth-first search. Since in general such a search process may not terminate, whether a single rewrite step can be performed with a conditional rule can be undecidable.

A Maude *system module* specifies a rewrite theory. The Maude syntax is so close to the corresponding mathematical notation for defining rewrite theories as to be almost self-explanatory. The general point to keep in mind is that each item: a sort, a subsort, an operation, an equation, a rule, etc., is declared with an obvious keyword: `sort`, `subsort`, `op`, `eq` (or `ceq` for conditional equations), `rl` (or `crl` for conditional rules), etc., with each declaration ended by a space and a period. Indeed, a rewrite theory $(\Sigma, E \cup A, R)$ is defined with the signature Σ specified using keywords `sort`, `subsort`, and `op`, equations in E using keyword `eq`, and equational axioms in A using keywords `assoc`, `comm` and `id:`, and rules in R using keyword `rl`. Another important point is the use of "mix-fix" user-definable syntax, with the argument positions specified by underbars; for example: `if_then_else_fi`.

Furthermore, the concrete syntax of equations in conditions has three variants, namely: ordinary equations `t = t'`; matching equations `t := t'`, where the term `t'` is required to match the pattern `t`; and abbreviated Boolean equations of the form `t`, with `t` a Boolean term, abbreviating the equation `t = true`.

For example, the system module `SORTING` below specifies a rewrite theory whose expressions are arrays, represented as sets of index-value pairs, has an equationally-defined `length` function measuring the length of an array, and has a single, unconditional rewrite rule `switch` that switches the values at two array positions.

```
mod SORTING is
  protecting NAT .
  sorts Pair PairSet .
  subsort Pair < PairSet .
  op (_,_) : Nat Nat -> Pair .
  op empty : -> PairSet .
  op __ : PairSet PairSet -> PairSet [assoc comm id: empty] .
  op length : PairSet -> Nat .
  vars I J V W : Nat .  var PS : PairSet .
  eq length(empty) = 0 .
  eq length((I, V) PS ) = length(PS) + 1 .
  rl [switch] : (J, V) (I, W) => (J, W) (I, V) .
endm
```

3 The Maude strategy language

In this section we describe the combinators of our strategy language and their semantics. We have a simple set-theoretic semantics for the strategies of a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ given by a function

$$\_@\_ : Strat \times T_\Sigma(X) \longrightarrow \mathcal{P}(T_\Sigma(X)),$$

which has an obvious extension to a function

$$\_@\_ : Strat \times \mathcal{P}(T_\Sigma(X)) \longrightarrow \mathcal{P}(T_\Sigma(X)),$$

where, if $s \in Strat$ and $U \subseteq T_\Sigma(X)$, we have $s \mathbin{@} U = \bigcup_{t \in U} s \mathbin{@} t$.

3.1 Idle and fail

The simplest strategies are the constants `idle` and `fail`. The first always succeeds, but without modifying the term t to which it is applied, that is, `idle` $@\, t = \{t\}$, while the second always fails, that is, `fail` $@\, t = \emptyset$.

3.2 Basic strategies

The basic strategies consist of the application of a rule (identified by the corresponding rule label) to a given term. In this case a rule is applied *anywhere* in the term where it matches satisfying its condition, with no further constraints on the substitution instantiation. In case of conditional rules, the default breadth-first search strategy is used for checking the rewrites in the condition. Therefore, if l is a rule label and t a term, the semantics of $l \mathbin{@} t$ is the set of terms to which t rewrites

in one step using the rule with label l anywhere where it matches and satisfies the rule's condition.

A slightly more general variant allows variables in a rule to be instantiated before its application by means of a substitution, that is, a mapping of variables to terms, so that the user has more control on the way the rule is applied.

$$\begin{aligned}
\langle \mathit{Substitution} \rangle &::= \texttt{none} \\
&\mid \langle \mathit{Variable} \rangle \texttt{ <- } \langle \mathit{Term} \rangle \\
&\mid \langle \mathit{Substitution} \rangle \texttt{ ; } \langle \mathit{Substitution} \rangle \\
\langle \mathit{BasicStrat} \rangle &::= \langle \mathit{Label} \rangle \\
&\mid \langle \mathit{Label} \rangle \texttt{ [} \langle \mathit{Substitution} \rangle \texttt{]} \\
\langle \mathit{Strat} \rangle &::= \langle \mathit{BasicStrat} \rangle
\end{aligned}$$

The unconstrained case `L` can also be expressed as `L[none]`, where `none` denotes the identity (empty) substitution.

For conditional rules, rewrite conditions can be controlled by means of strategy expressions. As before, the substitution can be omitted if it is empty.

$$\begin{aligned}
\langle \mathit{StratList} \rangle &::= \langle \mathit{Strat} \rangle \\
&\mid \langle \mathit{Strat} \rangle \ \langle \mathit{StratList} \rangle \\
\langle \mathit{BasicStrat} \rangle &::= \langle \mathit{Label} \rangle \texttt{ [[} \langle \mathit{Substitution} \rangle \texttt{]] \{ } \langle \mathit{StratList} \rangle \texttt{ \}}
\end{aligned}$$

A strategy expression of the form `L[S]{E1 ... En}` denotes a basic strategy that applies *anywhere* in a given state term the rule `L` with variables instantiated by means of the substitution `S` and using `E1`, ..., `En` as strategy expressions to check the rewrites in the condition of `L`. The number of rewrite condition fragments appearing in the condition of rule `L` must be exactly n for the expression to be meaningful.

3.3 Top

The most common case allows applying a rule anywhere in a given term, as explained above, but we also provide an operation to restrict the application of a rule only to the *top* of the term, because in some examples like structural operational semantics, the only interesting or allowed rewrite steps happen at the top.

$$\langle \mathit{Strat} \rangle ::= \texttt{top(} \langle \mathit{BasicStrat} \rangle \texttt{)}$$

`top(BE)` applies the basic strategy `BE` only at the top of a given state term. Note, however, that even applying a rule at the top is nondeterministic due to the possibility of multiple matches, because matching takes place modulo the equational attributes of the operators, such as associativity, commutativity, or identity.

3.4 Tests

Tests are seen as strategies that check a property on a state, so that the test *qua* strategy is successful if true and fails if false. In the first case, the state is not

changed. That is, for T a test and t a term, $T \mathbin{@} t$ will evaluate to $\{t\}$ if T succeeds on t, and to $\emptyset$ if it fails, so that T acts as a filter on its input.

Since matching is one of the basic steps that take place when applying a rule, the strategies that test some property of a given state term are based on matching. As in applying a rule, we distinguish between matching anywhere and matching only at the top of a given term.

$$
\begin{aligned}
\langle \mathit{EqCondition} \rangle &::= \langle \mathit{BoolTerm} \rangle \\
&\mid \langle \mathit{Term} \rangle \texttt{ = } \langle \mathit{Term} \rangle \\
&\mid \langle \mathit{Term} \rangle \texttt{ := } \langle \mathit{Term} \rangle \\
&\mid \langle \mathit{EqCondition} \rangle \texttt{ /\ } \langle \mathit{EqCondition} \rangle \\
\langle \mathit{Test} \rangle &::= \texttt{amatch } \langle \mathit{Pattern} \rangle \texttt{ [s.t. } \langle \mathit{EqCondition} \rangle \texttt{]} \\
&\mid \texttt{match } \langle \mathit{Pattern} \rangle \texttt{ [s.t. } \langle \mathit{EqCondition} \rangle \texttt{]} \\
&\mid \texttt{xmatch } \langle \mathit{Pattern} \rangle \texttt{ [s.t. } \langle \mathit{EqCondition} \rangle \texttt{]} \\
\langle \mathit{Strat} \rangle &::= \langle \mathit{Test} \rangle
\end{aligned}
$$

`amatch T s.t. C` is a test that, when applied to a given state term `T'`, is successful if there is a subterm of `T'` that matches the pattern `T` (that is, matching is allowed *anywhere* in the state term) and then the condition `C` is satisfied with the substitution for the variables obtained in the matching, and fails otherwise. `match T s.t. C` corresponds to matching only at the *top*. When the condition `C` is simply `true`, it can be omitted. `xmatch T s.t. C` performs matching at the top but *with extension* [7] modulo the attributes of the top operator of `T`.

3.5 Regular expressions

Basic strategies can be combined so that strategies are applied to execution paths. The first strategy combinators we consider are the typical regular expression constructions: concatenation, union, and iteration.

$$
\begin{aligned}
\langle \mathit{Strat} \rangle &::= \langle \mathit{Strat} \rangle \texttt{ ; } \langle \mathit{Strat} \rangle && \text{concatenation} \\
&\mid \langle \mathit{Strat} \rangle \texttt{ | } \langle \mathit{Strat} \rangle && \text{union} \\
&\mid \langle \mathit{Strat} \rangle \texttt{ *} && \text{iteration (0 or more)} \\
&\mid \langle \mathit{Strat} \rangle \texttt{ +} && \text{iteration (1 or more)}
\end{aligned}
$$

The concatenation operator is associative and the union operator is associative and conmutative. This commutativity of union provides a form of nondeterminism in the way the solutions are found.

If E, E' are strategy expressions and t is a term, then $(E \texttt{ ; } E') \mathbin{@} t = E' \mathbin{@} (E \mathbin{@} t)$, $(E \texttt{ | } E') \mathbin{@} t = (E \mathbin{@} t) \cup (E' \mathbin{@} t)$, and $E \texttt{+} \mathbin{@} t = \bigcup_{i \geq 1} E^i \mathbin{@} t$, where $E^1 = E$ and $E^n = (E \texttt{ ; } E^{n-1})$ for $n > 1$. Of course, $E \texttt{ *} = \texttt{idle | } E \texttt{ +}$. For example, a strategy of the form `E ; P` (with `P` a test) will filter out all those results from `E` that do not satisfy the test `P`.

3.6 If-then-else and its derived strategies

Our next strategy combinator is a typical if-then-else, but generalized so that the first argument is also a strategy. We have borrowed this idea from Stratego [27], but it also appears in ELAN [4, Example 5.2].

$$\langle Strat \rangle ::= \langle Strat \rangle \texttt{ ? } \langle Strat \rangle \texttt{ : } \langle Strat \rangle$$

The behavior of the strategy expression `E ? E' : E''` is as follows: in a given state term, the strategy `E` is evaluated; if `E` is successful, the strategy `E'` is evaluated in the resulting states, otherwise `E''` is evaluated in the *initial* state. That is, by definition, this combinator satisfies the equation

$$(E \texttt{ ? } E' \texttt{ : } E'') \mathbin{@} t = \textbf{if } (E \mathbin{@} t) \neq \emptyset \textbf{ then } E' \mathbin{@} (E \mathbin{@} t) \textbf{ else } E'' \mathbin{@} t \textbf{ fi}.$$

Note that, as mentioned above, in general the first argument is a strategy expression and not just a test. Since a test is a strategy, we have the particular case `P ? E' : E''` for a test `P` where evaluation coincides with the typical Boolean case distinction: `E'` is evaluated when the test `P` is true and `E''` when the test is false, taking into account that a test *qua* strategy fails when false.

Using the if-then-else combinator, we can define many other useful strategy combinators as derived operations. `E orelse E'` evaluates `E` in a given state; if such evaluation is successful, its results are the final ones, but if it fails, then `E'` is evaluated in the initial state.

$$\texttt{E orelse E'} = \texttt{E ? idle : E'}$$

`not(E)` reverses the result of evaluating `E`, so that `not(E)` fails when `E` is successful and vice versa.

$$\texttt{not(E)} = \texttt{E ? fail : idle}$$

An interesting use of `not(E)` is the following "normalization" (or "repeat until the end") operation `E !`:

$$\texttt{E !} = \texttt{E * ; not(E)}$$

`try(E)` evaluates `E` in a given state; if it is successful, the corresponding result is given, but if it fails, the initial state is returned.

$$\texttt{try(E)} = \texttt{E ? idle : idle}$$

Evaluation of `test(E)` checks the success/failure result of `E`, but it does not change the given initial state.

$$\texttt{test(E)} = \texttt{not(E) ? fail : idle}$$

Notice that `test(E)` = `not(not(E))`.

3.7 Rewriting of subterms

With the previous combinators we cannot force the application of a strategy to a specific subterm of the given initial term. In particular, the scope of the substitution in the `(a/x)match` combinators is only the corresponding condition. We can have

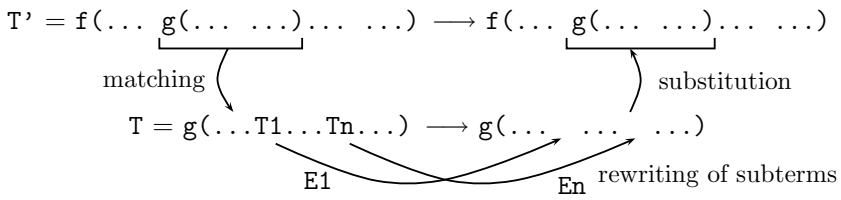

Fig. 1. Behavior of the `xmatchrew` combinator.

more control over the way different subterms of a given state are rewritten by means of the (`a`/`x`)`matchrew` combinators.

$$
\begin{aligned}
\langle \mathit{TermStratList} \rangle ::= {} & \langle \mathit{Term} \rangle \texttt{ using } \langle \mathit{Strat} \rangle \\
| \ & \langle \mathit{TermStratList} \rangle \texttt{ , } \langle \mathit{TermStratList} \rangle \\
\langle \mathit{Strat} \rangle ::= {} & \texttt{amatchrew } \langle \mathit{Pattern} \rangle \texttt{ [s.t. } \langle \mathit{EqCondition} \rangle \texttt{] by} \\
& \quad \langle \mathit{TermStratList} \rangle \\
| \ & \texttt{matchrew } \langle \mathit{Pattern} \rangle \texttt{ [s.t. } \langle \mathit{EqCondition} \rangle \texttt{] by} \\
& \quad \langle \mathit{TermStratList} \rangle \\
| \ & \texttt{xmatchrew } \langle \mathit{Pattern} \rangle \texttt{ [s.t. } \langle \mathit{EqCondition} \rangle \texttt{] by} \\
& \quad \langle \mathit{TermStratList} \rangle
\end{aligned}
$$

When the strategy expression

```
amatchrew T s.t. C by T1 using E1, ..., Tn using En
```

is applied to a state term `T'`, first a subterm of `T'` that matches `T` and satisfies `C` is selected. Then, the terms `T1`, ..., `Tn` (which must be disjoint subterms of `T`), instantiated appropriately, are rewritten as described by the strategy expressions `E1`, ..., `En`, respectively. The results are combined in `T` and then substituted in `T'`, in the way illustrated in Figure 1.

The strategy expressions `E1`, ..., `En` can make use of the variables instantiated in the matching, thus taking care of information extracted from the state term (see the example in Section 4.1).

The version `matchrew` works in the same way, but performing matching only at the top, while `xmatchrew` performs matching at the top with extension. In all cases, when the condition is `true` it can be omitted.

The *congruence operators* used in ELAN and Stratego [4,27] are special cases of the `matchrew` combinator, as shown in [20].

3.8 Recursion

Recursion is achieved by giving a name to a strategy expression and using this name in the strategy expression itself or in other related strategies. This is done in strategy modules (see Section 3.9). Concrete examples will be shown in Section 4.

3.9 Strategy modules and commands

Given a Maude system module `M`, the user can write one or more strategy modules to define strategies for `M`. Such strategy modules have the following form:

```
smod STRAT is
  protecting M .
  including STRAT1 .   ...    including STRATj .
  strat E1 : T11 ... T1m @ K1 .
  sd E1(P11,...,P1m) := Exp1 .
     ...
  strat En : Tn1 ... Tnp @ Kn .
  sd En(Pn1,...,Pnp) := Expn .
  csd En(Qn1,...,Qnp) := Expn' if C .
endsd
```

where `M` is the system module whose rewrites are being controlled, `STRAT1`, ..., `STRATp` are imported strategy submodules, `E1`, ..., `En` are identifiers, and `Exp1`, ..., `Expn` are strategy expressions (over the language of labels provided by `M`), where the identifiers can appear, thus allowing (mutually) recursive definitions.

The basic idea is that these strategy declarations provide useful abbreviations for strategy expressions `E` that the user can then utilize in a (strategy rewrite) command `srew T using E`, which rewrites a term `T` using a strategy expression `E`.

A strategy identifier can have *data arguments*, that are terms built with the syntax defined in the system module `M`. When a strategy identifier is declared (with the keyword `strat`), the types of its arguments (if any) are specified between the symbols `:` and `@`. After the symbol `@`, the type of the terms to which this strategy can be applied is also specified.

A strategy definition (introduced with the keyword `sd`) associates a strategy expression (on the righthand side of the symbol `:=`) with a strategy identifier (on the lefthand side) with patterns as arguments, used to capture the values passed when the strategy is invoked. These strategy definitions can be conditional (with keyword `csd`). There may be several definitions for the same strategy identifier but they should refer to disjoint cases of the arguments, due either to the usage of different constructors in the patterns or to the conditions used.

A strategy module can be *parametric*. As for other parameterized Maude modules [7], the requirements that a concrete parameter must fulfill are specified in a theory, called in this case a *strategy theory*. In strategy theories, syntax for the terms (sorts and operators) and strategy identifiers can be declared. For example, the following strategy theory requires a sort `State`, two operators from `State` to `Bool`, and a strategy identifier `expand` without arguments and applicable to `State` terms.

```
sth BT-ELEMS is
  protecting BOOL .
  sort State .
  op isOk : State -> Bool .
  op isSolution : State -> Bool .
  strat expand @ State .
```

```
endsth
```

A *parameterized* strategy module, say `STRAT`, can have several strategy theories `P1`, ..., `Pn` as formal parameters. Its header will then be declared with syntax `STRAT{X1 :: P1, ..., Xn :: Pn}`. A parameterized strategy module can then be *instantiated* with actual parameters that are strategy modules `S1`, ..., `Sn` that satisfy, respectively, the requirements in the corresponding strategy theories `P1`, ..., `Pn`. As done in Maude for instantiating any other kind of parameterized module [7], the binding of each formal parameter `Pi` to its corresponding actual parameter `Si` is specified by a *view*, that is, by a theory interpretation that maps sorts in the parameter theory `Pi` to sorts in the strategy module `Si`, and, likewise, maps operators to operators or terms, and strategy identifiers to strategy expressions. In Section 4.3 we show a complete example of a parameterized strategy module.

4 Some Examples

In this section we show some examples to illustrate the use of the strategy language.

4.1 Blackboard

The first example is a simple game. We have a blackboard on which several natural numbers have been written. A legal move consists in selecting two numbers in the blackboard, removing them, and writing their arithmetic mean. The objective of the game is to get the greatest possible number written on the blackboard at the end. The specification of the game in Maude is also quite simple.

```
mod BLACKBOARD is
  protecting NAT .
  sort Blackboard .
  subsort Nat < Blackboard .
  op __ : Blackboard Blackboard -> Blackboard [assoc comm] .
  vars M N : Nat .
  rl [play] : M N => (M + N) quo 2 .
endm
```

A player can choose the numbers randomly, or can follow some strategy. Possible strategies consist in taking always the two greatest numbers, or the two smallest, or taking the greatest and the smallest. The following `EXT-BLACKBOARD` module extends the `BLACKBOARD` module with operations to get the maximum and the minimum numbers in a blackboard, and for removing an element in the blackboard.

```
mod EXT-BLACKBOARD is
  including NAT .
  including BLACKBOARD .
  ops max min : Blackboard -> Nat .
  op remove : Nat Blackboard -> Blackboard .
  vars M N X Y : Nat .
  var B : Blackboard .
  eq max(N) = N .
  eq max(N B) = if N > max(B) then N else max(B) fi .
```

$Y := 2$
while $Y \leq N$ do
  $X := Y$
  while $X > 1 \ \wedge \ V[X-1] > V[X]$ do
    switch $V[X-1]$ and $V[X]$
    $X := X - 1$
  $Y := Y + 1$

Fig. 2. Insertion sort.

```
  eq min(N) = N .
  eq min(N B) = if N < min(B) then N else min(B) fi .
  eq remove(X, X B) = B .
endm
```

The strategy module `BLACKBOARD-STRAT` below defines the three mentioned strategies. Note how the `matchrew` strategy constructor is used to get information about the state term that is then used in the definition of how the rule `play` should be applied.

```
smod BLACKBOARD-STRAT is
  protecting EXT-BLACKBOARD .
  var B : Blackboard .
  vars X Y : Nat .
  strat maxmin @ Blackboard .
  sd maxmin := (matchrew B s.t. X := max(B) /\ Y := min(B) by
                 B using play[M <- X ; N <- Y] ) ! .
  strat maxmax @ Blackboard .
  sd maxmax := (matchrew B s.t. X := max(B) /\ Y := max(remove(X,B)) by
                 B using play[M <- X ; N <- Y] ) ! .
  strat minmin @ Blackboard .
  sd minmin := (matchrew B s.t. X := min(B) /\ Y := min(remove(X,B)) by
                 B using play[M <- X ; N <- Y] ) ! .
endsm
```

```
Maude> srew 2000 20 2 200 10 50 using maxmin .
result NzNat :  178
Maude> srew 2000 20 2 200 10 50 using maxmax .
result NzNat :  77
Maude> srew 2000 20 2 200 10 50 using minmin .
result NzNat :  1057
```

4.2 Insertion sort

In this section we present a strategy that implements the insertion sort algorithm. The imperative pseudocode for this algorithm is shown in Figure 2 (for sorting an array $V[1..N]$). The algorithm keeps two indices, one pointing to the next element to be inserted between the already sorted elements, and another pointing to the element which is being inserted.

First we define a module `SORTING` (shown in Section 2) that specifies arrays as sets of pairs and a rule to switch the values in two positions of the array.

The following strategy module defines the strategies `insort` and `insert` that

rewrite terms of sort `PairSet` and represent the loops in the algorithm in a recursive way. Both strategies have a natural number as data argument. They represent the indices used by the algorithm. The expression $X - 1$ is represented as `sd(X, 1)`, where `sd` is the predefined symmetric difference operation in the `NAT` module. Notice that the conjunction in the inner loop is separated in two conditions, and that the last strategy definition is conditional, with condition `X > 1`.

```
smod INSERTION-SORT-STRAT is
  protecting SORTING .
  var PS : PairSet .
  vars X Y V W : Nat .
  strat insort : Nat @ PairSet .
  sd insort(Y) := try(match PS s.t. Y <= length(PS) ;
                      insert(Y) ;
                      insort(Y + 1)) .
  strat insert : Nat @ PairSet .
  sd insert(1) := idle .
  csd insert(X) := try(amatch (sd(X,1), V) (X, W) s.t. V > W ;
                       switch[J <- sd(X,1) ; I <- X] ;
                       insert(sd(X,1)))
                   if X > 1 .
endsm
```

```
Maude> srew (1, 18) (2, 14) (3, 11) (4, 15) (5, 12) using insort(2) .
result PairSet : (1, 11) (2, 12) (3, 14) (4, 15) (5, 18)
```

Sometimes a strategy needs to remember some information about what it has already done in order to know what it has to do next. In the above insertion sort strategy, this information is maintained in the arguments of the strategies `insort` and `insert`. But depending on the way this information has to be modified, sometimes this "memory" for keeping auxiliary information can be maintained as part of the term being rewritten. In [20] we showed a different way of implementing the insertion sort algorithm using this "memory-based" approach.

4.3 Backtracking

In this section we show a *parametric* strategy useful for solving a problem using backtracking. It requires that (partial) solutions are represented as terms of a sort `State`, and that there are predicates `isOk`, to check if a partial solution is extensible to a complete solution, and `isSolution`, to check if we already have a solution. It also assumes a strategy `expand` that extends a partial solution. These parametric requirements are collected in the strategy theory `BT-ELEMS` shown in Section 3.9. With these ingredients we can then define a generic strategy `solve` that defines how a problem has to be solved by means of backtracking.

```
smod BT-STRAT{X :: BT-ELEMS} is
  var S : X$State .
  strat solve @ X$State .
  sd solve := (match S s.t. isSolution(S)) ? idle
                                           : (expand ;
                                              match S s.t. isOk(S) ;
                                              solve) .
```

```
endsm
```

This strategy first checks if it has already obtained a solution. If this is the case, it finishes. Otherwise, it applies the strategy `expand`, checks if the extension is ok, and continues recursively.

We can use this strategy to find a way out of a labyrinth. The labyrinth is built on an 8×8 grid where some cells have a wall that cannot be crossed. The entry is on the top-left position $[1, 1]$ and the exit to be found is at the bottom-right position $[8, 8]$. Solutions are represented as lists of positions that indicate the path to be followed; a list is a solution if it finishes in the exit; and a partial solution satisfies the predicate `isOk` if its final position is on the grid without wall, and the list does not contain duplications. The following module `LABYRINTH` defines these elements, where we use the predefined module `LIST`, which is parameterized with respect to the theory `TRIV`.

```
fmod POSITIONS is
  protecting NAT .
  sort Pos .
  op [_,_] : Nat Nat -> Pos .
endfm

view Pos from TRIV to POSITIONS is
  sort Elt to Pos .
endv

mod LABYRINTH is
  including LIST{Pos} .
  op contains : List{Pos} Pos -> Bool .
  ops isSolution isOk : List{Pos} -> Bool .
  op next : List{Pos} -> Pos .
  op wall : -> List{Pos} .
  vars X Y : Nat .
  var P Q : Pos .
  var L : List{Pos} .
  eq wall = *** the wall has been omitted *** .
  eq isSolution(L [8,8]) = true .
  eq isSolution(L) = false [owise] .
  eq contains(nil, P) = false .
  eq contains(Q L, P) = if P == Q then true else contains(L, P) fi .
  eq isOk(L [X,Y]) = X >= 1 and Y >= 1 and X <= 8 and Y <= 8
                     and not(contains(wall, [X,Y])) and
                         not(contains(L, [X,Y])) .
  crl [extend] : L => L P if next(L) => P .
  rl [next] : next(L [X,Y]) => [X + 1, Y] .
  rl [next] : next(L [X,Y]) => [X, Y + 1] .
  rl [next] : next(L [X,Y]) => [sd(X, 1), Y] .
  rl [next] : next(L [X,Y]) => [X, sd(Y, 1)] .
endm
```

A nonempty list is extended by means of the rewrite rule `extend`, that appends any of the positions adjacent to the last one. This rule has to be applied to the top of the list representing the current partial solution, and using the rules `next` to

solve the condition. This is precisely what the following strategy `expand` specifies.

```
smod LABYRINTH-STRAT is
  protecting LABYRINTH .
  strat expand @ List{Pos} .
  sd expand := top(extend{next}) .
endsm
```

The following view indicates the source theory, the target strategy module, and the mapping of each sort, operator or strategy identifier in the source theory. Unmentioned elements get the identity mapping, like the `isOk` and `isSolution` operations.

```
sview LABYRINTH-BT-ELEM from BT-ELEMS to LABYRINTH-STRAT is
  sort State to List{Pos} .
endsv

smod LABYRINTH-BT-STRAT is
  including BT-STRAT{LABYRINTH-BT-ELEM} .
endsm
```

4.4 Abstract congruence closure

Congruence closure is a decision procedure for the word problem associated with a finite set of ground equations. Since it can be used to answer the satisfiability of a conjunction of equalities and inequalities, it also provides a decision procedure for the theory of uninterpreted function symbols. We present the *abstract* formulation of congruence closure given by Tiwari, which allows a high-level proof of its correctness [25,2]. We then show how the inference rules have natural and very direct representations as rewrite rules that can be executed in Maude. Following Tiwari, we also show how Shostak's [24] and Downey-Sethi-Tarjan's [13] congruence closure *algorithms* can be specified as *strategies* in our strategy language. In this way we obtain prototypes for these algorithms in Maude in a straighforward manner. Note that, since the correctness of congruence closure only depends on that of the inference rules [25,2], and the strategy language only allows rewrites that are correct in the rewrite theory being controlled, the analysis of these algorithms can be reduced to control and efficiency issues: they can *never* perform an incorrect inference. Of course, they can still be analyzed with respect to completeness.

Tiwari's congruence closure inference rules are as follows, where K is a set of

constants, E is a set of equations, and R is a set of oriented equations.

Extension	$\dfrac{(K, E[t], R)}{(K \cup \{c\}, E[c], R \cup \{t \to c\})}$	if $t \to c$ is a D-rule
Simplification	$\dfrac{(K, E[t], R \cup \{t \to c\})}{(K, E[c], R \cup \{t \to c\})}$	
Orientation	$\dfrac{(K \cup \{c\}, E \cup \{t \approx c\}, R)}{(K \cup \{c\}, E, R \cup \{t \to c\})}$	if $t \succ c$
Deletion	$\dfrac{(K, E \cup \{t \approx t\}, R)}{(K, E, R)}$	
Deduction	$\dfrac{(K, E, R \cup \{t \to c, t \to d\}}{(K, E \cup \{c \approx d\}, R \cup \{t \to d\})}$	
Collapse	$\dfrac{(K, E, R \cup \{s[t] \to d, t \to c\}}{(K, E, R \cup \{s[c] \to d, t \to c\}}$	if $s \not\equiv t$

Their representation as rewrite rules in Maude follows very closely the above abstract formulation, with the exception that the set K of constants is represented as a counter to generate fresh constants. Some auxiliary functions, assumed in the inference rules and having a straightforward equational definition, are used. These auxiliary definitions are omitted for the sake of brevity.

```
crl [Ext] : < K, E u = v, R >
         => < K + 1, E u' = v, R t -> c >
         if subterm(u) => t                *** u[t]
         /\ isOpConstants?(t)              *** t -> c is a D-rule
         /\ c := c(K)                      *** new constant
         /\ u' := subst(u,t,c) .

crl [Sim] : < K, E u = v, R t -> c >
         => < K, E u' = v, R t -> c >
         if subterm(u) => t
         /\ u' := subst(u,t,c) .

crl [Ori] : < K, E t = c, R >
         => < K, E, R t -> c >
         if t > c .

rl [Del] : < K, E t = t, R >
         => < K, E, R > .

crl [Ded] : < K, E, R t -> c t -> d >
         => < K, E, R c -> d t -> d >
         if c > d .

crl [Col] : < K, E, R u -> d t -> c >
```

```
=> < K, E, R u' -> d t -> c >
if subterm(u) => t
/\ t =/= u                    *** proper subterm
/\ u' := subst(u,t,c) .
```

Shostak's [24] and Downey-Sethi-Tarjan's [13] congruence closure algorithms can be viewed as different strategies to apply the same congruence closure inference rules [25,2].

Tiwari et al.'s specification of Shostak's and Downey-Sethi-Tarjan's algorithms [25,2] is as strategies defined in terms of *regular expressions*. However, since this formulation relies on a general understanding of regular expressions, and not in a precise semantics for regular expressions as strategies, the regular expression specifications given are in some sense *ambiguous*. The source of the ambiguity resides in the lack of a notation to distinguish between a strategy E^*, which repeatedly applies E zero, one, or more times, and a strategy $E! = E^*; not(E)$, which repeatedly applies E "to the bitter end" (see Sections 3.5 and 3.6).

Their specifications of Shostak's and Downey-Sethi-Tarjan's algorithms, and their corresponding, more precise specification in our strategy language are as follows:

- Shostak's algorithm [24]:

$$\mathbf{Shos} = ((\mathbf{Sim}^* \circ \mathbf{Ext}^*)^* \circ (\mathbf{Del} \cup \mathbf{Ori}) \circ (\mathbf{Col} \circ \mathbf{Ded}^*)^*)^*$$

```
sd Shos1 := (test(Sim | Ext) ; Sim ! ; Ext !)! ;
            try( Del | Ori ) ;
            (test(Col | Ded) ; try(Col) ; Ded !)!  .

sd Shos := matchrew S:State by
             S:State using Shos1 ;
                           (match S:State orelse Shos) .
```

- Downey-Sethi-Tarjan algorithm [13]:

$$\mathbf{DST} = ((\mathbf{Col} \circ (\mathbf{Ded} \cup \{\epsilon\}))^* \circ (\mathbf{Sim}^* \circ (\mathbf{Del} \cup \mathbf{Ori}))^*)^*$$

```
sd start := (Ext ; Sim !)! ; Ori ! .

sd DST1 := (test(Col | Ded) ; try(Col) ; try(Ded))! ;
           ( Sim ! ; (Del | Ori))! .

sd DST = matchrew S:State by
           S:State using DST1 ;
                         (match S:State orelse DST) .
```

We can now, for example, solve a congruence closure problem in Maude using the abstract congruence closure rules as controlled by the `DST` strategy:

```
Maude> srew < 0, 'a.S = 'b.S
              'f['f['a.S]] = 'f['b.S], mtRlS >
       using start ; DST .

result State :
```

```
< 5, mtEqS, 'a.S -> c(0)
            'b.S -> c(1)
            c(0) -> c(1)
            c(2) -> c(4)
            c(3) -> c(4)
            'f[c(1)] -> c(3)
            'f[c(4)] -> c(4) >
```

5 Implementation

The first proposal of the language was implemented as a prototype by using the Maude metalevel features (Section 5.1). This prototype has been relatively easy to develop and very useful for experimental purposes to reach a definitive strategy language design. Currently, the enhanced strategy language is being implemented in C++, so that it can be integrated with the rest of the Maude system (Section 5.2).

5.1 Reflective prototype implementation

Using the Maude metalevel, we implemented a prototype of the strategy language as an extension of Full Maude [7, Part II]. It consists of several functions that work with a labelled version of the conceptual computation tree produced when applying a strategy `E` to a given state term `T`. Nodes in this tree are tuples formed by a term, a strategy, and possibly other information (used to know which is the next child to be explored; the specific information saved depends on the top constructor in the strategy expression). The root is `< T, E >`, and the children of a node `< T', E' >` are the terms obtained from `T'` by rewriting as described by `E'`, paired with the corresponding remainder of `E'`. In a successful path, the strategy at the leaf node is empty, meaning that nothing is left to do, which corresponds to a complete successful application of the strategy `E`.

The two main functions are `first` and `next`. The combination of these two functions serves to find all the solutions for the application of a strategy to a given state term. They are implemented in a mutually recursive way, distinguishing cases on the strategy expression in the last node of the given path, and with the help of the metalevel descent functions `metaApply`, `metaXapply`, `metaMatch`, and `metaXmatch` [7, Section 10.4].

The metalanguage features of Maude allow completing the prototype with a user interface where strategy modules can be loaded, and commands to rewrite a term using a strategy can be executed. These commands allow a step-by-step generation of all the possible results of rewriting a term using a strategy.

5.2 Maude system implementation

The implementation of the Maude strategy language is challenging: We need to perform a search taking steps in the rewrite graph that correspond to labels allowed by the strategy language, and there may be various interactions between the two due to substitutions, pattern matching, subexpressions, and substrategies for condi-

tions. Without rewrite conditions, the finitary nature of matching in the supported theories ensures that the rewrite graph is finitely branching though it may have infinite depth (non-terminating sequences of rewrites); with rewrite conditions we do not even have the finite branching property.

The strategy language may be considered to have finite branching but most useful strategy expressions will describe rewriting sequences of unbounded depth.

We thus have a situation where our search tree is in general both infinitely branching and infinitely deep.

The best we can do under such circumstances is to support *fairness*, that is, if t can rewrite to t' using strategy s, then solution t' will be found in a finite (but potentially impractical in terms of both time and space) computation.

5.2.1 Processes

In attempting to achieve fairness we view the growing search tree as a pool of *processes*, each with a subject term to rewrite and a stack of strategy expressions to use. The stack of strategy expressions essentially corresponds to a concatenation of strategies and arises naturally, both from the possibility of recursively defined strategies and the accumulation of "pending" strategies during the decomposition of more complex strategy expressions.

The processes exist on a circular double linked list. Processes can pop out of existence at any time and new processes can be created, just ahead of the currently running process. This location is effectively the back of the queue with respect to the running process and ensures that unbounded expansion of the process queue cannot cause starvation. Each process, when it runs, does a small amount of computation to advance its piece of the search.

The two most important kinds of processes in the Maude implementation are decomposition processes and application processes. The goal of a decomposition process is to decompose the strategy expression on the top of its stack into more decomposition processes in the case of a strategy operator, or into an application process in the case of a rule label. The goal of an application process is to search for one step rewrites from the subject term t, to a new term t', that are consistent with a given label. For each such t', an application process forks off a new decomposition process to rewrite t' using the remaining strategy expression stack. At each step, a decomposition process pops a strategy expression s from its stack and takes some action, based on the top operator of s. Some sample actions are shown in Table 1. A decomposition process succeeds when its stack becomes empty.

5.2.2 Tasks

It turns out that round-robin driven processes are not sufficient. Sometimes we want to treat a chunk of the search tree as an entity in its own right—as a subsearch. One case where this occurs is the s_1 `?` s_2 `:` s_3 operator. Here we are interested to know when all attempts to rewrite with s_1 on a given term t fail, so we can run s_3 on t. Another situation where it is advantageous to have subsearches is the `not(`s`)` and `test(`s`)` operators. Once we have a single success for rewriting with s on a given

Popped strategy expression	Action
`idle`	Running process yields.
`fail`	Running process terminates.
l (label)	Create application process for label l, with same term and strategy stack as running process. Running process terminates.
s_1 `;` s_2	Push s_2, followed by s_1 on the stack of the running process.
s_1 `|` s_2	Clone running process; push s_1 on the clone's stack and s_2 on the original process stack.
s `*`	Clone running process, push s `*` followed by s on the stack of the running process.
s `+`	Push s `*` followed by s on the stack of the running process.
d (defined strategy)	Push e on the stack of the running process, where e is the strategy expression defining d.

Table 1
Sample decomposition actions

term t, we can kill the entire subsearch as an efficiency measure. The management of subsearches is done by *tasks*, which are an event driven fan-in counterpart to the round-robin driven fan-out processes. An `srew` command generates the root task.

Each process and each task (except the root task) will belong to some task and will live on a double linked list belonging to that task. When a process p creates a new process or task, that process or task belongs to p's owner. When a task t creates a new process or task, t gets to choose whether the new entity belongs to t or to t's owner.

When a process or task succeeds, it unlinks itself from its owners list and informs the owning task of the success. When a process or task terminates it likewise unlinks itself from its owners list. In both cases, it checks if it was the last process or task belonging to its owner. In that case it informs its owner that subsearch is exhausted.

After root tasks, the most important kind of task in the Maude implementation is the branch task. Like a process, a branch task has a term t to rewrite and a stack st of strategy expressions for continuation. A branch task also has an initial strategy s and possibly other information depending on what options are used. A branch task forks off a decomposition process to handle rewriting t with s and waits for successes, or for the subsearch to end in failure. Possible actions in the failure case are termination, and creating a new decomposition process using t and st, after optionally pushing a new strategy expression s_2 on st. For each success t' there is also the possibilities of creating a new decomposition process using t'

and st, after optionally pushing a new strategy expression s_1 on st, and creating a new branch task to repeat the original search with s on t'. Thus branch tasks generalize a number of strategy language operators including s_1 `?` s_2 `:` s_3, s `!`, `not(`s`)` and `test(`s`)`.

5.2.3 Efficiency considerations

Stacks of strategy expressions play a key rôle in both processes and tasks. To avoid repeated copying of these structures we can use a persistent stack with new entries being created via hash consing to avoid duplication and to give us a fast check for equality between two stacks.

When we create a decomposition process for some task, we can check via a hash table in that task whether we have already created a decomposition process for that task with the same subject term and strategy expression stack. If so, creating the new decomposition process would be redundant.

6 Conclusions

In this paper we have put forward three key ideas:

(i) Rewriting logic is a logical framework to represent logics and inference systems.

(ii) Automated deduction methods should be specified by:
- inference systems, and
- different strategies to apply inference rules.

(iii) Automated deduction systems can be prototyped/implemented at a very high level in a rewriting logic language having a strategy language.

We have also presented a specific strategy language for Maude, illustrating its use in automated deduction. We have tried to achieve a relatively simple, yet expressive, strategy language design and have validated experimentally our design using various automated deduction and programming language semantics applications. However, the particular strategy language design of choice is not essential: our general points apply as well to other rewriting languages. Therefore, the essential point is that the vision put forward in points (i)–(iii) above can be effectively carried out not just in Maude, but in any rewriting language with adequate support for strategies.

An interesting future research direction is investigating *distributed specifications and implementations of a strategy language*, since this offers the promise of more efficient and scalable executions, also in automated deduction.

Acknowledgements

We thank the organizers of *Strategies 2006* for their invitation to present this work in such a nice environment, and the referees for their helpful comments.

References

[1] L. Bachmair and N. Dershowitz. Completion for rewriting modulo a congruence. *Theor. Comput. Sci.*, 67(2&3):173–201, 1989.

[2] L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, 31(2):129–168, 2003.

[3] D. Basin, M. Clavel, and J. Meseguer. Reflective metalogical frameworks. *ACM Transactions on Computational Logic*, 5(3):528–576, 2004.

[4] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: A functional semantics. *International Journal of Foundations of Computer Science*, 12:69–95, 2001.

[5] M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications.* CSLI Publications, Stanford University, 2000.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.2)*, December 2005. http://maude.cs.uiuc.edu/maude2-manual.

[8] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *CAFE: An Industrial-Strength Algebraic Formal Method*. Elsevier, 2000. http://maude.cs.uiuc.edu.

[9] M. Clavel and J. Meseguer. Internal strategies in a reflective logic. In *Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction (Townsville, Australia, July 1997)*, pages 1–12, 1997.

[10] M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245–288, 2002.

[11] M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science*, 2006. To appear.

[12] S. Conchon and S. Krstic. Strategies for combining decision procedures. In *Proceedings of the 9th Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *Lecture Notes for Computer Science*, pages 537–553. Springer, 2003.

[13] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpressions problem. *Journal of the ACM*, 27(4):758–771, 1980.

[14] M. Hidalgo-Herrero, A. Verdejo, and Y. Ortega-Mallén. Using Maude and its strategies for defining a framework for analyzing Eden semantics. In S. Antoy, editor, *The Sixth International Workshop on Reduction Strategies in Rewriting and Programming, WRS'06*, Electronic Notes in Theoretical Computer Science. Elsevier. To appear.

[15] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 257–321. MIT Press, 1991.

[16] D. Kapur. Shostak's congruence closure as completion. In H. Comon, editor, *8th International Conference on Rewriting Techniques and Applications, RTA'97*, volume 1232 of *Lecture Notes for Computer Science*, pages 23–37. Springer, 1997.

[17] P. Lescanne. Completion procedures as transition rules + control. In J. Díaz and F. Orejas, editors, *TAPSOFT'89 Proceedings of the International Joint Conference on Theory and Practice of Software Development, Barcelona, Spain, March 13-17, 1989*, volume 351 of *Lecture Notes in Computer Science*, pages 28–41. Springer, 1989.

[18] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[19] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Second Edition, Volume 9*, pages 1–87. Kluwer Academic Publishers, 2002. First published as SRI Technical Report SRI-CSL-93-05, August 1993.

[20] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In N. Martí-Oliet, editor, *Proceedings Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27 – April 4, 2004*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 417–441. Elsevier, 2005.

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

[22] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and abstract DPLL modulo theories. In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'04), Montevideo, Uruguay*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2005.

[23] F. Rosa-Velardo, C. Segura, and A. Verdejo. Typed mobile ambients in Maude. In H. Cirstea and N. Martí-Oliet, editors, *Proceedings of the 6th International Workshop on Rule-Based Programming (RULE 2005)*, volume 147 of *Electronic Notes in Theoretical Computer Science*, pages 135–161. Elsevier, 2006.

[24] R. E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, July 1978.

[25] A. Tiwari. *Decision Procedures in Automated Deduction*. PhD thesis, State University of New York, 2000.

[26] A. Verdejo and N. Martí-Oliet. Basic completion by means of Maude strategies. Paper in preparation, 2006.

[27] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer, editor, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes for Computer Science*, pages 216–238. Springer, 2004.