Internal Strategies in a Reflective Logic*

Manuel Clavel and José Meseguer

Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA

Abstract. After introducing the basic notions of reflective logic and internal strategies, we discuss in detail how reflection can be systematically exploited to design a strategy language internal to a reflective logic in the concrete case of rewriting logic and Maude; and we illustrate the advantages of this new approach to strategies by showing how the rules of inference for Knuth-Bendix completion can be given strategies corresponding to completion procedures in a completely modular way, not requiring any change whatsoever to the inference rules themselves.

1 Introduction

This paper is part of a long-term effort to study and axiomatize reflective logics and to systematically exploit reflection to bring deduction strategies within the fold of logic. In this way, more powerful, modular, and flexible notions of deduction strategy can be defined and verified. In earlier work [7, 8] we have proposed general axiomatic notions of reflective logic and internal strategy language, and have begun to show how they can be realized in a concrete language based on reflective rewriting logic like Maude [6]. Here, we discuss in greater detail how reflection can be systematically exploited to design a strategy language internal to a reflective logic in the case of rewriting logic and Maude; and we illustrate the advantages of this new approach to strategies by showing how the rules of inference for Knuth-Bendix completion can be given strategies corresponding to completion procedures in a completely modular way, not requiring any change whatsoever to the inference rules themselves.

A reflective approach to strategies seems entirely natural, and, as explained later, has also been adopted by other authors in the context of higher-order logics. Since strategies naturally generalize proofs—in the sense that they yield sets of proofs as their results—in a reflective logic they can be *reified*, that is, they can be brought down to the object level the same way that other metalevel entities such as proofs and theories can. Furthermore, their semantics can be *internalized* in the logic in question, that is, they can be given a declarative semantics within the same logic that they control, in such a way that their execution becomes logical deduction. It then becomes possible to reason formally about such internal

* Supported by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114, National Science Foundation Grant CCR-9633363, and by the Advanced Software Enrichment Project of the Information-Technology Promotion Agency, Japan (IPA).

strategies using all the formal methods available in the logic. Internal strategies replace the traditional "Logic + Control" treatment of strategies as extralogical constructs by the inclusion "Control $\subseteq$ Logic" of strategies as logical entities.

1.1 Reflective Logics and Internal Strategies in a Nutshell

Given a logic, we say that it is *reflective* relative to a class $\mathcal{C}$ of theories—where typically the class $\mathcal{C}$ corresponds to theories that are describable using a finite piece of language, such as finitely presented or recursively enumerable theories—if we can find inside $\mathcal{C}$ a *universal theory* $\mathcal{U}$ that can simulate all other theories in the class, in the sense that—denoting by $sen(T)$ the set of meaningful sentences in the language of a theory T—there is a function, called a *representation function*, $\overline{(\_ \vdash \_)} : \bigcup_{T \in \mathcal{C}} \{T\} \times sen(T) \longrightarrow sen(\mathcal{U})$ such that for each $T \in \mathcal{C}$ and $\varphi \in sen(T)$, $T \vdash \varphi \iff \mathcal{U} \vdash \overline{T \vdash \varphi}$. Since $\mathcal{U}$ itself is representable, representation can be iterated, so that we immediately have a 'reflective tower' $T \vdash \varphi \iff \mathcal{U} \vdash \overline{T \vdash \varphi} \iff \mathcal{U} \vdash \overline{\mathcal{U} \vdash \overline{T \vdash \varphi}} \ \ldots$ This definition of reflective logic assumes general axiomatic notions of logic and entailment such as those provided by the theory of general logics [18]. This is the approach taken in [7, 8] to make the above definition fully precise.

Given a logical theory T in a logic, a *strategy* is a computational way of looking for certain proofs of some theorems of T (see [8] for a general axiomatization of deductive strategies). This may be done by having a strategy language $S(T)$ associated with T in which strategies controlling deduction in T can be defined. For example, in theorem provers in the LCF tradition, there is a metalanguage, namely ML, in which "tactics" and "tacticals" can be defined for exactly this purpose [23]. Even if quite flexible strategies can thus be defined, such a language may remain a programming language external to the logic that it is controlling. In this situation, control becomes an extralogical feature of the system.

If strategies can be defined *inside* the logic that they control, we are in a much better situation, since formal reasoning within the system can be applied to the strategies themselves. Basically, an *internal strategy language* is a theory-transforming function S that sends each theory T to another theory $S(T)$ in the same logic whose deductions simulate controlled deductions of T. Reflective logics offer good opportunities for defining internal strategy languages of this kind, because the metalevel being controlled can be expressed within the logic. In [8] we gave a general method for defining and proving correct an internal strategy language, and showed how the correctness of a universal theory greatly simplifies the proof of correctness of a given strategy language. We have used the Maude rewriting logic language [6] as an experimental vehicle for the exploration of these techniques. Similar ideas about specifying internal strategies are also being adopted for the strategies of the ELAN rewriting logic language [4], whose strategy language was initially defined as an external language. However, in ELAN reflection and strategies have not yet been fully integrated. In the context of typed lambda calculi, the important advantages of having an internal strategy language have also been stressed by several authors [1, 9, 25].

1.2 Rewriting Logic and Maude

A rewrite theory T consists of a signature Σ of operators, a set E of equations, and a set of labeled rewrite rules. The deductions of T are rewrites modulo E using such rules. Such rewriting deductions are formalized as proofs using four simple inference rules [19].

Rewriting logic is a reflective logic[2] in the sense we have just explained [8]. Since the sentences meaningful in the language of a rewrite theory T are rewrite sequents $t \longrightarrow t'$ where t, t' are Σ-terms, the general notion of reflection takes in this case the following form. The class $\mathcal{C}$ is that of finitely presentable rewrite theories—that is, theories whose signature, and sets of equations and rules are all finite. There is then a finitely presentable theory $\mathcal{U}$ and a representation function $\overline{(\_ \vdash \_)}$ which encodes a pair consisting of a rewrite theory T in $\mathcal{C}$ and a sentence $t \longrightarrow t'$ in T as a sentence $\langle \overline{T}, \overline{t} \rangle \longrightarrow \langle \overline{T}, \overline{t'} \rangle$ in $\mathcal{U}$, in such a way that $T \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle \overline{T}, \overline{t} \rangle \longrightarrow \langle \overline{T}, \overline{t'} \rangle$, where the function $\overline{(\_)}$ recursively defines the representation of rules, terms, etc. as terms in $\mathcal{U}$.

Maude is a logical language based on rewriting logic [19, 22, 20]. An introduction to Maude and its interpreter implementation can be found in [6]. For our present purposes the key point is that the Maude implementation supports an arbitrary number of levels of reflection and gives the user access to important reflective capabilities, including the possibility of defining and using internal strategy languages. In [8, 6] we explained how such languages can be defined, and introduced a simple but interesting internal strategy language. Here we define another such strategy language, and use it to show how reflection can give a fully modular and declarative executable specification of completion procedures.

1.3 Reflection and Internal Strategies for Completion

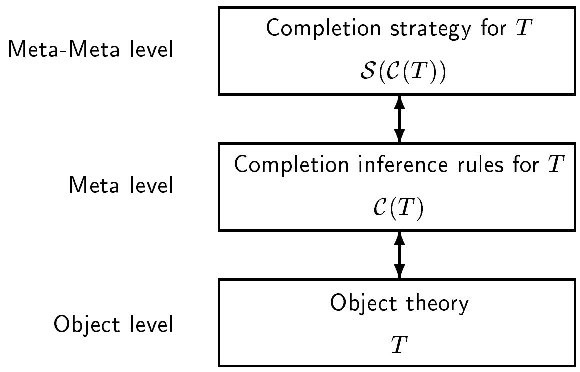
As it is well known, the Knuth-Bendix completion algorithm [14] attempts to construct a convergent rewrite system from a finite axiomatization of an equational theory. An abstract view of completion as an inference system $\mathcal{C}$ was proposed by Bachmair, Dershowitz and Hsiang [2]. In this view, a (*standard*) *completion procedure* is any program that takes a set of equations and rules, $(E_0 \cup R_0)$, and a reduction ordering $\succ$ containing R_0, and uses some strategy to apply the inference rules of $\mathcal{C}$ to generate a derivation from $(E_0 \cup R_0)$. A completion procedure is called correct if all non-failing derivations result in a convergent system.

By proving correctness of this more abstract notion of completion, Bachmair, Dershowitz and Hsiang were able to guarantee the correctness of a wide class of different completion procedures. A key element of their methodology is to cleanly separate the logical, or proof-theoretical, component of the completion method from issues pertaining to strategy and control. This is precisely what reflection and a convenient internal strategy language allow us to accomplish.

[2] The proof in [8] treats the case of unsorted and unconditional rewrite theories; the extension to sorted and conditional theories is unproblematic and will appear elsewhere.

There have been several attempts to use this abstract view of completion in actual rewrite-rule implementations of completion procedures [15, 13]. The challenge has been to control the rewriting inference process generated by the rules in $\mathcal{C}$—which in principle could go in many undesired directions—by means of adequate *strategies*. Unfortunately, these implementations have not been able to preserve the separation between the logical and the control components of the completion method. For example, control of the rewriting process is deeply based on the data structure on which the inference rules operate. As a consequence, improvements in the control of the rewriting process required changes in the original data structure and consequently in the inference system $\mathcal{C}$ as shown in [15]. In particular, proofs of correctness based on the original inference system need to be redone for the new inference rules combining inference and control.

We believe that the difficulties encountered in previous approaches are quite intrinsic and probably cannot be overcome without the use of new techniques such as reflection. Essentially, the possibility of allowing several levels of reflection is the key to a fully modular approach in which no change whatsoever is required in the original inference rules. This clean separation by levels is illustrated in the figure below. The theory T that we want to complete appears at the object level. The inference rules $\mathcal{C}$ —parameterized by T— are clearly metalevel rules defining a theory one level up. The key point is that the strategies controlling the application of such inference rules and therefore yielding different completion procedures appear as terms in the theory $\mathcal{S}(\mathcal{C}(T))$, corresponding to instantiating an internal strategy language $\mathcal{S}$ with the inference rules $\mathcal{C}$ as its object theory. Since $\mathcal{S}$ transforms any theory into a metalevel theory, and $\mathcal{C}$ is already a metalevel theory, $\mathcal{S}(\mathcal{C}(T))$ is a theory in the meta-metalevel. Keeping these three levels of reflection separate is the key to maintaining modularity. Of course, in other applications, involving for example meta-strategies that control the application of lower-level strategies, more than three levels of reflection may in fact be required.



2 A Reflective Kernel in Maude

It is possible to define many internal strategy languages in Maude. However, the implementation and the proof of correctness of such languages should rely on the notion of a basic *reflective kernel*, that is, on some basic functionality provided by the universal theory $\mathcal{U}$. The idea proposed in [8] is to first define a *strategy language kernel* as a function *Meta-Level* of rewrite theories, that sends T to a definitional extension of $\mathcal{U}$—or a suitable subtheory of $\mathcal{U}$— that defines how rewriting in T is accomplished at the metalevel. A typical semantic definition that one wants to have in *Meta-Level*(T) is that of *meta-apply*$(\overline{l}, \overline{t})$, that simulates at the metalevel one step of rewriting at the top of a term t using the rule labeled l in T. Proving the correctness of such a small strategy language kernel is then easily done, by using the correctness of $\mathcal{U}$ itself as a universal theory. The next step is to define a strategy language of choice, say *Strategy*, as a function sending each theory T to a theory that extends *Meta-Level*(T) by additional strategy expressions and corresponding semantic rules, all of which are recursive definitional extensions of those in the kernel in an appropriate sense, so that their correctness can then be reduced to that of the kernel.

The Maude implementation supports this novel style of (meta)programming of strategies through the module META-LEVEL below. Formally, META-LEVEL is a module-transforming operation which maps a module T to another module META-LEVEL[T] that is a definitional extension of $\mathcal{U}$. In particular, the following operations, that specify how rewriting in T is accomplished at the metalevel and whose correctness is based on the correctness of $\mathcal{U}$, are defined:

1. meta-reduce takes the meta-representation t of a term as its only argument and is evaluated as follows: (a) t is converted to the term it represents; (b) this term is fully reduced using the equations E in T; (c) the resulting term is converted to a meta-term which is returned.
2. meta-apply2 takes the meta-representation t of a term, a rule label l, a natural number n in Peano representation, and a substitution σ, and is evaluated as follows: (a) t is converted to the term it represents; (b) this term is fully reduced using the equations E in T; (c) the resulting term is matched against all rules with label l partially instantiated with σ, with matches that fail to satisfy the condition of their rule discarded; (d) the first n successful matches are discarded; (e) if there is an $(n+1)$th match, its rule is applied using that match; otherwise {error*, empty} is returned; (f) the new term is fully reduced using the equations E in T; (g) the resulting term is converted to a meta-term which is returned paired with the match used in the reduction. The operator {_,_} is used to construct this pair.
3. meta-apply takes the meta-representation t of a term, a rule label l, and a natural number n in Peano representation, and the result is the first component of the pair returned by the evaluation of meta-apply2 with t, l, n, and the empty substitution as its arguments.

Therefore, META-LEVEL can be considered in our terminology as a kernel internal strategy language for rewriting logic. Its semantic equations extend those of $\mathcal{U}$ and are omitted. The operations ext-term and ext-subs below extract the first and second component respectively from a pair constructed with {_,_}. The actual signature of the module META-LEVEL is the following:

```
mod META-LEVEL[T :: MODULE] is
sorts Op Var Term TermList Label Nat Assignment Substitution ResultPair .
subsort Op < Term . subsort Var < Term . subsort Term < TermList .
subsort Assignment < Substitution .
op 0 : → Nat . op suc_ : Nat → Nat .
op _[_] : Op TermList → Term . op _,_ : TermList TermList → TermList .
op error* : → Term .
op _←_ : Var Term → Assignment . op empty : → Substitution .
op _;_ : Substitution Substitution → Substitution [assoc comm] .
op {_,_} : Term Substitution → ResultPair .
op ext-term : ResultPair → Term . op ext-subs : ResultPair → Substitution .
op meta-reduce : Term → Term . op meta-apply : Term Label Nat → Term .
op meta-apply2 : Term Label Nat Substitution → ResultPair . endm
```

3 Internal Strategies in Maude

In many cases we need to have good ways of controlling the rewriting inference process—which in principle could go in many undesired directions—by means of adequate strategies. Many systems, for example theorem provers and declarative languages implementations, support certain strategies of this nature. However, such strategies are often *external* to the language they control: they may constitute a separate programming language external to the logic, or may be part of the language's "extralogical features". Since rewriting logic is reflective, strategies can be made *internal* to the logic. That is, they can be defined by rewrite rules, and can be reasoned about as with rules in any other theory.

We illustrate this idea by partially specifying a basic internal strategy language in Maude as a module-transforming operation STRATEGY which maps a module T to another module STRATEGY[T] that extends the strategy kernel META-APPLY, defined in Section 2.

We introduce the different function symbols and we give informal explanations when needed. Due to lack of space the formal semantics of some of the functions symbols is omitted; it will appear elsewhere (see also [7] for a complete specification of an internal strategy language similar to the one presented here). Notice that, as an important consequence of the methodology we are following, there is great freedom for defining many different strategy languages inside Maude. This can be done in a completely user-definable way, so that users are not limited by a fixed and closed strategy language. Also, even if some users decide to adopt a particular strategy language because of its good features, such a language remains fully *extensible*, so that new features and new strategies can be defined on top of them.

As expected, the module STRATEGY is parameterized by the theory MODULE and imports the metalevel of its parameter, that is, META-LEVEL[T] for the theory T whose rewriting inference process we want to control. A strategy expression in STRATEGY has the form rew $\langle\ t \parallel w\ \rangle$ with S, where t stands for the metarepresentation of a term in T, S is the rewriting strategy we wish to compute, and w records information relevant for S. Since this information typically has to do with terms in T that somehow are important in the computation of S, w stands for the metarepresentation of a term constructed in T, which may involve some convenient data structures defined by the user. More generally, w can be a term of a sort *Record* containing the sort *Term* of metarepresentations of terms as a subsort. As we will see, as the computation of a given strategy expression proceeds, t gets rewritten by controlled aplication of rules in T, w may be updated, and S is rewritten into the remaining strategy to be computed. In case of termination, this is the idle strategy and we are done. The strategy expression failure is returned when a requested strategy cannot be carried out.

mod STRATEGY **[**T :: MODULE**] is**
protecting META-LEVEL**[**T**]** .
sorts *Record Condition Operation MetaVar Binding Strategy StrategyExpression* .
subsort *Term* $<$ *Record* .
op rew ⟨_||_⟩ with_ : *Term Record Strategy* $\rightarrow$ *StrategyExpression* .
op failure : $\rightarrow$ *StrategyExpression* . **op** idle : $\rightarrow$ *Strategy* .

The most basic strategy we can express in our language is to apply a rule l to a term t. A strategy of the form apply [l] will rewrite t with l using the built-in function meta-apply with the first successful match, if any; otherwise, it will return failure. We can also apply a rule l partially instantiated with a substitution σ, using a strategy of the form apply [l] with σ. In this case, the built-in function meta-apply2 is called.

op apply [_] : *Label* $\rightarrow$ *Strategy* .
op apply [_] with _ : *Label Substitution* $\rightarrow$ *Strategy* .
var t : *Term* . **var** l : *Label* . **var** σ : *Substitution*. **var** w : *Record* .
vars S S' : *Strategy* .
eq rew ⟨ t || w ⟩ with apply [l] = **if** meta-apply(t, l, 0) == error*
  then failure **else** rew ⟨ meta-apply(t, l, 0) || w ⟩ with idle **fi** .
eq rew ⟨ t || w ⟩ with apply [l] with σ = **if** meta-apply2(t, l, 0, σ) == {error*, empty}
  then failure **else** rew ⟨ ext-term(meta-apply2(t, l, 0, σ)) || w ⟩ with idle **fi** .

A strategy of the form apply [l, n] if c will rewrite a term t with the first successful match after the n^{th} one, if any, that also satisfies a user-defined condition c over ⟨ t || w ⟩; otherwise, it will return failure. We also define apply [l, n] with σ if c, with its expected semantics. The operation instc instantiates a condition with a given substitution. The semantics of eval c in ⟨ t || w ⟩ will typically be given by the user as an extension of STRATEGY.

op apply [_,_] if _ : *Label Nat Condition* $\rightarrow$ *Strategy* .
op apply [_,_] with _ if _ : *Label Nat Substitution Condition* $\rightarrow$ *Strategy* .
op instc : *Condition Substitution* $\rightarrow$ *Condition* .
op eval _ in ⟨_||_⟩ : *Condition Term Record* $\rightarrow$ *Term* .
var c : *Condition* . **var** n : *Nat* .
eq rew ⟨ t || w ⟩ with apply [l, n] if c =
  if meta-apply2(t, l, n, empty) == {error*, empty} **then** failure
  else if eval instc(c, ext-subs(meta-apply2(t, l, n, empty))) in ⟨ t || w ⟩ == 'true
    then rew ⟨ ext-term(meta-apply2(t, l, n, empty)) || w ⟩ with idle
    else rew ⟨ t || w ⟩ with apply [l, suc(n)] if c **fi fi** .

The control strategies declared below should be fairly easy to understand by now. Notice that given a sequence of strategies, rew-aux forces a step-by-step computation of the sequence, in the sense that a strategy will not be considered until the previous one has not been fully computed. We use this fact to define the strategy exitloop, which causes the innermost enclosing loop to be exited.

op _;_ : *Strategy Strategy* $\rightarrow$ *Strategy* .
op or {_} else {_} : *Strategy Strategy* $\rightarrow$ *Strategy* .
op if _ then {_} else {_} : *Condition Strategy Strategy* $\rightarrow$ *Strategy* .
op iterate {_} : *Strategy* $\rightarrow$ *Strategy* . **op** while _ {_} : *Condition Strategy* $\rightarrow$ *Strategy* .
op loop {_} : *Strategy* $\rightarrow$ *Strategy* . **op** exitloop : $\rightarrow$ *Strategy*.
op rew-aux _ with_ : *StrategyExpression Strategy* $\rightarrow$ *StrategyExpression* .
eq rew $\langle\ t \parallel w\ \rangle$ with S ; S' = **if** rew $\langle\ t \parallel w\ \rangle$ with S == failure
  then failure **else** rew-aux (rew $\langle\ t \parallel w \rangle$ with S) with S' **fi** . [...]
eq rew $\langle\ t \parallel w\ \rangle$ with loop { S } = rew-aux (rew $\langle\ t \parallel w\ \rangle$ with S) loop { S } .
eq rew-aux (rew $\langle\ t \parallel w\ \rangle$ with idle) with S = rew $\langle\ t \parallel w\ \rangle$ with S .
eq rew-aux (rew $\langle\ t \parallel w\ \rangle$ with exitloop) with S ; S' = rew $\langle\ t \parallel w\ \rangle$ with exitloop . [...]
eq rew-aux (rew $\langle\ t \parallel w\ \rangle$ with exitloop) with loop { S } = $\langle\ t \parallel w\ \rangle$.

A strategy of the form rew $\langle\ t \parallel w\ \rangle$ with let (b) { S } may be used to bind metavariables to certain values inside a strategy S. A metavariable is a constant in the metalevel that can be instantiated to the metarepresentation of any term. A strategy of the form rew $\langle\ t \parallel w\ \rangle$ with apply [l] andlet (b) { S } can be used to apply a rule l to a term t and to bind metavariables inside a strategy S to values that depend on the way in which l is applied. In a similar way, we define apply [l] with σ andlet (b) { S }, apply [l, 0] if c andlet (b) { S }, and apply [l, 0] with σ if c andlet (b) { S }, with their obvious semantics. Finally, the task of updating the relevant information for strategies to come is accomplished with strategies of the form set o, where o is an operation on $\langle\ t \parallel w\ \rangle$ that, in the same form that a condition, is defined using eval.

subsort *MetaVar* < *Term* . **subsort** *Operation* < *Term* .
op _$\hookrightarrow$_ : *Term Term* $\rightarrow$ *Binding* . **op** _,_ : *Binding Binding* $\rightarrow$ *Binding* **[assoc]** .
op let (_) {_} : *Binding Strategy* $\rightarrow$ *Strategy* . **op** bind : *Strategy Binding* $\rightarrow$ *Strategy* .
op set _ : *Operation* $\rightarrow$ *Strategy* . [...]
var b : *Binding* . **var** o : *Operation* .
eq rew $\langle\ t \parallel w\ \rangle$ with let (b) { S } = rew $\langle\ t \parallel w\ \rangle$ with bind(S, b) . [...]
eq rew $\langle\ t \parallel w\ \rangle$ with set o = rew $\langle\ t \parallel$ eval o in $\langle\ t \parallel w\ \rangle\ \rangle$. **endm**

4 Completion Procedures as Internal Strategies

To illustrate our ideas and the semantics of the strategies introduced above, we give a full specification of Huet's algorithm for Knuth-Bendix completion [11] as a strategy in an extension of STRATEGY. The system module COMPLETION below specifies in rewriting logic the completion inference rules KB presented in [10] which are expressed as rewrite rules. Completion of an equational theory is a method that naturally takes place at the metalevel of the theory. Consequently, COMPLETION is parameterized by MODULE and imports the metalevel of the theory we want to complete. As expected, COMPLETION is also parameterized by the theory OTERM that introduces an operation $\succ$ as a well-founded partial (irreflexive) ordering on metarepresentations of terms, that is, there exists no infinite chain $t_1 \succ t_2 \succ \cdots$. In addition to the contructors needed to build sets of equations and rules, the following operations are declared: $\triangleright$, that defines a well-founded ordering on rules for a given reduction ordering $\succ$; nm, that reduces a term to its normal form with respect to a given set of rules; and cp that computes the critical pairs between two rules.

mod COMPLETION[ET :: EQUATIONAL-TH , O :: OTERM] **is**
protecting META-LEVEL[ET] .
sorts *Eqt SetEqt Rl SetRl ERPair* . **subsort** *Eqt* < *SetEqt* . **subsort** *Rl* < *SetRl* .
op $\{\_ \simeq \_\}$: *Term Term* → *Eqt* [comm] .
op $\_ \cup \_$: *SetEqt SetEqt* → *SetEqt* [**assoc comm id:** empty] .
op $\{\_ \to \_\}$: *Term Term* → *Rl* .
op $\_ \cup \_$: *SetRl SetRl* → *SetRl* [**assoc comm idem id:** empty] .
op $\_;\_$: *SetEqt SetRl* → *ERPair* . **op** $\_\triangleright\_$: *Rl Rl* → *Bool* .
op nm : *Term SetRl* → *Term* . **op** cp : *Rl Rl* → *SetEqt* .
vars $s\ t\ l\ r$: *Term* . **var** E : *SetEqt* . **var** R : *SetRl* .
crl [delete] : $(E \cup \{s \simeq t\}; R) \Rightarrow (E; R)$ **if** $s == t$.
crl [compose] : $(E; R \cup \{s \to t\}) \Rightarrow (E; R \cup \{s \to \mathsf{nm}(t, R)\})$ **if** $\mathsf{nm}(t, R) =/= t$.
rl [simplify] : $(E \cup \{s \simeq t\}; R) \Rightarrow (E \cup \{\mathsf{nm}(s, R) \simeq \mathsf{nm}(t, R)\}; R)$.
crl [orient] : $(E \cup \{s \simeq t\}; R) \Rightarrow (E; R \cup \{s \to t\})$ **if** $s \succ t$.
crl [collapse] : $(E; R \cup \{s \to t\} \cup \{l \to r\}) \Rightarrow (E \cup \{\mathsf{nm}(s, \{l \to r\}) \simeq t\}; R \cup \{l \to r\})$
if $\mathsf{nm}(s, \{l \to r\}) =/= s$ **and** $s \to t \triangleright l \to r$.
rl [deduce] : $(E; R \cup \{s \to t\} \cup \{l \to r\}) \Rightarrow$
$(E \cup \mathsf{cp}(\{s \to t\}, \{l \to r\}); R \cup \{s \to t\} \cup \{l \to r\})$.

In [11] Huet proved the correctness of a Knuth-Bendix completion algorithm in which critical pairs are generated incrementally, that is, critical pairs between rules that have been previously considered are not recomputed. In order to provide the strategy language with a convenient data structure to keep this information, we extend COMPLETION with the constructor info, to record for any rule its status—marked or unmarked—and a list of the rules with which critical pairs have been already generated; we also introduce the concatenation operation $\_,\_$, to form an ordered list of these records. Finally, we define several operations to update and retrieve this information, with the expected semantics. The operation adequate checks the ordering of two rules in the list and whether or not critical pairs between them have been already generated. Finally, non-empty-set-eqt checks if a set of equations is empty.

sorts *Status ListRl InfoRl ListInfoRl* . **subsort** *InfoRl* < *ListInfoRl* .
ops marked unmarked : → *Status* . **op** info : *Rl Status ListRl* → *InfoRl* .
op $\_,\_$; *ListInfoRl ListInfoRl* → *ListInfoRl* [**assoc**] . **op** empty-record : → *ListInfoRl* .
op unmarked-rl? : *ERPair ListInfoRl* → *Bool* . **op** non-empty-set-eqt? : *ERPair* → *Bool* .
op adequate : *Rl Rl ListInfoRl* → *Bool* . **op** select-unmarked-rl : *ListInfoRl* → *Rl* .
op add-unmarked-rl : *Rl ListInfoRl* → *ListInfoRl* . **op** mark-rl : *Rl ListInfoRl* → *ListInfoRl* .
op add-cp-rl : *Rl Rl ListInfoRl* → *ListInfoRl* . [. . .] **endm**

Now we can define STRATEGY-COMPLETION for an equational theory and an ordering as an extension of STRATEGY applied to the corresponding COMPLETION module. Notice how operations and conditions are easily defined using user-defined operations.

As in Huet's algorithm, one of three things will happen to a rewriting process started with a term rew $\langle\, \overline{(E\,;\,\emptyset)} \| \,\overline{\text{empty-record}}\,\rangle$ with H-completion: 1) it will stop with success, rewriting to a strategy expression of the form rew $\langle\, \overline{(\emptyset\,;\,R)} \,\|\, \overline{w}\,\rangle$ with idle, where R is a convergent rewrite system; 2) it will stop with failure, rewriting to rew $\langle\, \overline{(E'\,;\,R')} \,\|\, \overline{w}\,\rangle$ with idle, where E' is a non-empty set of equations; or 3) it will loop forever. Finally, notice that because COMPLETION itself imports the metalevel of another theory, STRATEGY-COMPLETION in fact operates at the meta-metalevel, as explained in Sect. 1.3. Since Maude is truly reflective, rewriting at any level in the 'reflective tower' is supported.

```
mod STRATEGY-COMPLETION[ET :: EQUATIONAL-TH, O :: OTERM] is
protecting STRATEGY[COMPLETION[ET, O]] .
ops α ρ γ δ : → MetaVar . ops non-empty-set-eqt? unmarked-rl? : → Condition .
op adequate : Term Term → Condition . op select-unmarked-rl : → Operation .
ops add-unmarked-rl mark-rl : Term → Operation .
op add-cp-rl : Term Term → Operation . op H-completion : → Strategy .
vars t r r' : Term . var w : Record .
eq eval unmarked-rl? in ⟨ t || w ⟩ = meta-reduce('unmarked-rl?[t, w]) . [...]
eq H-completion =
loop { while (non-empty-set-eqt){
            apply [simplify] andlet (λ ↪ 'nm['s, 'R], ρ ↪ 'nm['t, 'R]){
                          or { apply [delete] with ('s ← λ ; 't ← ρ) } else {
                             or { apply [orient] with ('s ← λ ; 't ← ρ) } else
                                 { exitloop } ;
                             set add-unmarked-rl('→[λ, ρ]) ;
                             iterate { apply [collapse] with ('s ← λ ; 't ← ρ) } ;
                             iterate { apply [compose] } } } } ;
    if (unmarked-rl) then { let ('→[λ, ρ] ↪ select-unmarked-rl) {
                        iterate { apply [deduce] with ('s ← λ ; 't ← ρ)
                             if adequate('→[λ, ρ], '→['l, 'r])
                             andlet (δ ↪ 'l, γ ↪ 'r) {
                                   set add-cp-rl('→[λ, ρ], '→[δ, γ]) } } ;
                        set add-marked-rl('→[λ, ρ]) }
                   } else { exitloop } } . endm
```

5 Concluding Remarks

A slightly different version of the completion procedure presented here has been executed in Maude to complete a number of example theories. The slight differences are either purely syntactic, or due to the fact that matching modulo idempotency has not yet been implemented, so that only matching modulo associativity and commutativity (AC) is used. In spite of the fact that three levels of reflection and AC matching are involved, the performance obtained, about 1,500 rewrites per second on a 120MHz Pentium laptop, seems adequate for a prototype. Besides completion, we have experimented with several other examples, such as an inductive theorem-prover for equational logic, strategies for different rewriting logic interpreters (these are strategies for the universal theory $\mathcal{U}$), and several games and AI problem-solving examples. In all cases our solutions cleanly separate the declarative part, given by the rules of the object theory, from the control part, given by internal strategies. Pita and Martí-Oliet [24] report an interesting application of Maude reflection to specify the management of broadband telecommunication networks in which an internal strategy language again permits a clean separation of the object specification from the control aspects.

Aknowledgements. We cordially thank Steven Eker for his extensions to the metalevel of Maude, on which the present experiments are based. We also thank him, Francisco Durán, Patrick Lincoln, and Narciso Martí-Oliet for their very helpful comments to a previous draft and many discussions on these topics. The suggestion that completion algorithms would be a good testing ground for our ideas on strategies was kindly made to us by Mark Stickel some time ago. We have also benefit from discussions on strategies with Maria Paola Bonacina, Claude and Hèlene Kirchner, Peter Borovanský and Pierre-Etienne Moreau. We express our sincere thanks to all of them.

References

1. W. E. Aitken, R. L. Constable, and J. L. Underwood. Metalogical frameworks II: Using reflected decision procedures. Technical Report, Computer Sci. Dept., Cornell University, 1993; also, lecture at the Max Planck Institut für Informatik, Saarbrücken, Germany, July 1993.
2. L. Bachmair, N. Dershowitz, and J. Hsiang. Ordering for equational proofs. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 346–357, 1986.
3. M. P. Bonacina and J. Hsiang. Parallelization of deduction strategies: An analytical study. *Journal of Automated Reasoning*, 13:1–33, 1994.
4. 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*, pages 168–188. Elsevier, September 1996.
5. 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 Theoretical Computer Science*, pages 35–50. Elsevier, September 1996.
6. 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 of *Electronic Notes in Theoretical Computer Science*, pages 65–89. Elsevier, September 1996.
7. M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, editor, *Proceedings of Reflection'96*, pages 263–288. Xerox PARC, 1996.
8. M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 125–147. Elsevier, September 1996.
9. R. L. Constable. Using reflection to explain and enhance type theory. In H. Schwichtenberg, editor, *Proof and Computation*, volume 139 of *Computer and System Sciences*, pages 109–144. Springer, 1995.

10. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. North-Holland, 1990.
11. G. Huet. A complete proof of correctness of the Knuth and Bendix completion algorithm. *Journal of Computer and System Sciences*, 23(1):11–21, 1981.
12. G. Kiczales, editor. *Proceedings of Reflection'96*. Xerox PARC, 1996.
13. H. Kirchner and P. E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proceedings 6th Conference on Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Computer Science*, pages 438–443. Springer-Verlag, 1995.
14. D. Knuth and P. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.
15. P. Lescanne. Completion procedures as transition rules + control. In M. Diaz and F. Orejas, editors, *TAPSOFT'89*, volume 351 of *Lecture Notes in Computer Science*, pages 28–41. Springer-Verlag, 1989.
16. P. Lincoln, J. Meseguer, and L. Ricciulli. The Rewrite Rule Machine Node Architecture and its Performance. In B. Buchberger and J. Volkert, editors, *Proceedings of CONPAR'94*, pages 509–520, 1994.
17. 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, editor, *Handbook of Philosophical Logic*, Kluwer Academic Publishers.
18. J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editors, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.
19. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
20. J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.
21. J. Meseguer. Rewriting logic as a semantic framework for concurrency: A progress report. In U. Montanari and V. Sassone, editors, *Proceedings of CONCUR'96*, Lecture Notes in Computer Science, pages 331–372. Springer-Verlag, 1996.
22. J. Meseguer and T. Winkler. Parallel programming in Maude. In J.-P. Banâtre and D. L. Mètayer, editors, *Research Directions in High-level Parallel Programming Languages*, volume 574 of *Lecture Notes in Computer Science*, pages 253–293. Springer-Verlag, 1992.
23. L. Paulson. Tactics and tacticals in Cambridge LCF. Technical Report 39, University of Cambridge, 1983.
24. I. Pita and N. Martí-Oliet. Using reflection to specify transaction sequences in rewriting logic. In *Proceedings of APPIA-GULP-PRODE*, 1997.
25. H. Rueß. Reflection of formal tactics in a deductive reflection framework. Manuscript, Universität Ulm, Abt. Künstliche Intelligenz, January 96.
26. B. Smith and A. Yonezawa, editors. *Proceedings of the IMSA'92 International Workshop on Reflection and Meta-Level Architecture*. Research Institute of Software Engineering, 1992.