Current Design and Implementation of the Cafe Prover and Knuth-Bendix Checker Tools

Manuel Clavel, Steven Eker, and José Meseguer
Computer Science Laboratory
SRI International

October 1997

1 Introduction

The present document gives detailed explanations of our reflective design for the inductive theorem prover and the Knuth-Bendix checker that we are building as part of the Cafe project. The Maude code for each of the tools is also given and discussed in its entirety.

Since the reflective nature of rewriting logic and Maude, and the closely related concept of *internal* strategies, are key features of our design, we dedicate two sections to explain in detail how these concepts are supported by the Maude implementation. Each tool, including the formal system that it implements and its corresponding Maude implementation, is then explained in a separate section.

In the rest of this introduction we give high-level informal explanations of the key concepts on reflection and strategies at the heart of our design and implementation for the tools, and of the tools themselves.

Rewriting logic is reflective [3, 4] in the precise sense that there is a finitely presented rewrite theory U such that for any finitely presented rewrite theory T (including U itselft) we have the following equivalence

$$T \vdash t \longrightarrow t' \iff U \vdash \langle \overline{T}, \overline{t} \rangle \longrightarrow \langle \overline{T}, \overline{t'} \rangle,$$

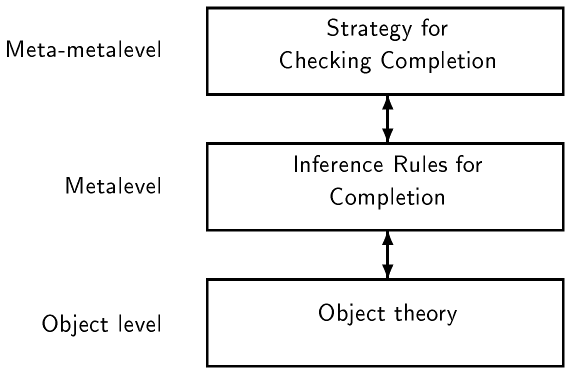
where $\overline{T}$ and $\overline{t}$ are terms representing T and t as data elements of U. Since U is representable in itself, we can achieve a "reflective tower" with an arbitrary number of levels of reflection, since we have

$$T \vdash t \longrightarrow t' \iff U \vdash \langle \overline{T}, \overline{t} \rangle \longrightarrow \langle \overline{T}, \overline{t'} \rangle \iff U \vdash \langle \overline{U}, \overline{\langle \overline{T}, \overline{t} \rangle} \rangle \longrightarrow \langle \overline{U}, \overline{\langle \overline{T}, \overline{t'} \rangle} \rangle \ldots$$

For efficiency reasons, the Maude implementation provides key features of the universal theory U in a built-in module called `META-LEVEL`. In particular, `META-LEVEL` provides sorts `Term` and `Module`, so that the representations $\overline{t}$ and $\overline{T}$ of a term t and a module T have sorts $\overline{t}$: `Term` and $\overline{T}$: `Module`. It also provides functions `meta-reduce`($\overline{T}$, $\overline{t}$) and `meta-apply`($\overline{T}$, $\overline{t}$, $\overline{l}$, $\overline{\sigma}$, n) that return, respectively, the representation of the reduced form of a term t using the equations in a module T, and the result of applying a rule labelled l in the module T to a term t at the top with the $(n+1)$th match consistent with the partial substitution σ. As the universal theory U that it implements in a built-in fashion, `META-LEVEL` can also support a reflective tower with an arbitrary number of levels of reflection.

`META-LEVEL` can then be used as a *reflective kernel* that can be extended in a completely user-definable way to specify *strategy languages* controlling the rewriting process in a way *internal* to rewriting logic itself. The basic idea is that `meta-apply` allows us to take *elementary steps* in the deduction process, but we can take *bigger steps* by defining more complex *strategy expressions* in a module extending `META-LEVEL` by *semantic equations* that define in rewriting logic the precise rewriting semantics for each of the constructors building up strategy expressions in our strategy language of choice. A simple strategy language of this nature, yet powerful enough for defining all the strategies used in the tools, is explained in detail in Section 3.

Based on these concepts, each of the Cafe tools has then a very simple design. Consider for example the design of the Knuth-Bendix checker. Its purpose is to analyze a Cafe functional module T, returning the set of critical pairs that cannot be simplified to the same term. The theory T that we want to check is at the object level. The rules of inference for Knuth-Bendix completion can be naturally expressed as a rewrite theory $\mathcal{C}$. But since this rewrite theory uses T as a data structure—that is, it actually uses its representation $\overline{T}$—the theory $\mathcal{C}$ should be defined at the metalevel. Checking confluence of the critical pairs in T corresponds to applying the rules in $\mathcal{C}$ with a given *strategy* (a different strategy would for example be needed for completing T instead of just checking whether it is already complete). But since the strategies for any rewrite theory belong to the metalevel of such a theory, and $\mathcal{C}$ is already at the metalevel, we need *three levels of reflection* to clearly distinguish levels and make our design entirely *modular*, so that, for example, we can change the strategy without any change whatsoever to the inference rules in $\mathcal{C}$. This is illustrated by the following picture



The same reflective design, with three levels of reflection, applies to the inductive theorem prover. Again, the module T about which we want to prove inductive theorems is at the object level. An inference system $\mathcal{I}$ for inductive proofs uses $\overline{T}$ as a data structure and therefore is a rewrite theory at the metalevel. Then, different *proof tactics* to guide the application of the inference rules are strategy expressions at the meta-metalevel.

It goes without saying that the actual implementation of the tools is still *work in progress*, to be extended and completed in the following months. We think that the present state of the inductive theorem prover is actually quite advanced, but the current Knuth-Bendix checker only handles at present *unsorted* and *unconditional* functional modules. We hope to extend it to a checker for order-sorted and conditional modules within the next few months, and to allow it to handle *commutative* operations as well if time permits us to do so.

2 A Reflective Kernel in Maude

Maude is a logical language based on rewriting logic [8, 10, 9]. An introduction to Maude and its interpreter implementation can be found in [2]. 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.

The reflective kernel of the Maude system is the built-in module `META-LEVEL`, in which:

- Maude terms are reified as elements of a data type `Term` of terms;
- Maude modules are reified as terms in a data type `Module` of modules;
- the process of reducing a term to normal form in a functional module (whose operations are assumed to be Church-Rosser and terminating) is reified by a function `meta-reduce`; and, similarly,
- the process of applying a rule of a system module to a subject term is reified by a function `meta-apply`.

The module `META-LEVEL` provides key functionality in the *universal theory* U for rewriting logic [3, 4], that is, a theory in rewriting logic such that for any other finitary rewrite theory $T = (\Sigma, E, R)$ and for any terms $t, t' \in T_\Sigma(\mathrm{X})$,we have

$$T \vdash t \longrightarrow t' \iff U \vdash \langle \overline{T}, \overline{t} \rangle \longrightarrow \langle \overline{T}, \overline{t'} \rangle,$$

where $\overline{T}$: `Module`, and $\overline{t'}$: `Term` are the data elements reifying T and t in U.

The current version of Maude has the following signature for `META-LEVEL`:

```
fmod META-LEVEL is
sorts Bool MachineInt NzMachineInt Qid Module Import
      ImportList QidList QidSet MachineIntList SortDecl
```

```
      SubsortDecl SubsortDeclSet Attr AttrSet OpDecl OpDeclList
      VarDecl VarDeclSet Term TermList Equation EquationSet
      Rule RuleSet MembAx MembAxSet Assignment Substitution ResultPair .
subsort NzMachineInt < MachineInt .
subsort Import < ImportList .
subsort Qid < QidList .
subsort Qid < QidSet .
subsort MachineInt < MachineIntList .
subsort SubsortDecl < SubsortDeclSet .
subsort Attr < AttrSet .
subsort OpDecl < OpDeclList .
subsort VarDecl < VarDeclSet .
subsort Qid < Term .
subsort Term < TermList .
subsort Equation < EquationSet .
subsort Rule < RuleSet .
subsort MembAx < MembAxSet .
subsort Assignment < Substitution .
op true : -> Bool .
op false : -> Bool .
op if_then_else_fi : (implemented by C++ code) .
op _==_ : (implemented by C++ code) .
op _=/=_ : (implemented by C++ code) .
op _===_ : (implemented by C++ code) .
op _=/==_ : (implemented by C++ code) .
cop 0 : -> MachineInt .
op -_ : MachineInt -> MachineInt .
op -_ : NzMachineInt -> NzMachineInt .
op ~_ : MachineInt -> MachineInt .
op _+_ : MachineInt MachineInt -> MachineInt .
op _-_ : MachineInt MachineInt -> MachineInt .
op _*_ : MachineInt MachineInt -> MachineInt .
op _*_ : NzMachineInt NzMachineInt -> NzMachineInt .
op _/_ : MachineInt NzMachineInt -> MachineInt .
op _%_ : MachineInt NzMachineInt -> MachineInt .
op _&_ : MachineInt MachineInt -> MachineInt .
op _|_ : MachineInt MachineInt -> MachineInt .
op _|_ : NzMachineInt NzMachineInt -> NzMachineInt .
op _^_ : MachineInt MachineInt -> MachineInt .
op _>>_ : MachineInt MachineInt -> MachineInt .
op _<<_ : MachineInt MachineInt -> MachineInt .
op _<_ : MachineInt MachineInt -> Bool .
op _<=_ : MachineInt MachineInt -> Bool .
op _>_ : MachineInt MachineInt -> Bool .
op _>=_ : MachineInt MachineInt -> Bool .
cop ' : -> Qid .
```

```
op conc : Qid Qid -> Qid .
op index : Qid MachineInt -> Qid .
cop protecting : Qid -> Import .
cop nilImportList : -> ImportList .
cop importList : ImportList ImportList -> ImportList [assoc] .
cop _[_] : Qid TermList -> Term .
cop _,_ : TermList TermList -> TermList [assoc] .
cop membPred : Term Qid -> Term .
cop lazyMembPred : Term Qid -> Term .
cop nilQidList : -> QidList .
cop qidList : QidList QidList -> QidList [assoc] .
cop emptyQidSet : -> QidSet .
cop qidSet : QidSet QidSet -> QidSet [assoc comm id: emptyQidSet] .
cop nilMachineIntList : -> MachineIntList .
cop machineIntList : MachineIntList MachineIntList ->
                     MachineIntList [assoc] .
cop sort : QidSet -> SortDecl .
cop subsort : Qid Qid -> SubsortDecl .
cop emptySubsortDeclSet : -> SubsortDeclSet .
cop subsortDeclSet : SubsortDeclSet SubsortDeclSet -> SubsortDeclSet
                     [assoc comm id: emptySubsortDeclSet] .
cop assoc : -> Attr .
cop comm : -> Attr .
cop idem : -> Attr .
cop id : Term -> Attr .
cop strat : MachineIntList -> Attr .
cop emptyAttrSet : -> AttrSet .
cop attrSet : AttrSet AttrSet -> AttrSet [assoc comm id: emptyAttrSet] .
cop opDecl : Qid QidList Qid AttrSet -> OpDecl .
cop nilOpDeclList : -> OpDeclList .
cop opDeclList : OpDeclList OpDeclList -> OpDeclList [assoc] .
cop varDecl : Qid Qid -> VarDecl .
cop emptyVarDeclSet : -> VarDeclSet .
cop varDeclSet : VarDeclSet VarDeclSet -> VarDeclSet
                 [assoc comm id: emptyVarDeclSet] .
cop eq : Term Term -> Equation .
cop ceq : Term Term Term Term -> Equation .
cop emptyEquationSet : -> EquationSet .
cop equationSet : EquationSet EquationSet -> EquationSet
                  [assoc comm id: emptyEquationSet] .
cop rl : Qid Term Term -> Rule .
cop crl : Qid Term Term Term Term -> Rule .
cop emptyRuleSet : -> RuleSet .
cop ruleSet : RuleSet RuleSet -> RuleSet [assoc comm id: emptyRuleSet] .
cop mb : Term Qid -> MembAx .
cop cmb : Term Qid Term Term -> MembAx .
```

```
cop emptyMembAxSet : -> MembAxSet .
cop membAxSet : MembAxSet MembAxSet -> MembAxSet
                [assoc comm id: emptyMembAxSet] .
cop fmod : Qid ImportList SortDecl SubsortDeclSet
           OpDeclList VarDeclSet MembAxSet EquationSet -> Module .
cop mod : Qid ImportList SortDecl SubsortDeclSet OpDeclList
          VarDeclSet MembAxSet EquationSet RuleSet -> Module .
cop error* : -> Term .
cop _<-_ : Qid Term -> Assignment .
cop emptySubstitution : -> Substitution .
cop _;_ : Substitution Substitution -> Substitution
          [assoc comm id: emptySubstitution] .
cop {_,_} : Term Substitution -> ResultPair .
op meta-reduce : Module Term -> Term .
op meta-apply : Module Term Qid Substitution MachineInt -> ResultPair .
endfm
```

The data types of integers and identifiers are built-in in Maude for efficiency and convenience. They are pre-defined in the modules `MACHINE-INT` and `QID`, that are imported by `META-LEVEL`. The sorts `MachineInt` and `NzMachineInt` provide integers and non-zero integers, each with the usual operations, having the expected attributes. The sort `Qid` contains quoted identifiers beginning with the apostrophe symbol, e.g., `'a`, `'b`, `'1040`, `'aratherlongidentifier`. The only operations defined over the elements of `Qid` are `conc` and `index`. The function `conc` concatenates identifiers in the following way: `conc('a, 'b)= 'ab`. The function `index` concatenates an integer with an identifier, e.g., `index('a, 1)= 'a1`.

As already mentioned, `META-LEVEL` provides an abstract data type `Module` to represent modules in Maude. As an example, we introduce the module NAT.

```
fmod NAT is
sorts Nat NzNat .
subsort NzNat < Nat .
op 0 : -> Nat .
op s : Nat -> NzNat .
op + : Nat Nat -> Nat [assoc comm] .
vars N M : Nat .
eq +(N, 0) = N .
eq +(N, s(M)) = s(+(N, M)) .
```

The representation of `NAT` in `META-LEVEL` is the following term of sort `Module`:

```
fmod('NAT,
     nilImportList,
     sort(qidSet('Nat, 'NzNat)),
     subsort('NzNat, 'Nat),
     opDeclList(
```

```
      opDecl('0, nilQidList, 'Nat, emptyAttrSet),
      opDecl('s, 'Nat, 'NzNat, emptyAttrSet),
      opDecl('+, qidList('Nat, 'Nat), 'Nat, attrSet(assoc, comm))),
    varDeclSet(
      varDecl('N, 'Nat),
      varDecl('M, 'Nat)),
    emptyMembAxSet,
    equationSet(
      eq('+['N, '0], 'N),
      eq('+['N, 's['M]], 's['+['N, 'M]])))
```

The operations `meta-reduce` and `meta-apply` are key for the reflective capabilities of the Maude system. The operation `meta-reduce` takes the representations in `META-LEVEL` of a module T and a term t or a membership predicate $t : s$ in that module. When the second argument is the representation of a term t, `meta-reduce` returns the representation of the term t fully reduced using the equations in T, e.g,

```
> meta-reduce(fmod('NAT,...), '+['s['0], '0]) .
+> 's['0]
```

Membership predicates are represented in `META-LEVEL` using the constructors `membPred` and `lazyMembPred`. When the second argument of `meta-reduce` is the representation of a membership predicate $t : s$ using the constructor `membPred`, the term t is fully reduced using the equations in T and then the representation of the value of the resulting membership predicate is returned, e.g.,

```
> meta-reduce(fmod('NAT,...), membPred('+['s['0], '0], 'NzNat)) .
+> 'true
```

On the other hand, if the constructor `lazyMembPred` is used to represent a membership predicate $t : s$, the term t is not reduced, so that only a sort that can be inferred without reducing the term yields the value true, e.g.,

```
> meta-reduce(fmod('NAT,...), membPred('+['s['0], '0], 'NzNat)) .
+> 'false
> meta-reduce(fmod('NAT,...), membPred('+['s['0], '0], 'Nat)) .
+> 'true
```

The operation `meta-apply` has five arguments. The first four arguments are representations in `META-LEVEL` of a module T, a term t in T, a label l of some rules in T, and a set of assignments (possibly empty) σ for the variables in those rules. The last argument is and a natural number n. An assignment $x \leftarrow t$ is represented in `META-LEVEL` using the constructor `_<-_`. The operation `meta-apply` is evaluated as follows:

1. the term t is first fully reduced using the equations in T;

2. 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;

3. the first n successful matches are discarded; if there is an $(n+1)$th match, its rule is applied using that match and the steps 4 y 5 below are taken; otherwise {`error*, emptySubstitution`} is returned;

4. the term resulting from applying the given rule with the $(n+1)$th match is fully reduced using the equations in T;

5. the pair formed using the constructor {_,_} whose first element is the representation of the resulting fully reduced term and whose second element is the representation of the match used in the reduction is returned.

A number of examples of applications of `meta-apply` will be shown in the next section.

The operations `meta-reduce` and `meta-apply` are functions equationally definable in the module `META-LEVEL`. This is a consequence of the fact that rewriting logic is a reflective logic (see [4]). However, for efficiency and convenience they are built-in functions in Maude. Assuming that all the built-in functions are in fact equationally defined, `META-LEVEL` then becomes a normal module in Maude that can itself be represented as a term of sort `Module` in `META-LEVEL`. The Maude implementation produces the exact same behavior as if `META-LEVEL` had thus been defined as a normal Maude module, but for efficiency reasons it perfoms the corresponding computations in a built-in manner. In this way, the Maude implementation supports a "reflective tower" with an arbitrary number of levels of reflection, e.g.,

```
> meta-reduce(fmod('META-LEVEL,...),
          'meta-reduce['fmod[''NAT,...],
                      '_[_][''+, '_,_['_[_][''s, ''0], ''0]]]) .
+> '_[_][''s, ''0],
```

where now the result is the term $\overline{\overline{\texttt{s(0)}}}$: `Term` that results from enconding $s(0)$ twice.

3 Internal Strategies in Maude

System modules in Maude are rewrite theories that need not to be Church-Rosser of terminating. 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*. In Maude, thanks to its reflective capabilities, strategies can be made *internal* to the system. That is, they can be defined by rewrite rules in a normal module in Maude, and can be reasoned about as with rules in any other module.

In fact, 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. The idea is to use the operations `meta-reduce` and `meta-apply` as basic strategy expressions, and then to extend the module `META-LEVEL` by additional strategy expressions and corresponding semantic rules. Here we follow the methodology for defining and proving correct internal strategy languages for reflective logics introduced in [3, 4].

To illustrate this idea, we introduce the module `SORTING` for sorting vectors of integers. Such vectors are represented as sets of pairs of integers, with the first component of each pair corresponding to the vector position and the second containing the number in that position.

```
mod SORTING is
protecting MACHINE-INT .
sorts Pair PairSet .
subsort Pair < PairSet .
op and : Bool Bool -> Bool [assoc comm] .
op pair : MachineInt MachineInt -> Pair .
op empty : -> PairSet .
op pairSet : PairSet PairSet -> PairSet [assoc comm id: empty] .
var I J X Y : MachineInt .
var B : Bool .

eq and(B, true) = B .
eq and(B, false) = false .

crl [sort]: pairSet(pair(J, X), pair(I, Y)) =>
            pairSet(pair(J, Y), pair(I, X))
            if and((J < I), (X > Y)) .

endm
```

In the present version of Maude, the sort `Bool` with constants `true` and `false` is added implicitly to any module. However, no Boolean functions are added, so that `and` is here defined explicitly. The sort `PairSet` contains sets P of pairs of integers. For the sake of the example, let us suppose that for any pair `pair`(i, x) in an input set P, $i \in \text{card}(P)$, and we cannot have two different pairs `pair`(i, x) and `pair`(j, y) in P such that $i = j$. That is, any input set P is indeed a vector of integers. Thus, the rule `[sort]` modifies a vector of integers in order to put it into sorted order. We can use the default-interpreter implemented in Maude for executing system modules, which applies the rules in a fair top-down fashion, to sort a vector of integers, e.g.,

```
> pairSet(pair(1, 3), pair(2, 2), pair(3, 1)) .
+> pairSet(pair(1, 1), pair(2, 2), pair(3, 3))
```

However, using the default-interpreter we have no control over the application of the rule `[sort]`. Although not a problem in this case, in general we want to control the way the rules are applied. As mentioned before, strategy languages can be defined within Maude in user-definable extensions of the module META-LEVEL. As an example, we introduce the following module STRATEGY.

```
fmod STRATEGY is
protecting META-LEVEL[SORTING] .
sorts MetaVar Binding BindingList Strategy StrategyExpression .
subsort MetaVar < Term .

ops I J N : -> MetaVar .
op binding : MetaVar Term -> Binding .
op nilBindingList : -> BindingList .
op bindingList : Binding BindingList -> BindingList .

op rewInWith : Module Term BindingList Strategy -> StrategyExpression .
op set : MetaVar Term -> Strategy .
op rewInWithAux : StrategyExpression Strategy -> StrategyExpression .
op idle : -> Strategy .
op failure : -> StrategyExpression .
op and : Strategy Strategy -> Strategy .
op apply : Qid -> Strategy .
op applyWithSubst : Qid Substitution -> Strategy .
op iterate : Strategy -> Strategy .
op while : Term Strategy -> Strategy .
op orelse : Strategy Strategy -> Strategy .

op extTerm : ResultPair -> Term .
op extSubst : ResultPair -> Substitution .
op update : BindingList Binding -> BindingList .
op applyBindingListSubst : Module Substitution BindingList -> Substitution .
op substituteMetaVars : TermList BindingList -> TermList .

var M : Module .
vars V V' F G L : Qid .
vars T T' : Term .
var TL : TermList .
var SB : Substitution .
vars B B' : Binding .
vars BL BL' : BindingList .
var MV MV' : MetaVar .
vars ST ST' : Strategy .
```

The declaration protecting META-LEVEL[Id], where Id is the name of a module T, imports the module META-LEVEL, declares a new constant Id of sort Module, and adds an equation making Id equal to the representation of T in META-LEVEL.

The function rewInWith computes strategy expressions. The first two arguments of rewInWith are the representations of a module T and a term t in META-LEVEL. The fourth argument is the strategy S we want to compute, and the third argument is used to store information that may be relevant for S. Our definition of rewInWith is such that, as the computation of a given strategy expression proceeds, t gets rewritten by controlled application of rules in T, the information stored in the third argument may be updated, and the strategy 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.

A basic strategy we can express is the application of a rule once at the top of a term (if the top operator has attributes containing axioms such as associativity or associativity and commutativity, matching is done with extension) with the first possible match found when no constraints are placed on the matching substitution. For this basic strategy, we introduce in our signature the constructor apply, whose only argument is an identifier corresponding to the rule label to be applied, and we define the value of rewInWith for this strategy, using the built-in operation meta-apply, as follows:

```
eq rewInWith(M, T, BL, apply(L)) =
   if meta-apply(M, T, L, emptySubstitution, 0) ==
     {error*, emptySubstitution}
   then failure
   else rewInWith(M, extTerm(meta-apply(M, T, L, emptySubstitution, 0)),
                  BL, idle) fi .
```

The operations extTerm and extSubst are selectors extracting the first and second component, respectively, from a pair constructed with {_,_}.

```
eq extSubst(T, SB) = SB .
eq extTerm(T, SB) = T .
```

We can see the computation of an apply-strategy expression with the following example:

```
> rewInWith(SORTING,
            'pairSet['pair['1, '3], 'pair['2, '2], 'pair['3, '1]],
            nilBindingList,
            apply('sort)).
+> rewInWith(SORTING,
            'pairSet['pair['1, '2], 'pair['2, '3], 'pair['3, '1]],
            nilBindingList,
            idle)
```

The information relevant for the computation of a strategy expression is recorded as a list of bindings of values to metavariables, where the values are of sort Term, that is, they are representations of terms, and metavariables are introduced by the users as constants of sort MetaVar. The sort MetaVar is declared as a subsort of the sort `Term`, so that in any expression in which the representation of a term t can appear, a metavariable—to which the representation of t may be bound—can appear as well.

The computation of the strategy `set` updates the recorded information. This is done by the function `update`. Notice that the terms whose representations are bound to metavariables are kept in fully reduced form, using the built-in operation `meta-reduce`. The representation of the term set to a metavariable may itself contain metavariables, that must be substituted using the representations of the terms they are bound to in the list of bindings present before the updating. This is done by the function `substituteMetaVars`. Recall that the default operational semantics for functional modules, and therefore for its metalevel representation by `meta-reduce`, is "eager" (i.e., bottom up or call-by-value).

```
eq rewInWith(M, T, BL, set(MV, T')) =
   rewInWith(M, T,
     update(BL,
            binding(MV, meta-reduce(M, substituteMetaVars(T', BL)))), idle) .

eq substituteMetaVars(T, nilBindingList) = T .
eq substituteMetaVars(MV, bindingList(binding(MV', T'), BL)) =
   if MV == MV' then T' else substituteMetaVars(MV, BL) fi .
eq substituteMetaVars(F, BL) = F .
eq substituteMetaVars(F[TL], BL) = F[substituteMetaVars(TL, BL)] .
eq substituteMetaVars((T, TL), BL) =
   (substituteMetaVars(T, BL), substituteMetaVars(TL, BL)) .

eq update(bindingList(binding(MV, T), BL), binding(MV', T')) =
   if MV == MV'
   then bindingList(binding(MV, T'), BL)
   else bindingList(binding(MV, T), update(BL, binding(MV', T'))) fi .
eq update(nilBindingList, B) = bindingList(B, nilBindingList) .
```

We can see the computation of a `set`-strategy expression with the following example:

```
> rewInWith(SORTING,
            'pairSet['pair['1, '3], 'pair['2, '2], 'pair['3, '3]],
            nilBindingList,
            set(I, '1)).
+> rewInWith(SORTING,
             'pairSet['pair['1, '3], 'pair['2, '2], 'pair['3, '3]],
             bindingList(binding(I, '1), nilBindingList),
             idle)
```

The computation of the strategy `applyWithSubst` applies a rule partially instantiated with a set of assignments once at the top of a term (if the top operator has attributes containing axioms such as associativity or associativity and commutativity, matching is done with extension) using the first successful match consistent with the given partial substitution. The representations of the terms assigned to variables may contain metavariables that must be substituted using the representations of the terms they are bound to in the current list of bindings. This is done by the function applyBindingListSubst.

```
eq rewInWith(M, T, BL, applyWithSubst(L, SB)) =
  if meta-apply(M, T, L, applyBindingListSubst(M, SB, BL), 0) ==
    {error*, emptySubstitution}
  then failure
  else rewInWith(M, extTerm(meta-apply(M, T, L,
                            applyBindingListSubst(M, SB, BL), 0)),
                 BL, idle) fi .

eq applyBindingListSubst(M, emptySubstitution, BL) = emptySubstitution .
eq applyBindingListSubst(M, ((V <- T); SB), BL) =
   ((V <- meta-reduce(M, substituteMetaVars(T, BL)));
    applyBindingListSubst(M, SB, BL)) .
```

The semantic equations for the strategies and, orelse, and iterate are defined as expected.

```
eq rewInWith(M, T, BL, and(ST, ST')) =
   if rewInWith(M, T, BL, ST) == failure
   then failure
   else rewInWithAux(rewInWith(M, T, BL, ST), ST') fi  .

eq rewInWith(M, T, BL, orelse(ST, ST')) =
   if rewInWith(M, T, BL, ST) == failure
   then rewInWith(M, T, BL, ST')
   else rewInWith(M, T, BL, ST) fi .

eq rewInWith(M, T, BL, iterate(ST)) =
   if rewInWith(M, T, BL, ST) == failure
   then rewInWith(M, T, BL, idle)
   else rewInWithAux(rewInWith(M, T, BL, ST), iterate(ST)) fi .
```

where the function rewInWithAux is defined by the semantic equation

```
eq rewInWithAux(rewInWith(M, T, BL, idle), ST) = rewInWith(M, T, BL, ST) .
```

which forces the computation of a sequence of strategies to proceed step-by-step, in the sense that a strategy will only be considered after the previous one has been fully computed. We can illustrate the computation of the above strategies by means of the following examples:

```
> rew rewInWith(SORTING,
              'pairSet['pair['1, '3], 'pair['2, '2], 'pair['3, '1]],
```

```
              nilBindingList,
              and(set(I, '1), apply('sort))) .
+> rewInWith(SORTING,
             'pairSet['pair['1, '2], 'pair['2, '3], 'pair['3, '1]],
             bindingList(binding(I, '1), nilBindingList),
             idle)
> rew rewInWith(SORTING,
              'pairSet['pair['1, '3], 'pair['2, '2], 'pair['3, '1]],
              bindingList(binding(J, '2), nilBindingList),
              orelse(applyWithSubst('sort, ('J <- '4)),
                     applyWithSubst('sort, ('J <- J)))).
+> rewInWith(SORTING,
             'pairSet['pair['1, '3], 'pair['2, '1], 'pair['3, '2]],
             bindingList(binding(J, '2), nilBindingList),
             idle)
> rew rewInWith(SORTING,
              'pairSet['pair['1, '3], 'pair['2, '2], 'pair['3, '1]],
              nilBindingList,
              iterate(apply('sort))).
+> rewInWith(SORTING,
             'pairSet['pair['1, '1], 'pair['2, '2], 'pair['3, '3]],
             nilBindingList,
             idle)
```

Finally, the strategy `while` makes the computation of a given strategy conditional to the satisfaction of a condition. This condition should be the representation in `META-LEVEL` of a term of sort `Bool`. As usual, the condition may contain metavariables that must be substituted using the representations of terms they are bound to in the current list of bindings.

```
eq rewInWith(M, T, BL, while(T', ST)) =
   if meta-reduce(M, substituteMetaVars(T', BL)) == 'true
   then (if rewInWith(M, T, BL, ST) == failure
         then rewInWith(M, T, BL, idle)
         else rewInWithAux(rewInWith(M, T, BL, ST), while(T', ST)) fi)
    else rewInWith(M, T, BL, idle) fi .
```

Now we can extend our basic strategy language to define, as an example, the algorithm for insertion sorting. The strategy `insert`(n) below can be used to sort a vector of integers of lenght n. The main loop in insertion sorting looks at each element of the vector of integers from the second to the n-th, and inserts it in the appropiate place among its predecessors in the vector.

```
op insert : MachineInt -> Strategy .
var Length : MachineInt .

eq insert(Length) =
```

```
  and(set(N, '2),
      while('_<=_[N, index(', Length)],
            and(set(I, N),
            and(while('_>_[I, '1],
                      and(applyWithSubst('sort,
                                       (('I <- I);
                                        ('J <- '_-_[I, '1]))),
                          set(I, '_-_[I, '1]))),
             set(N, '_+_[N, '1]))))) .
```

For example, we can use the strategy insert to sort a vector of integers of length 10:

```
> rew rewInWith(SORTING,
                'pairSet['pair['1, '10], 'pair['2, '9], 'pair['3, '8],
                'pair['4, '7], 'pair['5, '6], 'pair['6, '5],
                'pair['7, '4], 'pair['8, '3], 'pair['9, '2], 'pair['10, '1]],
                nilBindingList,
                insert(10)) .
+> rewrites: 3655 in 189ms cpu (269ms real) (19236 rewrites/second)
+> result StrategyExpression:
   rewInWith(SORTING,
             'pairSet['pair['1, '1], 'pair['2, '2], 'pair['3, '3],
             'pair['4, '4], 'pair['5, '5], 'pair['6, '6],
             'pair['7, '7], 'pair['8, '8], 'pair['9, '9],
             'pair['10, '10]],
              bindingList(binding(N, '11), bindingList(binding(I, '1),
                         nilBindingList)),
              idle)
```

4 A Knuth-Bendix Completion Checker

We discuss the inference rules for completion, their implementation in Maude, and the strategy of the checker.

4.1 Inference Rules for Completion

The module KBC below is a partial specification in Maude of the inference system KB for Knuth-Bendix completion introduced in [1]. Here, we will follow the presentation of the system KB given in [6].

KBC specifies the subset of rules of inference KBC⊆KB that are needed to check that a terminating rewrite theory R is confluent. Thus, only the rules **Simplify**, **Delete**, and **Deduce** in KB are specified. Further details regarding the complete specification of the inference system KB can be found in [5].

```
mod KBC is
```

```
protecting META-LEVEL .
sorts EquationC EquationCSet Position
      PositionSet UnifPair ComplPair .

subsorts MachineInt < Position < PositionSet .
subsorts EquationC < Term EquationCSet .

op and : Bool Bool -> Bool [assoc comm] .
op or : Bool Bool -> Bool [assoc comm] .
op nor : Bool Bool -> Bool [assoc comm] .
op eqc : Term Term -> EquationC [assoc comm].
op emptyEquationCSet : -> EquationCSet .
op equationCSet : EquationCSet EquationCSet -> EquationCSet
                  [assoc comm id: emptyEquationCSet] .
op size : VarDeclSet TermList -> MachineInt .
op max : MachineInt MachineInt -> MachineInt .
op min : MachineInt MachineInt -> MachineInt .
op greaterEquationC : VarDeclSet EquationC EquationC -> Bool .
op isVar? : VarDeclSet Term -> Bool .
op rename : VarDeclSet TermList -> TermList .
op emptyPos : -> Position .
op pos : Position Position -> Position [assoc] .
op nullPos : -> Position .
op emptyPosSet : -> PositionSet .
op posSet : PositionSet PositionSet -> PositionSet
            [assoc comm id: emptyPosSet] .
op getSubterm : Term Position -> Term .
op getSubtermAux : TermList Position -> Term .
op replace : Term Term Position -> Term .
op replaceAux : TermList Term Position -> Term .
op allNonVarPos : VarDeclSet Term -> PositionSet .
op allNonVarPosAux : VarDeclSet Term Position PositionSet -> PositionSet .
op nextPosition : Term Position -> Position .
op nextPositionUp : Term Position -> Position .
op allCritPairs : VarDeclSet Rule Rule PositionSet -> EquationSet .
op makeCritPair : Term Term Term Position UnifPair -> Equation .
op unifPair : VarDeclSet EquationCSet Substitution -> UnifPair .
op failure : -> Substitution .
op break : TermList TermList -> EquationCSet .
op occurs? : Qid TermList -> Bool .
op substitute : TermList Substitution -> TermList .
op substituteEquationCSet : EquationCSet Substitution -> EquationCSet .
op substituteSubstitution : Substitution Substitution  -> Substitution .
op complPair :  EquationSet Module -> ComplPair .
op convert : Module -> Module .
```

```
var B B' : Bool .
var N  M : MachineInt .
var QI F G X Y S L L' : Qid .
var T T' T1 T1' : Term .
var TL TL' : TermList .
var P : Position .
var Ps : PositionSet .
var EqsC : EquationCSet .
var SB SB' : Substitution .
var ImportL : ImportList .
var SortD : SortDecl .
var SubsortDs : SubsortDeclSet .
var OpDl : OpDeclList .
var VarDs : VarDeclSet .
var MembAxs : MembAxSet .
vars Eqs Equations : EquationSet .
vars Rls Rules : RuleSet .
var RewriteTh : Module .
```

As already mentioned, Boolean functions like `and`, `or`, and `nor` are not built-in functions in the current version of Maude and therefore need to be explicitly defined.

```
eq and(false, B) = false .
eq and(true, B) = B .
eq or(true, B) = true .
eq or(false, B) = B .
eq nor(B, B') = or(B, B') == false .
```

Let N_+^* be the set of sequences of positive integers, with Λ the empty sequence in N_+^*, and $\_ \cdot \_$ the concatenation operation on sequences. We call the members of N_+^* *positions*, and denote them by p, p', p'' and so on.

Given a signature Σ, a set of variables X, and a term $t \in T_\Sigma(X)$ we can define its *set of positions* $\mathcal{P}(t) \subseteq N_+^*$ and *the subterm* $t|_p$ *of* t *at position* p, as follows:

1. if $t = x \in X$, then $\mathcal{P}(t) = \{\Lambda\}$ and $t|_\Lambda = t$.
2. if $t = f(t_1, \ldots, t_n)$, then $\mathcal{P}(t) = \{\Lambda\} \cup \{i \cdot p) \mid i \leq n, p \in \mathcal{P}(t_i)\}$, $f(t_1, \ldots, t_n)|_\Lambda = f(t_1, \ldots, t_n)$, and $f(t_1, \ldots, t_n)|_{i \cdot p} = t_i|_p$.

Finally, a the term t with its subterm $t|_p$ replaced by the term t' is denoted $t[t']_p$. In our specification, `pos` is the constructor symbol for positions, and `emptyPos` is the empty sequence of positions.

The function `getSubterm`(t, p) returns $t|_p$, if $p \in \mathcal{P}(t)$; otherwise, it returns `error*`.

```
eq getSubterm(T, emptyPos) = T .
```

```
eq getSubterm(F, N) = error* .
eq getSubterm(F[TL], N) = getSubtermAux(TL, N) .

eq getSubtermAux((T, TL), N) =
   if N == 1 then T else getSubtermAux(TL, (N - 1)) fi .
eq getSubtermAux(T, N) =
   if N == 1 then T else error* fi .

eq getSubterm(F, pos(N, P)) = error* .
eq getSubterm(F[TL], pos(N, P)) =
   getSubtermAux(TL, pos(N, P)) .

eq getSubtermAux((T, TL'), pos(N, P)) =
   if N == 1 then getSubterm(T, P)
   else getSubtermAux(TL', pos((N - 1), P)) fi  .
eq getSubtermAux(T, pos(N, P)) =
   if N == 1 then getSubterm(T, P) else error* fi  .
```

The function replace(t, t', p) returns the term $t[t']_p$.

```
eq replace(T, T', emptyPos) = T' .

eq replace(F, T', N) = error* .
eq replace(F[TL], T', N) = F[replaceAux(TL, T', N)] .

eq replaceAux((T, TL), T', N) =
   if N == 1 then (T', TL)
   else (T, replaceAux(TL, T', (N - 1))) fi .
eq replaceAux(T, T', N) =
   if N == 1 then T' else error* fi .

eq replace(F, T', pos(N, P)) = error* .
eq replace(F[TL], T', pos(N, P)) =
   F[replaceAux(TL, T', pos(N, P))] .

eq replaceAux((T, TL), T', pos(N, P)) =
   if N == 1 then (replace(T, T', P), TL)
   else (T, replaceAux(TL, T', pos((N - 1), P))) fi .
eq replaceAux(T, T', pos(N, P)) =
   if N == 1 then replace(T, T', P)
   else error* fi .
```

The function `isVar?`(X, t) is a Boolean predicate that returns `true` when t is a variable in the set X; otherwise, it returns `false`.

```
eq isVar?(emptyVarDeclSet, T) = false .
eq isVar?(varDeclSet(varDecl(X, S), VarDs), F) =
   if X == F then true else isVar?(VarDs, F) fi  .
eq isVar?(varDeclSet(varDecl(X, S), VarDs), F[TL]) = false .
```

The function `occurs?`$(x, \vec{t})$ is a Boolean predicate that returns `true` when the variable x occurs in any of the terms in the term list $\vec{t}$; otherwise, it returns `false`.

```
eq occurs?(X, F) = X == F .
eq occurs?(X, F[TL]) = occurs?(X, TL) .
eq occurs?(X, (T, TL)) =
   if occurs?(X, T) then true else occurs?(X, TL) fi .
```

A substitution is uniquely defined by a mapping from variables to terms, and is written out as $\{x_1 \leftarrow s_1, \ldots, x_n \leftarrow s_n\}$. The function `substitute`$(t, \sigma)$ returns $t\sigma$, for σ a set of assignments $\{x_1 \leftarrow s_1, \ldots, x_n \leftarrow s_n\}$, where, of course, the x_i must be distinct.

```
eq substitute(T, emptySubstitution) = T .
eq substitute(X, ((Y <- T); SB)) =
   if X == Y then T else substitute(X, SB) fi .
eq substitute(F[TL], SB) = F[substitute(TL, SB)] .
eq substitute((T, TL), SB) = (substitute(T, SB), substitute(TL, SB)) .
```

An equation $s = t$ has a *solution* σ if $s\sigma = t\sigma$. When a solution exists, the least solution with respect to subsumption (see [6]) is called the *most general unifier*. In [7, 6], unification is presented as a step-by-step process of transforming multisets of equations until a "solved form" is obtained from wich the most general unifier can be extracted when the original equation is solvable. A *solved form* is a set of equations $\{x_1 = s_1, \ldots, x_n = s_n\}$ such that the x_i are distinct and no x_i is a variable in any s_j. Then, the most general unifier is the substitution $\{x_1 \leftarrow s_1, \ldots, x_n \leftarrow s_n\}$.

We specify now the set of transformation rules MM (see [6]) operating on expressions of the form `unifPair`(X, EC, S) with X a set of variables, EC a set of equations and S a set of assignments, where EC contains the equations yet to be solved and S the partial solution. Notice that the equations EC are unoriented as opposed to the equations in a system module $T = (\Sigma, E, R)$ that are always oriented. For that reason, we introduce the constructor `eqc`, that should be declared commutative. The current version of Maude does not support yet rewriting module commutativity. However, the effect of rewriting modulo commutativity can be obtained by declaring `eqc` associative and commutative. This solution forces us to declase the sort `EquationC` a subsort of the sort `Term`. This awkward subsort declaration will be removed from our specification as soon as rewriting modulo commutativity is supported in Maude.

```
*** Delete:
eq unifPair(VarDs, equationCSet(eqc(T, T), EqsC), SB) =
```

```
   unifPair(VarDs, EqsC, SB) .

*** Decompose:
eq unifPair(VarDs, equationCSet(eqc(F[TL], F[TL']), EqsC), SB) =
   unifPair(VarDs, equationCSet(break(TL, TL'), EqsC), SB) .

eq break(T, T') = eqc(T, T') .
eq break((T, TL), (T', TL')) = equationCSet(eqc(T, T'), break(TL, TL')) .

*** Conflict:
ceq unifPair(VarDs, equationCSet(eqc(F, G), EqsC), SB) =
    unifPair(VarDs, emptyEquationCSet, failure)
    if and(nor(isVar?(VarDs, F), isVar?(VarDs, G)), F =/= G) .
ceq unifPair(VarDs, equationCSet(eqc(G, F[TL]), EqsC), SB) =
    unifPair(VarDs, emptyEquationCSet, failure)
    if isVar?(VarDs, G) == false .
ceq unifPair(VarDs, equationCSet(eqc(F[TL], G[TL']), EqsC), SB) =
    unifPair(VarDs, emptyEquationCSet, failure) if F =/= G .

*** Merge:
ceq unifPair(VarDs, equationCSet(eqc(X, T), eqc(X, T'), EqsC), SB) =
    unifPair(VarDs, equationCSet(eqc(X, T), eqc(T, T'), EqsC), SB)
    if and(isVar?(VarDs, X),
           greaterEquationC(VarDs, eqc(X, T'), eqc(T, T'))) .

eq greaterEquationC(VarDs, eqc(T, T'), eqc(T1, T1')) =
   or((max(size(VarDs, T), size(VarDs, T')) >
       max(size(VarDs, T1), size(VarDs, T1'))),
      and((max(size(VarDs, T), size(VarDs, T')) ==
               max(size(VarDs, T1), size(VarDs, T1'))),
              ((max(size(VarDs, T), size(VarDs, T')) -
                  min(size(VarDs, T), size(VarDs, T'))) >
               (max(size(VarDs, T1), size(VarDs, T1')) -
                  min(size(VarDs, T1), size(VarDs, T1')))))) .

eq size(VarDs, F) = if isVar?(VarDs, F) == true then 0 else 1 fi .
eq size(VarDs, F[TL]) = (1 + size(VarDs, TL)) .
eq size(VarDs, (T, TL)) = (size(VarDs, T) + size(VarDs, TL)) .

eq max(N, M) = if N > M then N else M fi .
eq min(N, M) = if N < M then N else M fi .

*** Check:
ceq unifPair(VarDs, equationCSet(eqc(X, T), EqsC), SB) =
    unifPair(VarDs, emptyEquationCSet, failure)
    if and(isVar?(VarDs, X), occurs?(X, T), X =/= T) .
```

```
*** Eliminate:
ceq unifPair(VarDs, equationCSet(eqc(X, T), EqsC), SB) =
    unifPair(VarDs, substituteEquationCSet(EqsC, (X <- T)),
             (substituteSubstitution(SB, (X <- T)); (X <- T)))
      if and(isVar?(VarDs, X), occurs?(X, T) == false) .
```

The functions substituteEquationCSet(EC, σ) and substituteSubstitution(P, σ) extend the function substitution to sets of equations and sets of assignments.

```
eq substituteEquationCSet(emptyEquationCSet, SB) = emptyEquationCSet .
eq substituteEquationCSet(equationCSet(eqc(T, T'), EqsC), SB) =
   equationCSet(eqc(substitute(T, SB), substitute(T', SB)),
                substituteEquationCSet(EqsC, SB)) .

eq substituteSubstitution(emptySubstitution, SB) = emptySubstitution .
eq substituteSubstitution(((X <- T); SB'), SB) =
     ((X <- substitute(T, SB)); substituteSubstitution(SB', SB)) .
```

Starting with unifPair(X, eqc(s, t), emptySubstitution) and using the unification (oriented) equations repeatedly until none is applicable results in unifPair(X, emptyEquationCSet, failure) iff $s = t$ has no solution, or else it reduces to unifPair(X, emptyEquationCSet, $\{x_1 \leftarrow s_1, \ldots, x_n \leftarrow s_n\}$), where $\{x_1 \leftarrow s_1, \ldots, x_n \leftarrow s_n\}$ is a most general unifier of s and t.

The function allNonVarPos(X, t) returns the set $\mathcal{P}_X(t) \subseteq \mathcal{P}(t) = \{p \mid t|_p \notin X\}$ of nonvariable positions of t.

```
eq allNonVarPos(VarDs, T) =
   posSet(emptyPos, allNonVarPosAux(VarDs, T, emptyPos, emptyPosSet)) .

eq allNonVarPosAux(VarDs, T, P, Ps) =
   if nextPosition(T, P) == nullPos
   then Ps
   else (if isVar?(VarDs, getSubterm(T, nextPosition(T, P))) == true
         then allNonVarPosAux(VarDs, T, nextPosition(T, P), Ps)
         else allNonVarPosAux(VarDs, T, nextPosition(T, P),
                              posSet(nextPosition(T, P), Ps)) fi) fi .

eq pos(emptyPos, P) = P .

eq nextPosition(T, P) =
   if getSubterm(T, pos(P, 1)) == error*
   then nextPositionUp(T, P)
   else pos(P, 1) fi .

eq nextPositionUp(T, emptyPos) = nullPos .
```

```
eq nextPositionUp(T, N) =
   if getSubterm(T, (N + 1)) == error*
   then nullPos
   else (N + 1) fi .

eq nextPositionUp(T, pos(P, N)) =
   if getSubterm(T, pos(P, (N + 1))) == error*
   then nextPositionUp(T, P)
   else pos(P, (N + 1))  fi .
```

An inference rule in KBC is specified as a rewrite rule between pairs `complPair`(E, T), where E is (the representation of) a set of equations and T is (the representation of) a system module corresponding to a rewrite theory $(\Sigma, \emptyset, R)$ and with a set of variables X declared. We assume that for any equation $t = t' \in E$, $t, t' \in T_\Sigma(X)$. Regarding the system module T, we assume the following:

1. the signature Σ is unsorted, all the symbols $\sigma \in \Sigma$ are free (no attributes are allowed), and for each variable x that appears in a rule in R, there is in the set X of declared variables a variable x^* that does not appear in any rule;
2. all rules in R appear duplicated, each of them labelled with a label `l`n, $1 \leq n \leq \mathrm{card}(R)/2$ which is the same for a rule and its duplicate;
3. finally, all the rules are unconditional and their left hand side is not a variable.

The rule `[simplify]` specifies the rule of inference **Simplify** in KB. It rewrites both sides of an equation $t = t'$. Notice that we use `meta-reduce` to simplify a term. Since, `meta-reduce`(T, t) simplifies t using only the equations in T, we need to convert the rules of T into equations. In a future version of Maude there will be a function `meta-rewrite`(T, t) that simplifies using rules as well (following a default interpreter), making the auxiliary function `convert`(T) that transforms a system module into a functional module unnecessary.

```
crl [simplify]:
     complPair(equationSet(eq(T, T'), Eqs), RewriteTh) =>
     complPair(equationSet(
                 eq(meta-reduce(convert(RewriteTh), T),
                    meta-reduce(convert(RewriteTh), T')), Eqs),
               RewriteTh)
     if or(meta-reduce(convert(RewriteTh), T) =/= T,
           meta-reduce(convert(RewriteTh), T') =/= T') .
```

The function `convert`(T) transforms a system module $T = (\Sigma, E, R)$ into a functional module $T' = (\Sigma, E \cup R^=)$, where $R^= = \{t = t' \mid t \longrightarrow t' \text{ in } R\}$.

```
eq convert(mod(QI, ImportL, SortD, SubsortDs, OpDl, VarDs, MembAxs,
               Equations, ruleSet(Rules, rl(L, T, T')))) =
   convert(mod(QI, ImportL, SortD, SubsortDs, OpDl, VarDs, MembAxs,
               equationSet(Equations, eq(T, T')), Rules)) .
eq convert(mod(QI, ImportL, SortD, SubsortDs, OpDl, VarDs, MembAxs,
               Equations, emptyRuleSet)) =
           fmod(QI, ImportL, SortD, SubsortDs, OpDl, VarDs, MembAxs,
               Equations) .
```

The rule `[delete]` specifies the rule of inference **Delete** in KB. It removes a trivial equation.

```
rl [delete]:
    complPair(equationSet(eq(T, T), Eqs), RewriteTh) =>
    complPair(Eqs, RewriteTh) .
```

If $l \longrightarrow r$ and $s \longrightarrow t$ are two rewrite rules (with variables that are distinct), p is the position of a nonvariable subterm of s, and μ is a most general unifier of $s|_p$ and l, then the equation $t\mu = s\mu[r\mu]_p$ is a *critical pair* formed from those two rules.

The rule `[deduce]` specifies the rule of inference **Deduce** in KB. It selects two rules $l \longrightarrow r$ and $s \longrightarrow t$ (not necessarily distinct) in T, and adds to E all the critical pairs formed from them. To determine overlaps, the variables in $s \longrightarrow t$ are renamed, so that they are disjoint.

```
rl [deduce]:
    complPair(Eqs, mod(QI, ImportL, SortD, SubsortDs,
                       OpDl, VarDs, MembAxs, emptyEquationSet,
                       ruleSet(rl(L, T, T'), rl(L', T1, T1'), Rls))) =>
    complPair(equationSet(Eqs,
                 allCritPairs(VarDs, rl(L, T, T'),
                              rl(L',rename(VarDs, T1), rename(VarDs, T1')),
                              allNonVarPos(VarDs, rename(VarDs, T1)))),
              mod(QI, ImportL, SortD, SubsortDs,
                  OpDl, VarDs, MembAxs, emptyEquationSet,
                  ruleSet(rl(L, T, T'), rl(L', T1, T1'), Rls))) .
```

The function `rename`(X, t) renames any variable $x \in X$ in t by x^*.

```
eq rename(VarDs, F[TL]) = F[rename(VarDs, TL)] .
eq rename(VarDs, (T, TL)) = (rename(VarDs, T), rename(VarDs, TL)) .
eq rename(VarDs, X) =
   if isVar?(VarDs, X) == true then conc(X, '*) else X fi .
```

The function `allCritPairs`$(X, l \longrightarrow r,\ s \longrightarrow t,\ \mathcal{Q})$, where $\mathcal{Q} \subseteq \mathcal{P}_X(t)$, returns the set of critical pairs generated by the overlaps of l over s at all positions $p \in \mathcal{Q}$. The function `allCritPair` assumes that the set of variables in $l \longrightarrow r$ and $s \longrightarrow t$ are disjoint.

```
eq allCritPairs(VarDs, rl(L, T, T'), rl(L', T1, T1'), posSet(P, Ps)) =
   equationSet(
      makeCritPair(T1, T1',T', P,
            unifPair(VarDs, eqc(T, getSubterm(T1, P)), emptySubstitution)),
      allCritPairs(VarDs,  rl(L, T, T'), rl(L', T1, T1'), Ps)) .
eq allCritPairs(VarDs, rl(L, T, T'), rl(L', T1, T1'), emptyPosSet) =
   emptyEquationSet .
```

When there is an overlap, the function `makeCritPair` generates the corresponding critical pair.

```
eq makeCritPair(T1, T1', T', P, unifPair(VarDs, EqsC, SB)) =
   if and((EqsC == emptyEquationCSet), (SB == failure))
   then emptyEquationSet
   else eq(substitute(T1', SB),
           replace(substitute(T1, SB), substitute(T', SB), P)) fi .
```

4.2 The Knuth-Bendix Checker as a Strategy

```
op checker : MachineInt -> Strategy .
var NumberRules : MachineInt .

eq checker(NumberRules) =
   and(set(I, '1),
   and(set(LI, 'index['', I]),
   and(while('_<=_[I, index(', NumberRules)],
            and(set(J, '1),
            and(set(LJ, 'index['', J]),
            and(while('_<=_[J, index(', NumberRules)],
                      and(applyWithSubst('deduce, (('L <- LI); ('L' <- LJ))),
                      and(set(J, '_+_[J, '1]),
                          set(LJ, 'index['', J])))),
            and(set(I, '_+_[I, '1]),
                set(LI, 'index['', I])))))),
   and(iterate(apply('delete)),
   and(iterate(apply('simplify)),
       iterate(apply('delete)))))) .
```

5 An Inductive Theorem Prover

After giving the inference rules, we give the Maude code implementing it and a few sample proof strategies.

5.1 Inductive Inference Rules

The following rules 1–6 provide a sound inference system for proving that a universally quantified atomic formula of the form $(\forall x)t = t'$, or of the form $(\forall x)t : s$

is an inductive consequence of a given membership algebra specification. Since order-sorted algebra can be embedded in membership algebra in a conservative way, they remain sound for proving inductive theorems of order-sorted functional specifications in Cafe.

1. **Test Set Induction**

$$\frac{M \vdash_{\text{ind}} (\forall X).p}{M \vdash_{\text{ind}} \mathit{step}(p, X \setminus \{x\}, x, s, u_1) \cdots M \vdash_{\text{ind}} \mathit{step}(p, X \setminus \{x\}, x, s, u_n)}$$

   where $x \in X$ has sort s and $\{u_1, \ldots, u_n\}$ is the test set for sort s and

$$\mathit{step}(p, X, x, s, u) =$$
$$(\forall \mathit{vars}(u)).\left(\left(\bigwedge_{y:s' \in \mathit{vars}(u),\ s' \leq s} (\forall X).p[y/x]\right) \Rightarrow (\forall X).p[u/x]\right)$$

2. **Proof in Variety**

$$\frac{M \vdash_{\text{ind}} p}{M \vdash p}$$

3. **Constants Lemmma**

$$\frac{M \vdash (\forall \{x_1, \ldots, x_n\}).p}{M \cup \{\texttt{op } c_1 \texttt{ :-> } s_1 . \cdots \texttt{op } c_n \texttt{ :-> } s_n .\} \vdash p[c_1/x_1, \ldots, c_n/x_n]}$$

   where x_i has sort s_i and $c_1, \ldots, c_n$ do not occur in M.

4. $\Rightarrow$ **Elimination**

$$\frac{M \vdash (\forall X_1).e_1 \wedge \cdots \wedge (\forall X_{n-1}).e_{n-1} \Rightarrow (\forall X_n).e_n}{M \cup \{\texttt{eq } e_1 . \cdots \texttt{eq } e_{n-1} .\} \vdash (\forall X_n).e_n}$$

   if for $i \in \{1, \ldots, n\}$, $\mathit{vars}(e_i) \subseteq X_i$.

$$\frac{M \vdash (\forall X_1).m_1 \wedge \cdots \wedge (\forall X_{n-1}).m_{n-1} \Rightarrow (\forall X_n).m_n}{M \cup \{\texttt{mb } m_1 . \cdots \texttt{mb } m_{n-1} .\} \vdash (\forall X_n).m_n}$$

   if for $i \in \{1, \ldots, n\}$, $\mathit{vars}(m_i) \subseteq X_i$.

5. **Proof by Rewriting for Ground Atoms**

$$\frac{M \vdash t = t'}{t \downarrow_M \equiv t' \downarrow_M} \quad \text{if } t,\ t' \text{ are ground.}$$

$$\frac{M \vdash t : s}{t \downarrow_M : s} \quad \text{if } t \text{ is ground.}$$

6. **Equational and Membership Lemmas**

$$\frac{M \vdash_{\text{ind}} p}{M \vdash_{\text{ind}} e \qquad M \cup \{\texttt{eq}\ e.\} \vdash_{\text{ind}} P}$$

$$\frac{M \vdash_{\text{ind}} m}{M \vdash_{\text{ind}} e \qquad M \cup \{\texttt{mb}\ m.\} \vdash_{\text{ind}} P}$$

The module ITP below is the specification in Maude of the inference system just introduced. The rewrite rules in the module mirror very closely the inference sytem described above. The remaining of the specification is a collection of equations defining the semantics of several auxiliary functions implicit in the inference rules.

```
mod ITP is
protecting META-LEVEL .
sorts IntString Equality Formula TermSet Test TestSet Goal GoalSet .

subsort MachineInt < IntString .
subsort Goal < GoalSet .
subsort Equality < Formula .
subsort Term < TermSet .
subsort Test < TestSet .

op trueFormula : -> Formula .
op equality : Term Term -> Equality .
op sortP : Term Qid -> Formula .
op implication : Formula Formula -> Formula .
op conjunction : Formula Formula -> Formula [assoc comm] .
op VQuantification : QidSet Formula -> Formula .
op proveinInitial : IntString Module Formula TestSet -> Goal .
op proveinVariety : IntString Module Formula -> Goal .
op equal : Term Term -> Goal .
op emptyIntString : -> IntString .
op intString : IntString IntString -> IntString [assoc] .
op emptyTermSet : -> TermSet .
op termSet : TermSet TermSet -> TermSet [assoc comm id: emptyTermSet] .
op test : Qid TermSet -> Test .
op emptyTestSet : -> TestSet .
op testSet : TestSet TestSet -> TestSet [assoc comm id: emptyTestSet] .
op emptyGoalSet : -> GoalSet .
op goalSet : GoalSet GoalSet -> GoalSet [assoc comm id: emptyGoalSet] .
op getVars : Module -> VarDeclSet .
op getSubsorts : Module -> SubsortDeclSet .
op findSort : Qid VarDeclSet -> Qid .
op makeNewGoals : IntString Module QidSet Qid Term
                  Term TermSet TestSet -> GoalSet .
```

```
op makeNewGoalsMb : IntString Module QidSet Qid Term Term
                    TermSet TestSet -> GoalSet .
op makeConjunctionHypoth : QidSet Term Term Qid QidSet -> Formula .
op makeConjunctionHypothMb : QidSet Term Term Qid QidSet -> Formula .
op vars : Term VarDeclSet -> QidSet .
op varsSubsort : QidSet VarDeclSet Qid SubsortDeclSet -> QidSet .
op subsort? : Qid Qid SubsortDeclSet -> Bool .
op substitute : TermList Substitution -> TermList .
op addNewConstants : QidSet Module -> Module .
op addToModule : Formula Module -> Module .
op varsToNewConstants : QidSet Formula -> Formula .
op varsToNewConstantsAux : QidSet TermList -> TermList .
op and : Bool Bool -> Bool [assoc comm] .
op inSet : Qid QidSet -> Bool .

var B : Bool .
var I : MachineInt .
var Idx : IntString .
vars QI X Y F S S_1 S_2 S_3 S_4 : Qid .
var Xs Ys : QidSet .
vars T T1 T2 : Term .
var TL : TermList .
var Ts : TermSet .
var ImportL : ImportList .
var Assg : Assignment .
var SortD : SortDecl .
var SubsortDs : SubsortDeclSet .
var OpDl : OpDeclList .
var VarDs : VarDeclSet .
var MembAxs : MembAxSet .
var Equations : EquationSet .
var FunctTh : Module .
var TSs : TestSet .
vars Alpha Beta : Formula .
var G : GoalSet .

crl [testSetInduction]:
    goalSet(proveinInitial(Idx, FunctTh,
                VQuantification(qidSet(X, Xs), equality(T1, T2)),
                testSet(test(S, Ts), TSs)), G) =>
    goalSet(makeNewGoals(intString(Idx, 1), FunctTh, Xs, X, T1, T2, Ts,
                         testSet(test(S, Ts), TSs)), G)
     if findSort(X, getVars(FunctTh)) == S .

crl [testSetInduction]:
    goalSet(proveinInitial(Idx, FunctTh,
```

```
                 VQuantification(qidSet(X, Xs), sortP(T, S_1)),
                 testSet(test(S, Ts), TSs)), G) =>
    goalSet(makeNewGoalsMb(intString(Idx, 1), FunctTh, Xs, X, T, S_1, Ts,
            testSet(test(S, Ts), TSs)), G)
       if findSort(X, getVars(FunctTh)) == S .

rl [inVariety]:
   goalSet(proveinInitial(Idx, FunctTh, Alpha, TSs), G)=>
   goalSet(proveinVariety(Idx, FunctTh, Alpha), G) .

rl [constantsLemma]:
   goalSet(proveinVariety(Idx, FunctTh,
                VQuantification(Xs, Alpha)), G) =>
   goalSet(proveinVariety(Idx,addNewConstants(Xs, FunctTh),
                varsToNewConstants(Xs, Alpha)), G) .

rl [implicationElimination]:
   goalSet(proveinVariety(Idx, FunctTh,
                implication(Alpha, Beta)), G) =>
   goalSet(proveinVariety(Idx,addToModule(Alpha, FunctTh),
                Beta), G) .

rl [byRewriting]:
   goalSet(proveinVariety(Idx, FunctTh, equality(T1, T2)), G) =>
   goalSet(equal(meta-reduce(FunctTh, T1), meta-reduce(FunctTh, T2)), G) .

rl [byRewriting]:
   goalSet(proveinVariety(Idx, FunctTh, sortP(T1, S)), G) =>
   goalSet(equal(meta-reduce(FunctTh, membPred(T1, S)), 'true), G) .

rl [lemmaIntroduction]:
   goalSet(proveinInitial(Idx, FunctTh, Alpha, TSs), G) =>
   goalSet(proveinInitial(intString(Idx, 1),
                        addToModule(Beta, FunctTh), Alpha, TSs),
          proveinInitial(intString(Idx, 2), FunctTh, Beta, TSs), G) .

eq equal(T, T) = emptyGoalSet .

eq implication(trueFormula, Beta) = Beta .
eq conjunction(Alpha, trueFormula) = Alpha .
eq VQuantification(emptyQidSet, Alpha) = Alpha .
eq VQuantification(Xs, VQuantification(Ys, Alpha)) =
   VQuantification(qidSet(Xs, Ys), Alpha) .

eq makeNewGoals(intString(Idx, I), FunctTh, Xs, X, T1, T2,
               termSet(T, Ts), TSs) =
```

```
   goalSet(proveinInitial(intString(Idx, I),
                FunctTh,
                VQuantification(vars(T, getVars(FunctTh)),
                    implication(
                       makeConjunctionHypoth(Xs, T1, T2, X,
                           varsSubsort(vars(T, getVars(FunctTh)),
                                       getVars(FunctTh),
                                       findSort(X, getVars(FunctTh)),
                                       getSubsorts(FunctTh))),
                       VQuantification(Xs,
                          equality(substitute(T1, (X <- T)),
                                   substitute(T2, (X <- T)))))),
                TSs),
    makeNewGoals(intString(Idx, _+_(1, I)), FunctTh, Xs, X,
               T1, T2, Ts, TSs)) .

eq makeNewGoals(intString(Idx, I), FunctTh, Xs, X, T1, T2,
              emptyTermSet, TSs) = emptyGoalSet .

eq makeNewGoalsMb(intString(Idx, I), FunctTh, Xs, X, T1, S,
                termSet(T, Ts), TSs) =
   goalSet(proveinInitial(intString(Idx, I),
                FunctTh,
                VQuantification(vars(T, getVars(FunctTh)),
                    implication(
                       makeConjunctionHypothMb(Xs, T1, S, X,
                           varsSubsort(vars(T, getVars(FunctTh)),
                                       getVars(FunctTh),
                                       findSort(X, getVars(FunctTh)),
                                       getSubsorts(FunctTh))),
                       VQuantification(Xs,
                          sortP(substitute(T1, (X <- T)), S)))),
                TSs),
    makeNewGoalsMb(intString(Idx, _+_(1, I)), FunctTh, Xs, X,
                T1, S, Ts, TSs)) .

eq makeNewGoalsMb(intString(Idx, I), FunctTh, Xs, X, T1, S,
                emptyTermSet, TSs) = emptyGoalSet .

eq getVars(fmod(QI, ImportL, SortD, SubsortDs,
              OpDl, VarDs, MembAxs, Equations)) = VarDs .
eq getSubsorts(fmod(QI, ImportL, SortD, SubsortDs,
             OpDl, VarDs, MembAxs, Equations)) = SubsortDs .

eq findSort(X, varDeclSet(varDecl(X, S), VarDs)) = S .

eq vars(X, varDeclSet(varDecl(Y, S), VarDs)) =
```

```
   if X == Y then X else vars(X, VarDs) fi .
eq vars(X, emptyVarDeclSet) = emptyQidSet .
eq vars(F[TL], VarDs) = vars(TL, VarDs) .
eq vars((T, TL), VarDs) = qidSet(vars(T, VarDs), vars(TL, VarDs)) .

eq varsSubsort(emptyQidSet, VarDs, S, SubsortDs) = emptyQidSet .
eq varsSubsort(X, VarDs, S, SubsortDs) =
   if subsort?(findSort(X, VarDs), S, SubsortDs) then X else emptyQidSet fi .

eq subsort?(S_1, S_2, subsortDeclSet(subsort(S_3, S_4), SubsortDs)) =
   if and((S_1 == S_3), (S_2 == S_4))
     then true
   else subsort?(S_1, S_2, SubsortDs) fi .
eq subsort?(S_1, S_2, emptySubsortDeclSet) = S_1 == S_2 .

eq addNewConstants(qidSet(X, Xs), fmod(QI, ImportL, SortD, SubsortDs, OpDl,
                   varDeclSet(varDecl(X, S), VarDs), MembAxs, Equations)) =
   addNewConstants(Xs, fmod(QI, ImportL, SortD, SubsortDs,
                   opDeclList(opDecl(conc(X, '*), nilQidList,
                                     S, emptyAttrSet), OpDl),
                   varDeclSet(varDecl(X, S), VarDs), MembAxs, Equations)) .
eq addNewConstants(emptyQidSet, fmod(QI, ImportL, SortD, SubsortDs, OpDl,
                   VarDs, MembAxs, Equations)) =
   fmod(QI, ImportL, SortD, SubsortDs, OpDl, VarDs, MembAxs, Equations) .

eq addToModule(VQuantification(Xs, equality(T1, T2)),
               fmod(QI, ImportL, SortD, SubsortDs,
                    OpDl, VarDs, MembAxs, Equations)) =
   fmod(QI, ImportL, SortD, SubsortDs, OpDl, VarDs, MembAxs,
        equationSet(eq(T1, T2), Equations)) .
eq addToModule(equality(T1, T2),
               fmod(QI, ImportL, SortD, SubsortDs,
                    OpDl, VarDs, MembAxs, Equations)) =
   fmod(QI, ImportL, SortD, SubsortDs, OpDl, VarDs, MembAxs,
        equationSet(eq(T1, T2), Equations)) .

eq addToModule(VQuantification(Xs, sortP(T1, S)),
               fmod(QI, ImportL, SortD, SubsortDs,
                    OpDl, VarDs, MembAxs, Equations)) =
   fmod(QI, ImportL, SortD, SubsortDs, OpDl, VarDs,
        membAxSet(mb(T1, S), MembAxs), Equations) .

eq addToModule(sortP(T1, S),
               fmod(QI, ImportL, SortD, SubsortDs,
                    OpDl, VarDs, MembAxs, Equations)) =
   fmod(QI, ImportL, SortD, SubsortDs, OpDl, VarDs,
        membAxSet(mb(T1, S), MembAxs), Equations) .
```

```
eq addToModule(conjunction(Alpha, Beta), FunctTh) =
   addToModule(Beta, addToModule(Alpha, FunctTh)) .

eq varsToNewConstants(emptyQidSet, Alpha) = Alpha .
eq varsToNewConstants(qidSet(X, Xs), implication(Alpha, Beta)) =
   implication(varsToNewConstants(qidSet(X, Xs), Alpha),
               varsToNewConstants(qidSet(X, Xs), Beta)) .
eq varsToNewConstants(qidSet(X, Xs), conjunction(Alpha, Beta)) =
   conjunction(varsToNewConstants(qidSet(X, Xs), Alpha),
               varsToNewConstants(qidSet(X, Xs), Beta)) .
eq varsToNewConstants(qidSet(X, Xs), VQuantification(Ys, Alpha)) =
   if inSet(X, Ys)
   then varsToNewConstants(Xs, VQuantification(Ys, Alpha))
   else varsToNewConstants(Xs, VQuantification(Ys,
            varsToNewConstants(X, Alpha))) fi .

eq varsToNewConstants(qidSet(X, Xs), equality(T1, T2)) =
   equality(varsToNewConstantsAux(qidSet(X, Xs), T1),
            varsToNewConstantsAux(qidSet(X, Xs), T2)) .
eq varsToNewConstants(qidSet(X, Xs), sortP(T1, S)) =
   sortP(varsToNewConstantsAux(qidSet(X, Xs), T1), S) .

eq varsToNewConstantsAux(Xs, F[TL]) = F[varsToNewConstantsAux(Xs, TL)] .
eq varsToNewConstantsAux(Xs, (T, TL)) =
    (varsToNewConstantsAux(Xs, T), varsToNewConstantsAux(Xs, TL)) .
eq varsToNewConstantsAux(Xs, Y) =
    if inSet(Y, Xs) then conc(Y, '*) else Y fi .

eq makeConjunctionHypoth(Xs, T1, T2, X, qidSet(Y, Ys)) =
   conjunction(VQuantification(Xs, equality(substitute(T1, (X <- Y)),
                                            substitute(T2, (X <- Y)))),
       makeConjunctionHypoth(Xs, T1, T2, X, Ys)) .

eq makeConjunctionHypoth(Xs, T1, T2, X, Y) =
  VQuantification(Xs, equality(substitute(T1, (X <- Y)),
                               substitute(T2, (X <- Y)))) .

eq makeConjunctionHypoth(Xs, T1, T2, X, emptyQidSet) = trueFormula .

eq makeConjunctionHypothMb(Xs, T1, S, X, qidSet(Y, Ys)) =
   conjunction(VQuantification(Xs, sortP(substitute(T1, (X <- Y)), S)),
   makeConjunctionHypothMb(Xs, T1, S, X, Ys)) .

eq makeConjunctionHypothMb(Xs, T1, S, X, Y) =
  VQuantification(Xs, sortP(substitute(T1, (X <- Y)), S)) .
```

```
eq makeConjunctionHypothMb(Xs, T1, S, X, emptyQidSet) = trueFormula .

eq substitute(T, emptySubstitution) = T .
eq substitute(X, ((Y <- T); SB)) = if X == Y then T else substitute(X, SB) fi .
eq substitute(F[TL], SB) = F[substitute(TL, SB)] .
eq substitute((T, TL), SB) = (substitute(T, SB), substitute(TL, SB)) .

eq inSet(X, qidSet(Y, Ys)) = if X == Y then true else inSet(X, Ys) fi .
eq inSet(X, emptyQidSet) = false .

eq and(B, true) = B .
eq and(B, false) = false .

eq intString(Idx, 0) = Idx .
endm
```

5.2 Some Proof Strategies

As an example, we give here the strategy expression definition of two useful proof strategies, a strategy `simplify` that tries to simplify the goals as much as possible, and a strategy `induction` that applies the induction rule and then simplifies the goals as much as possible.

```
op simplify : Term -> Strategy .
op induction : Term -> Strategy .

var I V : Term

eq simplify(I) =
   and(applyWithSubst('toVariety, ('Idx <- I)),
       orelse(apply('byRewriting),
              and(apply('constantsLemma),
                        orelse(apply('byRewriting),
                              and(apply('implicationElimination),
                                  and(orelse(apply('constantsLemma),
                                     idle),
                                  apply('byRewriting)))))) .
eq induction(X) =
   and(applyWithSubst('testSetInduction, ('X <- V)),
       iterate(simplify)) .
```

References

[1] L. Bachmair, N. Dershowitz, and J. Hsiang. Orderings for equational proofs. In *Proceedings of the Symposium on Logic in Computer Science*, pages 346–357. IEEE, June 1986.

[2] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *First International Workshop on Rewriting Logic*, volume 4, pages 65–89, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[3] M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, editor, *Proceedings of Reflection'96, San Francisco, California, April 1996*, pages 263–288. Xerox PARC, 1996.

[4] 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, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

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

[6] 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.

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

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

[9] 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.

[10] J. Meseguer and T. Winkler. Parallel programming in Maude. In *Research Directions in High-Level Parallel Programming Languages*, number 574 in Lecture Notes in Computer Science, pages 253–293. Springer-Verlag, 1992.