Design and Implementation of the Cafe Prover and Church-Rosser Checker Tools

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

March 1998

Contents

1 Introduction

The present document gives detailed explanations of our reflective design for the inductive theorem prover and the Church-Rosser checker that we have built as part of the Cafe project. The Maude code for each of the tools is also given and discussed in its entirety. Furthermore, examples illustrating the use of the tools and methodological suggestions drawn from the examples are also given.

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 and illustrated with examples 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 [4, 5] in the precise sense that there is a finitely presented rewrite theory U such that for any finitely presented rewrite theory T (including U itself) we have the following equivalence

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

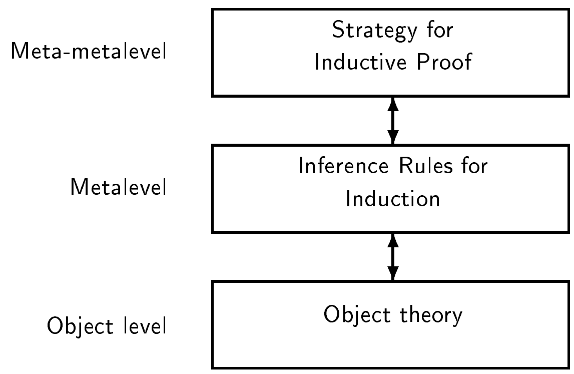
where $\overline{T}$ and $\overline{t}$ are terms representing T and t as data elements of U. Since U is representable in itself, we can achieve a "reflective tower" with an arbitrary number of levels of reflection, since we have

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

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` has sorts `Term` and `Module`, so that the representations $\overline{t}$ and $\overline{T}$ of a term t and a module T have sorts `Term` and `Module`, respectively. `META-LEVEL` has also functions `meta-reduce`($\overline{T}$, $\overline{t}$) and `meta-apply`($\overline{T}$, $\overline{t}$, $\overline{l}$, $\overline{\sigma}$, n) which 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 explained later, `META-LEVEL` also provides several other useful functions, including test for membership in a sort, least sort of a term, and several functions on the sorts of a module. 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 these tools has then a very simple design. Consider, for example, the design of the inductive theorem prover. Its purpose is to prove inductive properties of a Cafe functional module T, which has an initial algebra semantics. The theory T about which we want to prove inductive properties is at the object level. The rules of inference for induction can be naturally expressed as a rewrite theory $\mathcal{I}$. But since this rewrite theory uses T as a data structure—that is, it actually uses its representation $\overline{T}$—the theory $\mathcal{I}$ should be defined at the metalevel. Proving an inductive theorem for T corresponds to applying the rules in $\mathcal{I}$ with some *strategy*. But since the strategies for any rewrite theory belong to the metalevel of such a theory, and $\mathcal{I}$ 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 $\mathcal{I}$. This is illustrated by the following picture



The inductive Church-Rosser checker tool has a similar reflective design. Again, the module T, that we want to check is Church-Rosser, is at the object level. An inference system $\mathcal{C}$ for checking the Church-Rosser property uses $\overline{T}$ as a data structure and therefore is a rewrite theory at the metalevel. Since the checking process can be described in a purely functional way, there is no need in this case for an additional strategy layer at the meta-metalevel: two levels suffice.

1.1 Acknowledgments

We have benefited very much from discussions and positive suggestions from many of our fellow members in the Cafe Project, particularly from Kokichi Futatsugi, Ataru Nakagawa, Toshimi Sanada, Masaki Ishiguro, and Masayuki Fujita. Thanks to their efforts, our tools have been successfully integrated within the Cafe environment.

The theoretical foundations of, and automated deduction techniques for, the tools have benefited very much from our ongoing collaboration with Adel Bouhoula and Jean-Pierre Jouannaud on theorem proving techniques for membership equational logic [1, 2], and from very fruitful discussions with them during the Cafe project.

The implicit induction approach adopted for the inductive theorem prover extends in a reflective way a similar explicit induction approach to equational theorem proving in OBJ proposed by Joseph Goguen [7, 8]. We have benefited considerably from Joseph Goguen advice and from the OBJ experience in the design of the inductive theorem prover.

2 A Reflective Kernel in Maude

Maude is a logical language based on rewriting logic [14, 16, 15]. An introduction to Maude and its interpreter implementation can be found in [3]. 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 [4, 5], 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 Sort 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 Qid < Sort .
  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) .
op 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 .
op ' : -> Qid .
op conc : Qid Qid -> Qid .
op index : Qid MachineInt -> Qid .
op protecting : Qid -> Import .
op nilImportList : -> ImportList .
op importList : ImportList ImportList -> ImportList
      [assoc id: nilImportList] .
op _[_] : Qid TermList -> Term .
op _,_ : TermList TermList -> TermList [assoc] .
op membPred : Term Qid -> Term .
op lazyMembPred : Term Qid -> Term .
op nilQidList : -> QidList .
op qidList : QidList QidList -> QidList [assoc id: nilQidList] .
op emptyQidSet : -> QidSet .
```

```
op qidSet : QidSet QidSet -> QidSet [assoc comm id: emptyQidSet] .
op nilMachineIntList : -> MachineIntList .
op machineIntList : MachineIntList MachineIntList -> MachineIntList
      [assoc id: nilMachineIntList] .
op sort : QidSet -> SortDecl .
op subsort : Qid Qid -> SubsortDecl .
op emptySubsortDeclSet : -> SubsortDeclSet .
op subsortDeclSet : SubsortDeclSet SubsortDeclSet -> SubsortDeclSet
      [assoc comm id: emptySubsortDeclSet] .
op assoc : -> Attr .
op comm : -> Attr .
op idem : -> Attr .
op id : Term -> Attr .
op leftId : Term -> Attr .
op rightId : Term -> Attr .
op strat : MachineIntList -> Attr .
op emptyAttrSet : -> AttrSet .
op attrSet : AttrSet AttrSet -> AttrSet
      [assoc comm id: emptyAttrSet] .
op opDecl : Qid QidList Qid AttrSet -> OpDecl .
op nilOpDeclList : -> OpDeclList .
op opDeclList : OpDeclList OpDeclList -> OpDeclList
      [assoc id: nilOpDeclList] .
op varDecl : Qid Qid -> VarDecl .
op emptyVarDeclSet : -> VarDeclSet .
op varDeclSet : VarDeclSet VarDeclSet -> VarDeclSet
      [assoc comm id: emptyVarDeclSet] .
op eq : Term Term -> Equation .
op ceq : Term Term Term Term -> Equation .
op emptyEquationSet : -> EquationSet .
op equationSet : EquationSet EquationSet -> EquationSet
      [assoc comm id: emptyEquationSet] .
op rl : Qid Term Term -> Rule .
op crl : Qid Term Term Term Term -> Rule .
op emptyRuleSet : -> RuleSet .
op ruleSet : RuleSet RuleSet -> RuleSet
      [assoc comm id: emptyRuleSet] .
op mb : Term Qid -> MembAx .
op cmb : Term Qid Term Term -> MembAx .
op emptyMembAxSet : -> MembAxSet .
op membAxSet : MembAxSet MembAxSet -> MembAxSet
      [assoc comm id: emptyMembAxSet] .
op fmod : Qid ImportList SortDecl SubsortDeclSet OpDeclList
      VarDeclSet MembAxSet EquationSet -> Module .
op mod : Qid ImportList SortDecl SubsortDeclSet OpDeclList
      VarDeclSet MembAxSet EquationSet RuleSet -> Module .
op error* : -> Term .
op _<-_ : Qid Term -> Assignment .
op emptySubstitution : -> Substitution .
op _;_ : Substitution Substitution -> Substitution
```

```
        [assoc comm id: emptySubstitution] .
  op _,_ : Term Substitution -> ResultPair .
  op errorSort : QidSet -> Sort .
  op meta-reduce : Module Term -> Term .
  op meta-apply : Module Term Qid Substitution MachineInt -> ResultPair .
  op sortLeq : Module Sort Sort -> Bool .
  op leastSort : Module Term -> Sort .
  op lesserSorts : Module Sort -> QidSet .
  op glbSorts : Module Sort Qid -> QidSet .
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,...), lazyMembPred('+['s['0], '0], 'NzNat)) .
+> 'false
> meta-reduce(fmod('NAT,...), lazyMembPred('+['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 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 and 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 [5]). 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 $\overline{\overline{\overline{\texttt{s(0)}}}}$: Term that results from enconding $s(0)$ twice.

Regarding the operations on sorts, the functions included in the module META-LEVEL are sortLeq, leastSort, lesserSorts and glbSorts. At the meta-level, the sorts given by the user in his/her module are represented as quoted identifiers, that is, terms of sort Qid. However, the module META-LEVEL has also a sort Sort defined to be a supersort of Qid. Sorts not defined by the user, as for example the "error sorts" added by the system to complete each connected component[1] are in this sort Sort.

The function leastSort takes as arguments a module and a term and computes the least sort of that term in the module (see, e.g., [9]). This function can return an error sort not defined by the user, and that is why it is defined as

```
  op leastSort : Module Term -> Sort .
```

For example, we can compute the least sort of the term '+['N, 's['M]] in the previous module NAT as follows.

```
Maude> red leastSort(fmod('NAT, ...), '+['N, 's['M]]) .
result Qid: 'Nat
```

Given a module M with subsort relation $\leq_M$, and sorts $s, s' \in S$, where S is the set of sorts in M, the Boolean function sortLeq(M,s,s') is true if and only if $s \leq_M s'$. Note that the sorts passed to the function are of sort Sort.

```
Maude> red sortLeq(fmod('NAT, ...), 'Nat, 'NzNat) .
result Bool: false
```

Given a module M and a sort s, the function lesserSorts returns the set of sorts strictly smaller than s in M. For example:

```
Maude> red lesserSorts(fmod('NAT, ...), 'Nat) .
result Qid: 'NzNat
```

Finally, the function glbSorts takes two sorts and a module as input and computes the set of maximal lower bounds of the two sorts[2]. Note the asymmetry in the declaration of this function, having as arguments, together with the module, a sort of sort Sort and another of sort Qid. This asymmetry might be eliminated in a future version of the system. The actual declaration is:

[1]These error sorts are in fact the *kinds* of the membership equational logic specification implicitly defined by the user's specification [17]. Each kind contains all the well-sorted terms of the connected component, plus additional error terms that cannot be assigned a sort

[2]Of course, when this set is a singleton $\{s\}$, then s will be the greatest lower bound of the two sorts, thus the notation glbSorts. In subsequent discussions, when we speak of the "greatest lower bound" we will always in fact mean the more general notion of the set of maximal lower bounds.

```
  op glbSorts : Module Sort Qid -> QidSet .
```

As an example for the use of this function, let us see how to compute the greatest lower bound for sorts Nat and NzNat in the module NAT presented above.

```
Maude> red glbSorts(fmod('NAT, ...), 'Nat, 'NzNat) .
result Qid: 'NzNat
```

3 Internal Strategies in Maude

System modules in Maude are rewrite theories that do not need to be Church-Rosser and 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 [4, 5].

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, $i \in \mathrm{card}(P)$, 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 length 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 appropriate 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 The Inductive Theorem Prover

In this section we present the specification in Maude of an inductive theorem prover to prove theorems about the initial model of a functional module in Cafe or in Maude[3]. As in the case of the Church-Rosser checker tool, it is very natural to adopt a "reflective" design to specify in Maude an inductive theorem-prover. As mentioned before, the idea is that the functional module T about which we want to prove inductive theorems is at the object level; an inference system $\mathcal{I}$ for inductive proofs uses $\overline{T}^m$ as data and therefore can be specified as a rewrite theory at the metalevel; then, different *proof tactics* to guide the application of the rewrite rules that specify the inference rules in $\mathcal{I}$ are strategies that can be represented at the meta-metalevel.

We first introduce an inference system for inductive proofs, and we explain how it can be specified in Maude as a rewrite theory at the metalevel. Then, we discuss the representation of proof tactics at the meta-metalevel.

4.1 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.

1. **Test Set Induction**

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

[3]Cafe functional modules are order-sorted equational theories; Maude functional modules are equational theories in membership equational logic. Thanks to the conservative embedding of order-sorted equational logic into membership equational logic [17], Cafe modules are easily translated into its corresponding Maude modules. The extra generality of membership equational logic is exploited thanks to this translation, because the inductive theorem prover can then prove not only equalities $t = t'$, but also membership assertions $t : s$, which, as we shall see, are crucially needed to establish sufficient generation of a module (even if it is order-sorted) or to prove inductively that a module is ground Church-Rosser

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

$$step(p, X, x, s, u) =$$

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

2. **Proof in Variety**

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

3. **Constants Lemma**

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

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

4. **Implication 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 $i \in \{1, \ldots, n\}$, $vars(e_i) \subseteq X_i$.

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

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

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**

$$\frac{M \vdash_{\mathrm{ind}} p}{M \vdash_{\mathrm{ind}} (\forall X).e \qquad M \cup \{\texttt{eq } e.\} \vdash_{\mathrm{ind}} p}$$

$$\frac{M \vdash_{\mathrm{ind}} p}{M \vdash_{\mathrm{ind}} (\forall X).m \qquad M \cup \{\texttt{mb } m.\} \vdash_{\mathrm{ind}} p}$$

The explicit induction style of the above rules of inference is similar to that used in other explicit induction approaches in equational theorem proving such as those of OBJ [7, 8] and LARCH [11]. The module `ITP` below is the specification in Maude of the inference system just introduced. Formulas are represented with the operations `trueFormula`, `equality`, `sortP`, `implication`, `conjunction`, and `VQuantification`.

```
mod ITP is
protecting META-LEVEL .
sort Formula .

op trueFormula : -> Formula .
```

```
op equality : Term Term -> Formula .
op sortP : Term Qid -> Formula .
op implication : Formula Formula -> Formula .
op conjunction : Formula Formula -> Formula [assoc comm] .
op VQuantification : QidSet Formula -> Formula .

var Xs Ys : QidSet .
vars Alpha Beta : Formula .

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) .
```

As an example, consider the following module `NAT` :

```
fmod NAT is
sorts Nat NzNat .
subsort NzNat < Nat .
op 0 : -> Nat .
op s : Nat -> NzNat .
op + : Nat Nat -> Nat  .
vars N M X Y Z : Nat .
eq +(N, 0) = N .
eq +(N, s(M)) = s(+(N, M)) .
```

The universally quantified atomic formula $(\forall\{\mathtt{N},\mathtt{M}\})$.`+(N, M)` $=$ `+(M, N)` is represented in `ITP` by the term

`VQuantification(qidSet('N, 'M), equality('+['N, 'M], '+['M, 'N]))`.

Similarly, $(\forall\{\mathtt{N},\mathtt{M}\})$.`+(N, M)` : `NzNat` is represented by the term

`VQuantification(qidSet('N, 'M), sortP('+['N, 'M], 'NzNat))`.

Sets of terms are built with the constructor `termSet`, with `emptyTermSet` the empty set of terms. The constructor `test` is used to represent the test set for a given sort. A set of test sets is built with the constructor `testSet`, with `emptyTestSet` the empty set of test sets.

```
sort TermSet Test TestSet .

subsort Term < TermSet .
subsort Test < TestSet .

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] .
```

For example, the term `test('Nat, termSet('0, s['N]))` is the representation of the test set [4] for the sort `Nat` in the module `NAT`.

The (sub)goals for our inductive theorem prover are represented with the constructors `proveinInitial` and `proveinVariety`, for proofs in the initial model and proofs in the variety, respectively. For example, the goal

$$\texttt{NAT} \vdash_{\mathrm{ind}} (\forall\{\texttt{N, M}\}).\texttt{+(N, M)} = \texttt{+(M, N)}$$

is represented in `ITP` by the term

```
proveinInitial($I$, $\overline{\texttt{NAT}}^m$,
     VQuantification(qidSet('N, 'M),
                     equality('+['N, 'M], '+['M, 'N])), $\overline{TS}$),
```

where I should be a string of positive numbers, and $\overline{TS}$ is the representation of the set of test sets for the sorts of `NAT`, that is, the term

```
testSet(test('Nat, termSet('Nat, termSet('0, s['N]))
        test('NzNat, 's['N])).
```

The strings of positive numbers are used to number the (sub)goals in a proof. Sets of (sub)goals are built with the constructor `goalSet`, with `emptyGoalSet` the empty set of goals. In what follows, we assume that any functional module T about which we want to proof inductive theorems satisfies the following requirements:

1. for any sorts s, s' in T, if $s < s'$, then `subsort` s `<` s' is part of the subsort declaration in T; and
2. for any variable x of sort s declared in T, x`*` is not a constant of sort s declared in T.

```
sorts IntString Goal GoalSet .

subsort MachineInt < IntString .
subsort Goal < GoalSet .

op intString : IntString IntString -> IntString [assoc] .

op proveinInitial : IntString Module Formula TestSet -> Goal .
op proveinVariety : IntString Module Formula -> Goal .

op emptyGoalSet : -> GoalSet .
op goalSet : GoalSet GoalSet -> GoalSet
            [assoc comm id: emptyGoalSet] .
```

[4]The correctness of a test set for a given sort depends on having shown that the subsignature of constructors on which the test set is based does indeed generate all terms of that sort, in the sense that any term of that sort is provably equal to a constructor term. As explained in Sections 4.4 and 4.5, given a subsignature of constructors for our specification we can use the inductive theorem prover to show that indeed the constructors generate all terms of all sorts, and then we can mechanically check or generate test sets for each sort. In what follows we shall assume that this has already been done for the specification in question.

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

The inference rule **Test Set Induction** is specified by the rules testSetInduction.

```
op findSort : Qid VarDeclSet -> Qid .
op getVars : Module -> VarDeclSet .

op makeNewGoals : IntString Module QidSet Qid Term
                  Term TermSet TestSet -> GoalSet .
op makeNewGoalsMb : IntString Module QidSet Qid Term Term
                    TermSet TestSet -> GoalSet .

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 .
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 .
```

For T a functional module in Maude with VD its variable declaration, the function `getVars(`$\overline{T}^m$`)` returns $\overline{VD}^m$.

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

For x a variable of sort s declared in a variable declaration VD, the function `findSort(`$\overline{x}^m$`, `$\overline{VD}^m$`)` returns $\overline{s}^m$.

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

Let T be a functional module in Maude, X a set of variables declared in T, $x \notin X$ a variable of sort s declared in T, t and t' terms in T, $\{u_1, \ldots, u_n\}$ the test set for sort s, and TS the set of test sets for sorts in T. Then, the function `makeNewGoals(`I`, `$\overline{T}^m$`, `$\overline{X}^m$`, `$\overline{x}^m$`, `$\overline{t}^m$`, `$\overline{t'}$`, `$\overline{\{u_1, \ldots, u_n\}}^m$`, `$\overline{TS}^m$`)` returns

$$\overline{M \vdash_{\mathrm{ind}} step(t = t', X, x, s, u_1) \cdots M \vdash_{\mathrm{ind}} step(t = t', X, x, s, u_n)}^m,$$

with each new subgoal conveniently numbered after I.

```
op substitute : TermList Substitution -> TermList .
op getSubsorts : Module -> SubsortDeclSet .
op vars : Term VarDeclSet -> QidSet .
op varsSubsort : QidSet VarDeclSet Qid SubsortDeclSet -> QidSet .
op makeConjunctionHypoth : QidSet Term Term Qid QidSet
                           -> Formula .
op makeConjunctionHypothMb : QidSet Term Term Qid QidSet
                             -> Formula .

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 .
```

The function makeNewGoalsMb(I, $\overline{T}^m$, $\overline{X}^m$, $\overline{x}^m$, $\overline{t}^m$, s', $\overline{\{u_1,\ldots,u_n\}}^m$, $\overline{TS}^m$) is similar to makeNewGoals but it returns instead

$$\overline{M \vdash_{\mathrm{ind}} \mathit{step}(t : s', X, x, s, u_1) \cdots M \vdash_{\mathrm{ind}} \mathit{step}(t : s', X, x, s, u_n)}^m,$$

with each new subgoal conveniently numbered after I.

```
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 .
```

For t a term and σ a substitution, the function substitute($\overline{t}^m$, $\overline{\sigma}^m$) returns $\overline{t\sigma}^m$.

```
var SB : Substitution .

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)) .
```

For T a functional module in Maude with SBD its subsort declaration, the function getSubsorts($\overline{T}^m$) returns $\overline{SBD}^m$.

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

For T a functional module, t a term in T, and VD the variable declaration of T, the function vars($\overline{t}^m$, $\overline{VD}^m$) returns $\overline{\mathcal{V}ar(t)}^m$.

```
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)) .
```

Let T be a functional module, VD and SBD its variable declaration and subsort declaration respectively, X a set of variables declared in T, and s a sort declared in T. Then, the function `varsSubsort(`$\overline{X}^m$`, `$\overline{VD}^m$`, `$\overline{s}^m$`, `$\overline{SBD}^m$`)` returns the subset $Y \subseteq X = \{y : s' \mid s' \leq s\}$. Notice that we assumed that, if $s' < s$ in T, then `subsort` s' `<` s is part of the subsort declaration of T.

```
op subsort? : Qid Qid SubsortDeclSet -> Bool .
op and : Bool Bool -> Bool [assoc comm] .

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

eq varsSubsort(emptyQidSet, VarDs, S, SubsortDs) = emptyQidSet .

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

For T a functional module, X and $\{y_1, \ldots, y_n\}$ sets of variables declared in T, t and t' terms in T, and x a variable declared in T, the function `makeConjunction-Hypoth(`$\overline{X}^m$`, `$\overline{t}^m$`, `$\overline{t'}^m$`, `$\overline{x}^m$`, `$\overline{\{y_1, \ldots, y_n\}}^m$`)` returns

$$\overline{(\forall X).(t = t')[y_1/x] \wedge \cdots \wedge (\forall X).(t = t')[y_n/x]}^m$$

```
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, emptyQidSet) =
   trueFormula .
```

The function `makeConjunctionHypothMb(`$\overline{X}^m$`, `$\overline{t}^m$`, `$\overline{s}^m$`, `$\overline{x}^m$`, `$\overline{\{y_1, \ldots, y_n\}}^m$`)` is similar to `makeConjunctionHypoth` but it returns instead

$$\overline{(\forall X).(t : s)[y_1/x] \wedge \cdots \wedge (\forall X).(t : s)[y_n/x]}^m$$

```
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, emptyQidSet) =
   trueFormula .
```

The inference rule **Proof in Variety** is specified by the rule `inVariety`.

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

The inference rule **Constants Lemma** is specified by the rule `constantsLemma`.

```
op addNewConstants : QidSet Module -> Module .
op varsToNewConstants : QidSet Formula -> Formula .

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

For T a functional module, and X a set of variables declared in T, the function `addNewConstants(`$\overline{X}^m$`, `$\overline{T}^m$`)` returns $\overline{T'}^m$, where T' is the module T with its signature extended in the following way: for each variable x of sort s in X, a new constant $x*$ of sort s is added to the signature of T. Notice that we assumed that for any functional module T, if x is a variable declared in T, then $x*$ is not a constant declared in T.

```
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) .
```

For T a functional module, $\{x_1, \ldots, x_n\}$ a set of variables declared in T, and α a formula about T, the function `varsToNewConstants(`$\overline{\{x_1, \ldots, x_n\}}^m$`, `$\overline{\alpha}^m$`)` returns $\overline{\alpha[x_1*/x_1, \ldots, x_n*/x_n]}^m$.

```
op varsToNewConstantsAux : QidSet TermList -> TermList .
op inSet : Qid QidSet -> Bool .

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

The inference rules for **Implication Elimination** are jointly specified by the rule `implicationElimination`.

```
op addToModule : Formula Module -> Module .

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

Let T be a functional module, and $\alpha = (\forall X_1).p_1 \wedge \ldots \wedge (\forall X_n).p_n$ a formula about T such that, for $1 \leq i \leq n$, p_i is either $t_i = t'_i$ or $t_i : s_i$. Then, the function `addToModule(`$\overline{\alpha}^m$`,` $\overline{T}^m$`)` returns $\overline{T'}$, where T' is the module T extended in the following way: for each component $(\forall X_i).t_i = t'_i$ in the conjunction α the equation `eq` $t_i = t'_i$ is added to the set of equations of T; and for each component $(\forall X_i).t_i : s_i$ in the conjunction α the membership axiom `mb` $t_i = t'_i$ is added to the set of membership axioms of T.

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

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) .
```

The inference rules for **Proof by Rewriting for Ground Atoms** are specified by the rules `byRewriting`. Notice the use of the built-in function `meta-reduce` in our specification of this inference rules.

```
op equal : Term Term -> Formula .

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

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

eq proveinVariety(Idx, FunctTh, equal(T, T)) = emptyGoalSet .
```

Finally, the inference rules **Equational and Membership Lemmas** are specified by the rule `lemmaIntroduction`.

```
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) .

endm
```

4.2 Proof strategies

In this section, we introduce a module PROOF-STRATEGY as a strategy language in which we can represent proof strategies for our inductive theorem prover. The module PROOF-STRATEGY imports the module STRATEGY introduced in Section 3.

```
fmod PROOF-STRATEGY is
*** protecting STRATEGY .
```

As an example, we can represent a proof strategy that tries to simplify a particular (sub)goal $T \vdash_{\mathrm{ind}} (\forall X).p$ as much as possible. In order to do that, we introduce the operation simplify whose only argument should be the metarepresentation of the string of positive numbers corresponding to the (sub)goal we want to simplify. Notice how we use the operation applyWithSubst to apply the rewrite rules in the module ITP only to the (sub)goal we are interested in simplifying.

```
op simplify : Term -> Strategy .

var I : Term .

eq simplify(I) =
 and(applyWithSubst('inVariety, ('Idx <- I)),
 orelse(applyWithSubst('byRewriting, ('Idx <- I)),
    and(applyWithSubst('constantsLemma, ('Idx <- I)),
    orelse(applyWithSubst('byRewriting, ('Idx <- I)),
       and(applyWithSubst('implicationElimination, ('Idx <- I)),
       and(orelse(applyWithSubst('constantsLemma, ('Idx <- I)),
              idle),
       applyWithSubst('byRewriting, ('Idx <- I)))))))) .
```

We can also represent a proof strategy that, for a given goal, applies the induction rule on a selected variable and then it simplifies the resulting subgoals as much as possible. In order to represent this proof strategy we add to PROOF-STRATEGY the operation induction whose only argument should be the meta-metarepresentation of the induction variable. Again, we use the operation applyWithSubst to apply the rule testSetInduction so that the actual induction variable is the desired one.

```
op induction : Term -> Strategy .

var IV : Term .

eq induction(IV) =
   and(applyWithSubst('testSetInduction, ('X <- IV)),
   iterate(and(apply('inVariety),
           orelse(apply('byRewriting),
                  and(apply('constantsLemma),
                  orelse(apply('byRewriting),
                         and(apply('implicationElimination),
                         and(orelse(apply('constantsLemma),
                                    idle),
                         apply('byRewriting))))))))) .
```

As an example, consider the module NAT introduced at the beginning of this section.

```
fmod NAT is
sorts Nat NzNat .
subsort NzNat < Nat .
op 0 : -> Nat .
op s : Nat -> NzNat .
op + : Nat Nat -> Nat  .
vars N M X Y Z : Nat .
eq +(N, 0) = N .
eq +(N, s(M)) = s(+(N, M)) .
endfm
```

Suppose that we want to prove the associativity of the operation `+` (addition) as defined in `NAT`, that is, $(\forall\{\texttt{X},\texttt{Y},\texttt{Z}\}).\texttt{+(X, +(Y, Z))} = \texttt{+(+(X, Y), Z)}$. We can start our proof applying the rule `testSetInduction` on the variable `Z`. In what follows, given a formula α, $\overline{\alpha}^f$ denotes the representation of the formula α in the module `ITP` as described in page 18. Also, we denote by `TSN` the representation of the set of test sets for the sorts of `NAT`, that is, the term

```
testSet(test('Nat, termSet('Nat, termSet('0, s['N])))
        test('NzNat, 's['N])).
```

$$
\begin{array}{l}
\texttt{Maude> rew rewInWith(}\overline{\texttt{ITP}}^m\texttt{,}\\
\qquad\qquad \texttt{'proveinInitial[}\\
\qquad\qquad\quad \texttt{'1, }\overline{\texttt{NAT}}^m\texttt{,}\\
\qquad\qquad\quad \overline{\overline{(\forall\{\texttt{X, Y, Z}\}).\texttt{+(X, +(Y, Z))} = \texttt{+(+(X, Y), Z)}}^f}^m\texttt{,}\\
\qquad\qquad\quad \overline{\texttt{TSN}}^m\texttt{],}\\
\qquad\qquad \texttt{nilBindingList,}\\
\qquad\qquad \texttt{applyWithSubst('testSetInduction, ('X <- }\overline{\overline{\texttt{Z}}^m}^m\texttt{))) .}
\end{array}
$$

$$
\begin{array}{l}
\texttt{rewrites: 17 in 9ms cpu (19ms real) (1700 rewrites/second)}\\
\texttt{result StrategyExpression:}\\
\quad \texttt{rewInWith(}\overline{\texttt{ITP}}^m\texttt{,}\\
\qquad \texttt{'goalSet[}\\
\qquad\quad \texttt{('proveinInitial[}\\
\qquad\qquad\quad \texttt{('intString['1, '1]), }\overline{\overline{\texttt{NAT}}^m}^m\texttt{,}\\
\qquad\qquad\quad \overline{\overline{(\forall\{\texttt{X,Y}\}).\texttt{+(X, +(Y, 0))} = \texttt{+(+(X, Y), 0)}}^f}^m\texttt{,}\\
\qquad\qquad\quad \overline{\texttt{TSN}}^m\texttt{]),}\\
\qquad\quad \texttt{('proveinInitial[}\\
\qquad\qquad\quad \texttt{('intString['1, '2]), }\overline{\overline{\texttt{NAT}}^m}^m\texttt{,}\\
\qquad\qquad\quad \overline{\overline{(\forall\{\texttt{N}\}).((\forall\{\texttt{X,Y}\}).\texttt{+(X, +(Y, N))} = \texttt{+(+(X, Y), N)}) \Rightarrow}^f}^m\\
\qquad\qquad\qquad\quad \overline{\overline{((\forall\{\texttt{X,Y}\}).\texttt{+(X, +(Y, s(N)))} = \texttt{+(+(X, Y), s(N))})}^f}^m\texttt{,}\\
\qquad\qquad\quad \overline{\texttt{TSN}}^m\texttt{])],}\\
\quad \texttt{nilBindingList, idle)}
\end{array}
$$

Next we can try to simplify the subgoal `'intString['1, '1]`, using the operation `simplify`.

```
Maude> rew rewInWith(
           ITP^m,
           'goalSet[
              ('proveinInitial[
                 ('intString['1, '1]),
                 NAT^m^m,
                 (∀{X,Y}).+(X, +(Y, 0)) = +(+(X, Y), 0)^f^m,
                 TSN^m]),
              ('proveinInitial[
                 ('intString['1, '2]),
                 NAT^m^m,
                 (∀{N}).((∀{X,Y}).+(X, +(Y, N)) = +(+(X, Y), N)) ⇒
                       ((∀{X,Y}).+(X, +(Y, s(N))) = +(+(X, Y), s(N)))^f^m)^f^m,
                 TSN^m])]),
           nilBindingList,
           simplify('intString['1, '1])) .

rewrites:  53 in 29ms cpu (49ms real) (1766 rewrites/second)
result StrategyExpression:
   rewInWith(
      ITP^m,
      ('proveinInitial[
         ('intString['1, '2]),
         NAT^m^m,
         (∀{N}).((∀{X,Y}).+(X, +(Y, N)) = +(+(X, Y), N)) ⇒
               ((∀{X,Y}).+(X, +(Y, s(N))) = +(+(X, Y), s(N)))^f^m)^f^m,
         TSN^m]),
      nilBindingList,
      idle)
```

To finish the proof we can use again the operation `simplify` to simplify the only remaining subgoal.

```
Maude> rew rewInWith(ITP^m,
           ('proveinInitial[
              ('intString['1, '2]),
              NAT^m^m,
              (∀{N}).((∀{X,Y}).+(X, +(Y, N)) = +(+(X, Y), N)) ⇒
                    ((∀{X,Y}).+(X, +(Y, s(N))) = +(+(X, Y), s(N)))^f^m)^f^m,
              TSN^m]),
           nilBindingList,
           simplify('intString['1, '2])) .

rewrites:  92 in 39ms cpu (109ms real) (2300 rewrites/second)
result StrategyExpression:
      rewInWith(ITP^m, 'emptyGoalSet, nilBindingList, idle)
```

Of course, in this case we could alternatively have used the `induction` strategy to prove "in one shot" the associativity of the operation `+` as defined in the module `NAT`.

$$
\begin{array}{l}
\texttt{Maude> rew rewInWith(} \\
\qquad\qquad \overline{\texttt{ITP}}^{m}\texttt{,} \\
\qquad\qquad \texttt{'proveinInitial[} \\
\qquad\qquad\quad \texttt{'1, } \overline{\texttt{NAT}}^{m}\texttt{,} \\
\qquad\qquad\quad \overline{\overline{(\forall\{\texttt{X, Y, Z}\}).\texttt{+(X, +(Y, Z))} = \texttt{+(+(X, Y), Z)}}^{f}}^{m}\texttt{ ,} \\
\qquad\qquad\quad \overline{\texttt{TSN}}^{m}\texttt{],} \\
\qquad\qquad \texttt{nilBindingList,} \\
\qquad\qquad \texttt{induction(}\overline{\overline{\texttt{Z}}^{m}}^{m}\texttt{)) .}
\end{array}
$$

$$
\begin{array}{l}
\texttt{rewrites: 139 in 79ms cpu (99ms real) (1737 rewrites/second)} \\
\texttt{result StrategyExpression:} \\
\qquad \texttt{rewInWith(}\overline{\texttt{ITP}}^{m}\texttt{, 'emptyGoalSet, nilBindingList, idle)}
\end{array}
$$

4.3 Allowing General Conditional Equations

To allow extra generality in the specifications accepted by the inductive theorem prover, conditional equations are not restricted to having a single Boolean condition. They can instead have a conjunction of equations. That is, the conditional equations allowed are of the form

$$t = t' \text{ if } t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n.$$

where now the t_i, t'_i can have arbitrary sorts, but each pair t_i, t'_i must of course be in the same connected component.

Since we assume that the original specification is ground Church-Rosser and terminating, operationally the satisfaction of each of the equations in the condition is interpreted as confluence, that is, as

$$t = t' \text{ if } t_1 \downarrow t'_1 \wedge \ldots \wedge t_n \downarrow t'_n. \qquad (\dagger)$$

To achieve this extra generality in conditional equations we have not modified the previous code. Instead, we have added a new function which, after making some changes in the specification given by the user to allow conjunctions in conditions, passes it to the prover itself.

The constructor for the conditional equations at the meta-level has the declaration

```
op ceq : Term Term Term Term -> Equation .
```

where the second and third arguments represent the condition. We then allow the extra generality of conditional equations of the form (†) by adopting the following meta-representation of such an equation, assuming that a term t is meta-represented as $\overline{t}^m$,

$$\texttt{eq(}\overline{t}^m\texttt{, }\overline{t'}^m\texttt{, 'and['\_==\_[}\overline{t_1}^m\texttt{ , }\overline{t'_1}^m\texttt{], ... , '\_==\_[}\overline{t_n}^m\texttt{ , }\overline{t'_n}^m\texttt{]], 'true)}$$

where `and` is a predefined operation. As with built-in operations the operator `and` cannot be redeclared nor redefined. The declaration of this operator is

```
op and : Bool Bool -> Bool [assoc comm] .
```

Its semantics is the usual one. As in general for all built-ins, `and` cannot be used in any other place in the input. Only in the conditions of the equations, and only in the way pointed out. We could consider it just as a piece of syntax we have added to facilitate use. Note also that the fourth component has to be `true` and that this is the only place where it can appear.

Recall that, given a user module M, a set of goals GS (represented as indicated in Section 4.1), a binding list BL, and a strategy ST, the input to the theorem prover is of the following form:

$$\texttt{rew rewInWith(}\overline{\texttt{ITP}}^{m}\texttt{, } GS\texttt{, } BL\texttt{, } ST\texttt{) .}$$

To work with specifications using the more general conditional equations as described above, the input must be given using a different function `rewInWith2`. That is, in the case of conditional specifications of this, more general, form the input has to be given as:

$$\texttt{rew rewInWith2(}\overline{\texttt{ITP}}^{m}\texttt{, } GS\texttt{, } BL\texttt{, } ST\texttt{) .}$$

The way in which these equations are treated here is quite similar to the way in which they are treated in the Church-Rosser Checker presented in Section 5. In order to avoid modifying the actual input to the prover, we add the definition of the operator `and`, as indicated above, to the input module and then call the function `rewInWith`. The introduction of this operator is accomplished by the function `addAnd`, which is located in the module `ITP`. This function takes a module of sort `Module` and adds to it the definition for the operator `and`, a variable `B@'Bool`, declared to be of the built-in sort `Bool`, and the corresponding equations for `and`.

```
var QI : Qid .
var IL : ImportList .
var SD : SortDecl .
var SSDS : SubsortDeclSet .
var ODL : OpDeclList .
var VDS : VarDeclSet .
var MAS : MembAxSet .
var ES : EquationSet .

op addAnd : Module -> Module .

eq addAnd(fmod(QI, IL, SD, SSDS, ODL, VDS, MAS, ES))
  = fmod(QI, IL, SD, SSDS,
         opDeclList(opDecl('and, qidList('Bool, 'Bool), 'Bool,
                           attrSet(comm, assoc)), ODL),
         varDeclSet(varDecl('B@'Bool, 'Bool), VDS), MAS,
         equationSet(eq('and['false, 'B@'Bool], 'false),
                     eq('and['true, 'B@'Bool], 'B@'Bool), ES)) .
```

The declaration of the function `rewInWith2` is included in the module `PROOF-STRATEGY`, and its definition is as follows:

```
var M : Module .
vars T T' T'' T''' : Term .
var TL : TermList .
```

```
var BL : BindingList .
var ST : Strategy .

op rewInWith2 : Module Term BindingList Strategy -> StrategyExpression .
op prepareGoalSet : TermList -> TermList .

eq rewInWith2(M, T, BL, ST)
  = rewInWith(M, prepareGoalSet(T), BL, ST) .

eq prepareGoalSet('goalSet[TL])
  = 'goalSet[prepareGoalSet(TL)] .

eq prepareGoalSet('proveinInitial[T, T', T'', T'''])
  = 'proveinInitial[T, meta-reduce(ITP, 'addAnd[T']), T'', T'''] .
eq prepareGoalSet(('proveinInitial[T, T', T'', T'''], TL))
  = ('proveinInitial[T, meta-reduce(ITP, 'addAnd[T']), T'', T'''],
     prepareGoalSet(TL)) .
eq prepareGoalSet('proveinVariety[T, T', T'', T'''])
  = 'proveinVariety[T, meta-reduce(ITP, 'addAnd[T']), T'', T'''] .
eq prepareGoalSet(('proveinVariety[T, T', T'', T'''], TL))
  = ('proveinVariety[T, meta-reduce(ITP, 'addAnd[T']), T'', T'''],
     prepareGoalSet(TL)) .
```

4.4 Sufficient Generation

Sufficient generation is a property somewhat weaker than sufficient completeness. Given an order-sorted specification $T = (S, \leq, \Sigma, \Gamma)$ and a subspecification of constructors $T_0 = (S, \leq, \Omega, \Gamma_0)$, with $T_0 \subseteq T$, we say that the initial algebra of T is *sufficiently complete* relative to T_0 if the unique Ω-homomorphism

$$\mathcal{T}_{T_0} \longrightarrow \mathcal{T}_T|_\Omega$$

from the initial algebra of T_0 to the Ω-algebra obtained from the initial algebra of $\mathcal{T}_T$ by forgetting the operations in $\Sigma - \Omega$ is an *isomorphism*. Instead, we say that the initial algebra of T is *sufficiently generated* by the subsignature of constructors Ω, if the unique Ω-homomorphism

$$\mathcal{T}_\Omega \longrightarrow \mathcal{T}_T|_\Omega$$

is surjective for each sort $s \in S$. This is equivalent to requiring that for each $s \in S$ and each $t \in \mathcal{T}_{\Sigma,s}$ there is a $t_0 \in \mathcal{T}_{\Omega,s}$ with

$$T \vdash t = t_0$$

Sufficient completeness is of course a stronger property than sufficient generation. But for being justified in using only constructor terms in the test sets of an inductive proof, only sufficient generation must be checked.

We give below a general method for checking that the initial algebra of T is sufficiently generated by a subsignature of constructors Ω. The advantage of this method is that we can use the inductive theorem prover to check the proof obligations that it generates. The correctness of the method can be regarded as a special case of the proof of a more general theorem by Bouhoula, Jouannaud and Meseguer on protecting extensions of specifications in membership equational logic [1, Theorem 57].

The method is as follows. Given T and a subsignature of constructors Ω, we define a new specification $E_\Omega T = (ES, \leq_E, E\Sigma, \Gamma)$ with:

- sorts those of S plus, for each connected component C of $(S, \leq)$ a new "error sort" E_C greater than all the sorts in C,
- order $\leq_E$ the obvious extension of $\leq$ to make each E_C the top of each component,
- operators $E\Sigma$ those of Ω (with same arity and coarity), plus for each $f : s_1 \ldots s_n \to s$ in $\Sigma - \Omega$ with $c(s_1), \ldots, c(s_n), c(s)$ the corresponding connected components, an operator $f : E_{c(s_1)} \ldots E_{c(s_n)} \to E_{c(s)}$, and
- axioms Γ exactly the same as those of T.

Then, to prove that the initial algebra of T is sufficiently generated by Ω, it is enough to show that for each[5] $f : s_1 \ldots s_n \to s$ in $\Sigma - \Omega$,

$$E_\Omega T \vdash_{\text{ind}} f(x_1, \ldots, x_n) : s \qquad (\ddagger)$$

where the variables $x_1, \ldots, x_n$ have respective sorts $s_1, \ldots, s_n$. Note that this inductive requirement takes place in membership equational logic. Therefore, we implicitly use the embedding of order-sorted equational logic into membership equational logic [17], and the fact that Maude's functional modules are theories in membership equational logic.

We can then use the inductive theorem prover to prove each of the proof obligations ($\ddagger$). Note that, since the specification $E_\Omega T$ is of course sufficiently generated by its own signature $E\Sigma$, and $E\Sigma$ has the property that for each $s \in S$ all operators of coarity less or equal to s are in Ω, when proving the proof obligations ($\ddagger$), we are justified in using only Ω-*terms* in the test sets of sorts $s \in S$ in $E_\Omega T$.

4.4.1 Proving Sufficient Generation of Natural Numbers

Let us consider the following specification for the natural numbers.

```
mod NAT is
  sorts Zero Nat .
  subsort Zero < Nat .
  op 0 : -> Zero .
  op s : Nat -> Nat .
  op (_+_) (_*_) : Nat Nat -> Nat [comm] .
  vars N M : Nat .
  eq 0 + N = N .
  eq s(N) + M = s(N + M) .
  eq 0 * N = 0 .
  eq s(N) * M = M + (N * M) .
endm
```

We will prove that the constructors for `NAT` are `0` and `s`. To do this we will introduce a new sort `ErrorSort` as supersort of `Nat` and will move up all the operations except those considered as candidates for constructors.

[5]Note that an operator declaration $f : s_1 \ldots s_n \to s$ can be in $\Sigma - \Omega$ while at the same time another declaration $f : s'_1 \ldots s'_n \to s'$ can be in Ω. For example, `s : Int -> Int` is not a constructor, but `s : Nat -> Nat` is a constructor.

```
mod ENAT is
  sorts Zero Nat ErrorNat .
  subsort Zero < Nat < ErrorNat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op (_+_) (_*_) : ErrorNat ErrorNat -> ErrorNat [comm] .
  vars N M : Nat .
  eq 0 + N = N .
  eq s(N) + M = s(N + M) .
  eq 0 * N = 0 .
  eq s(N) * M = M + (N * M) .
endm
```

The first thing we need in order to be able to use the theorem prover is to obtain a test set for ENAT. What we can do is to consider all the operators of ENAT as constructors, something we are always justified in doing, since the entire operator set obviously generates the initial algebra. For this purpose we call the function generateTestSets with the ENAT module previously prepared as explained in Section 4.5. The preparation in question consists in a metalevel syntactic variant CNAT of the module ENAT in which all the operators are syntactically marked as constructors in copDecl declarations. Therefore, CNAT is a module in a supersort EModule of Module corresponding to an extended syntax that allows us to distinguish constructors and non-constructors using the copDecl operator for the constructors. The transformed module CNAT is the following module of sort EModule.

```
fmod(
   'CNAT,
   nilImportList,
   sort(qidSet('Zero, 'Nat, 'ErrorNat)),
   subsortDeclSet(subsort('Zero, 'Nat), subsort('Nat, 'ErrorNat)),
   eOpDeclList(
      copDecl('0, nilQidList, 'Zero, emptyAttrSet),
      copDecl('s, 'Nat, 'Nat, emptyAttrSet),
      copDecl('+, qidList('ErrorNat, 'ErrorNat), 'ErrorNat, comm),
      copDecl('*, qidList('ErrorNat, 'ErrorNat), 'ErrorNat, comm)),
   varDeclSet(
      varDecl('N, 'Nat), varDecl('M, 'Nat)),
   emptyMembAxSet,
   equationSet(
      eq('+['0, 'N], 'N ),
      eq('+['s['N], 'M], 's['+['N, 'M]]),
      eq('*['0, 'N], '0),
      eq('*['s['N], 'M], '+['M, '*['N, 'M]]))) .
```

Then, calling the function generateTestSets with CNAT as argument gives us the following result[6]. The function returns a pair consisting of a modified module, in

[6]Note that since the interaction is done at the meta-metalevel, as detailed in Section 4.5, the way to call the function would be

red meta-reduce(ITP, $\overline{\overline{\texttt{CNAT}}}^{m}$) .

And the result would be at the meta-metalevel also.

which some variables may have been introduced, and the corresponding family of test sets.

```
Maude> red generateTestSets(CNAT) .
rewrites: 292 in 0ms cpu (9ms real) (~ rewrites/second)
result TestSetSolution:
testSetSolution(
    fmod(
       'CNAT,
       nilImportList,
       sort(qidSet('Zero, 'Nat, 'ErrorNat)),
       subsortDeclSet(subsort('Zero, 'Nat), subsort('Nat, 'ErrorNat)),
       eOpDeclList(
          copDecl('0, nilQidList, 'Zero, emptyAttrSet),
          copDecl('s, 'Nat, 'Nat, emptyAttrSet),
          copDecl('+, qidList('ErrorNat, 'ErrorNat), 'ErrorNat, comm),
          copDecl('*, qidList('ErrorNat, 'ErrorNat), 'ErrorNat, comm)),
       varDeclSet(
          varDecl('N, 'Nat), varDecl('M, 'Nat),
          varDecl('ErrorNat&'0, 'ErrorNat),
          varDecl('ErrorNat&'1, 'ErrorNat)),
       emptyMembAxSet,
       equationSet(
          eq('+['0, 'N], 'N ),
          eq('+['s['N], 'M], 's['+['N, 'M]]),
          eq('*['0, 'N], '0),
          eq('*['s['N], 'M], '+['M, '*['N, 'M]]))),
    testSet(
       test('Zero, '0),
       test('Nat, termSet('0, 's['N])),
       test('ErrorNat,
            termSet('0, '*['ErrorNat&'0, 'ErrorNat&'1],
                    's['N], '+['ErrorNat&'0, 'ErrorNat&'1]))))
```

Note that new variables have been introduced, and that it is important to use this module in order to be able to use the theorem prover with the test sets returned. Transforming back the module returned with the test sets to the original module syntax without constructor declarations we obtain the following module ENAT at the metalevel.

```
    fmod(
       'ENAT,
       nilImportList,
       sort(qidSet('Zero, 'Nat, 'ErrorNat)),
       subsortDeclSet(subsort('Zero, 'Nat), subsort('Nat, 'ErrorNat)),
       opDeclList(
          opDecl('0, nilQidList, 'Zero, emptyAttrSet),
          opDecl('s, 'Nat, 'Nat, emptyAttrSet),
          opDecl('+, qidList('ErrorNat, 'ErrorNat), 'ErrorNat, comm),
          opDecl('*, qidList('ErrorNat, 'ErrorNat), 'ErrorNat, comm)),
```

```
varDeclSet(
   varDecl('N, 'Nat), varDecl('M, 'Nat),
   varDecl('ErrorNat&'0, 'ErrorNat),
   varDecl('ErrorNat&'1, 'ErrorNat)),
emptyMembAxSet,
equationSet(
   eq('+['0, 'N], 'N ),
   eq('+['s['N], 'M], 's['+['N, 'M]]),
   eq('*['0, 'N], '0),
   eq('*['s['N], 'M], '+['M, '*['N, 'M]])))
```

In order to continue using the same notation introduced above, we will call this module $\overline{\texttt{ENAT}}^m$. And let us call `ENATTS` to the test sets returned:

```
testSet(
   test('Zero, '0),
   test('Nat, termSet('0, 's['N])),
   test('ErrorNat,
        termSet('0, '*['ErrorNat&'0, 'ErrorNat&'1],
                's['N], '+['ErrorNat&'0, 'ErrorNat&'1])))
```

Now we can prove the proof obligations for sufficient generation by the method already explained. To prove that the operators `0 : -> Nat` and `s : Nat -> Nat` are constructors for `NAT`, we need to prove inductively that in the module `ENAT`, given variables `N` and `M` of sort `Nat`, `N + M` and `N * M` are of sort `Nat`. That is, we need to prove inductively the following goals:

$$(\forall\{\texttt{N, M}\}).\texttt{N + M: Nat} \text{ and } (\forall\{\texttt{N,M}\}).\texttt{N * M: Nat}.$$

The first goal can be proved using the strategy `induction` on the variable `N`.

```
Maude> rew rewInWith(
            $\overline{ITP}^m$,
            ('proveinInitial[
                '1,
                $\overline{\overline{ENAT}^m}^m$,
                $\overline{\overline{(\forall\{N, M\}).N + M: Nat}^f}^m$,
                $\overline{ENATTS}^m$],
            nilBindingList,
            induction($\overline{\overline{N}^m}^m$)) .

rewrites: 142 in 29ms cpu (39ms real) (4733 rewrites/second)
result StrategyExpression:
     rewInWith($\overline{ITP}^m$, 'emptyGoalSet, nilBindingList, idle)
```

To prove the second goal we have to start introducing the previous result as a lemma. We can do it by applying the rule `lemmaIntroduction` with the variable `Beta` instantiated to the meta-representation of the formula as described in page 18.

```
red rewInWith(
```

```
        ITP^m,
        ('proveinInitial[
           '1,
           ENAT^m^m,
           (∀{N, M}).N * M: Nat^f^m,
           ENATTS^m]),
        nilBindingList,
        applyWithSubst(
           'lemmaIntroduction,
              (('Idx <- '1);
               ('Beta <- (∀{N, M}).N + M: Nat^f^m )))) .
```

The result obtained is the following:

```
result StrategyExpression:
rewInWith(
   ITP^m,
   'goalSet[
      ('proveinInitial[
         ('intString['1, '1]),
         ENAT ∪ {mb N + M: Nat .}^m^m,
         (∀{N, M}).N * M: Nat^f^m,
         ENATTS^m]),
      ('proveinInitial[
         ('intString['1, '2]),
         ENAT^m^m,
         (∀{N, M}).N + M: Nat^f^m ])],
         ENATTS^m])],
   nilBindingList,
   idle)
```

Both of these subgoals can then be proved by induction. Let us start with `1.1`; to prove it, we apply the rule `testSetInduction` to this subgoal on the variable `N` as follows:

```
red rewInWith(ITP^m,
        'goalSet[
           ('proveinInitial[
              ('intString['1, '1]),
              ENAT ∪ {mb N + M: Nat .}^m^m,
              (∀{N, M}).N * M: Nat^f^m,
              ENATTS^m]),
           ('proveinInitial[
              ('intString['1, '2]),
              ENAT^m^m,
              (∀{N, M}).N + M: Nat^f^m ])],
              ENATTS^m])],
```

```
        nilBindingList,
        applyWithSubst(
          'testSetInduction,
          (('X <- N) ; ('Idx <- ('intString['1, '1]))))))) .
```

The result:

```
result StrategyExpression:
rewInWith(
  ITP,
  'goalSet[
    ('proveinInitial[
      ('intString['1, '1, '1]),
      ENAT ∪ {mb N + M: Nat .},
      (∀{M}).0 * M: Nat,
      ENATTS]),
    ('proveinInitial[
      ('intString['1, '1, '2]),
      ENAT ∪ {mb N + M: Nat .},
      (∀{N}).((∀{M}).N + M: Nat) ⇒ ((∀{M}).s(N) + M: Nat),
      ENATTS]),
    ('proveinInitial[
      ('intString['1, '2]),
      ENAT,
      (∀{N, M}).N + M: Nat},
      ENATTS])],
  nilBindingList,
  idle)
```

We have obtained two new subgoals, `1.1.1` and `1.1.2`, one for each one of the terms in the test set of `Nat`. These two subgoals can be proved by rewriting applying the `simplify` strategy.

```
red rewInWith(
        ITP,
        'goalSet[
          ('proveinInitial[
            ('intString['1, '1, '1]),
            ENAT ∪ {mb N + M: Nat .},
            (∀{M}).0 * M: Nat,
            ENATTS]),
          ('proveinInitial[
            ('intString['1, '1, '2]),
            ENAT ∪ {mb N + M: Nat .},
            (∀{N}).((∀{M}).N + M: Nat) ⇒ ((∀{M}).s(N) + M: Nat),
            ENATTS]),
          ('proveinInitial[
```

$\texttt{('intString['1, '2]),}$
$\overline{\overline{\texttt{ENAT}}^{m}}^{m}\texttt{,}$
$\overline{\overline{\texttt{(}\forall\texttt{\{N, M\}).N + M: Nat}}^{f}\texttt{\}}}^{m}\texttt{,}$
$\overline{\texttt{ENATTS}}^{m}\texttt{])],}$
$\texttt{nilBindingList,}$
$\texttt{and(simplify('intString['1, '1, '1]),}$
$\texttt{simplify('intString['1, '1, '2]))))) .}$

The result:

$\texttt{result StrategyExpression:}$
$\texttt{rewInWith(}$
$\overline{\texttt{ITP}}^{m}\texttt{,}$
$\texttt{'proveinInitial[}$
$\texttt{('intString['1, '2]),}$
$\overline{\overline{\texttt{ENAT}}^{m}}^{m}\texttt{,}$
$\overline{\overline{\texttt{(}\forall\texttt{\{N, M\}).N + M: Nat}}^{f}\texttt{\}}}^{m}\texttt{,}$
$\overline{\texttt{ENATTS}}^{m}\texttt{])],}$
$\texttt{nilBindingList,}$
$\texttt{idle)}$

The only remaining subgoal is `1.2`, that was proved already. We can prove it again applying the `induction` strategy on the variable `N`.

$\texttt{red rewInWith(}$
$\overline{\texttt{ITP}}^{m}\texttt{,}$
$\texttt{'proveinInitial[}$
$\texttt{('intString['1, '2]),}$
$\overline{\overline{\texttt{ENAT}}^{m}}^{m}\texttt{,}$
$\overline{\overline{\texttt{(}\forall\texttt{\{N, M\}).N + M: Nat}}^{f}\texttt{\}}}^{m}\texttt{,}$
$\overline{\texttt{ENATTS}}^{m}\texttt{])],}$
$\texttt{nilBindingList,}$
$\texttt{induction(}\overline{\overline{\texttt{N}}^{m}}^{m}\texttt{)))))) .}$

With this we have completed the proof. The final result is:

$\texttt{result StrategyExpression:}$
$\texttt{rewInWith(}\overline{\texttt{ITP}}^{m}\texttt{, 'emptyGoalSet, nilBindingList, idle)}$

4.4.2 Proving Sufficient Generation of Integers

Let us consider now the following specification:

```
mod INT is
   sorts Int Neg .
   protecting NAT .
   subsorts Zero < Neg < Int .
   subsort Nat < Int .
   ops s p : Int -> Int .
```

```
   op p : Neg -> Neg .
   ops (_+_) (_*_) : Int Int -> Int [comm] .
   op -_ : Int -> Int .
   op square : Int -> Nat .
   vars N M : Nat .
   vars I J : Int .
   vars P Q : Neg .
   eq s(p(I)) = I .
   eq p(s(I)) = I .
   eq I + 0 = I .
   eq I + s(N) = s(I + N) .
   eq I + p(P) = p(I + P) .
   eq I * 0 = 0 .
   eq I * s(N) = (I * N) + I .
   eq I * p(P) = (I * P) + (- I) .
   eq - 0 = 0 .
   eq - s(N) = p(- N) .
   eq - p(P) = s(- P) .
   eq square(I) = I * I .
endm
```

Let us prove now that the operators

`0 :  -> Nat`, `s :  Nat -> Nat` and `p :  Neg -> Neg`

are constructors for `Int`. As we have done before, we will introduce a new sort `ErrorInt` as a supersort of `Int`, and we will move up to the sort `ErrorInt` the rest of the operations. Again, we get a module `EINT` to be used in proving the proof obligations of sufficient generation, plus its corresponding test sets, obtained by calling the function `generateTestSets`. The argument in the call has to be a module of sort `EModule` as explained in Section 4.5, that is, the meta-representation `CINT` of the module `EINT` obtained by declaring all operators as constructors. As in the previous example, we will denote by $\overline{\texttt{EINT}}^m$ the module returned after changing each constructor operator declaration back to a standard operator declaration. Note that some additional variables have been introduced in the test test generation.

```
fmod(
   'EINT,
   nilImportList,
   sort(qidSet('Zero, 'Nat, 'Neg, 'Int, 'ErrorInt)),
   subsortDeclSet(
      subsort('Zero, 'Nat),
      subsort('Nat, 'Int),
      subsort('Zero, 'Neg),
      subsort('Neg, 'Int),
      subsort('Int, 'ErrorInt)),
   opDeclList(
      opDecl('0, nilQidList, 'Zero, emptyAttrSet),
      opDecl('s, 'Nat, 'Nat, emptyAttrSet),
      opDecl('+, qidList('ErrorInt, 'ErrorInt), 'ErrorInt, comm),
      opDecl('*, qidList('ErrorInt, 'ErrorInt), 'ErrorInt, comm),
```

```
    opDecl('s, 'ErrorInt, 'ErrorInt, emptyAttrSet),
    opDecl('p, 'ErrorInt, 'ErrorInt, emptyAttrSet),
    opDecl('p, 'Neg, 'Neg, emptyAttrSet),
    opDecl('+, qidList('ErrorInt, 'ErrorInt), 'ErrorInt, comm),
    opDecl('*, qidList('ErrorInt, 'ErrorInt), 'ErrorInt, comm),
    opDecl('-, 'ErrorInt, 'ErrorInt, emptyAttrSet),
    opDecl('-, 'ErrorInt, 'ErrorInt, emptyAttrSet),
    opDecl('-, 'ErrorInt, 'ErrorInt, emptyAttrSet),
    opDecl('square, 'ErrorInt, 'ErrorInt, emptyAttrSet)),
 varDeclSet(
    varDecl('N, 'Nat), varDecl('M, 'Nat),
    varDecl('I, 'Int), varDecl('J, 'Int),
    varDecl('P, 'Neg), varDecl('Q, 'Neg),
    varDecl('ErrorInt&'0, 'ErrorInt),
    varDecl('ErrorInt&'1, 'ErrorInt)),
 emptyMembAxSet,
 equationSet(
    eq('+['0, 'N], 'N ),
    eq('+['s['N], 'M], 's['+['N, 'M]]),
    eq('*['0, 'N], '0),
    eq('*['s['N], 'M], '+['M, '*['N, 'M]]),
    eq('s['p['I]], 'I),
    eq('p['s['I]], 'I),
    eq('+['I, '0], 'I ),
    eq('+['I, 's['N]], 's['+['I, 'N]]),
    eq('+['I, 'p['Q]], 'p['+['I, 'Q]]),
    eq('*['I, '0], '0),
    eq('*['I, 's['N]], '+['*['I, 'N], 'I]),
    eq('*['I, 'p['Q]], '+['*['I, 'Q], '-['I]]),
    eq('-['0], '0),
    eq('-['s['N]], 'p['-['N]]),
    eq('-['p['Q]], 's['-['Q]]),
    eq('square['I], '*['I, 'I]))
```

And we will call EINTTS the test sets:

```
testSet(
   test('Zero, '0),
   test('Nat, termSet('0, 's['N])),
   test('Neg, termSet('0, 'p['P])),
   test('Int, termSet('0, 's['N], 'p['P])),
   test('ErrorInt,
        termSet('0, '*['ErrorInt&'0, 'ErrorInt&'1], 's['N],
                '+['ErrorInt&'0, 'ErrorInt&'1], 'p['ErrorInt&'0],
                '-['ErrorInt&'0], 'square['ErrorInt&'0])))
```

To prove that the operators mentioned above are the constructors of our original module INT we need to show inductively that EINT satisfies the following goals:

$$(\forall\{\texttt{I}\}).\texttt{s(I): Int}$$
$$(\forall\{\texttt{I}\}).\texttt{p(I): Int}$$

$(\forall\{\texttt{I}\}).\texttt{-I: Int}$

$(\forall\{\texttt{I,J}\}).\texttt{I + J: Int}$

$(\forall\{\texttt{I,J}\}).\texttt{I * J: Int}$

$(\forall\{\texttt{I}\}).\texttt{square(I): Nat}$

Instead of starting with the simpler goals, we will start proving

$$\texttt{EINT} \vdash_{\text{ind}} (\forall\{\texttt{I,J}\}).\texttt{I * J: Int},$$

and we will see that most of the other goals have to be introduced as lemmas, and have to be proved in the main proof. Let us begin introducing the lemma $(\forall\{\texttt{I}\}).\texttt{s(I): Int}$.

red rewInWith(
    $\overline{\texttt{ITP}}^m$,
    ('proveinInitial[
        '1,
        $\overline{\overline{\texttt{EINT}}^m}^m$,
        $\overline{\overline{(\forall\{\texttt{I, J}\}).\texttt{I * J: Int}}^f}^m$,
        $\overline{\texttt{EINTTS}}^m$]),
    nilBindingList,
    applyWithSubst(
        'lemmaIntroduction,
        (('Idx <- '1); ('Beta <- $\overline{\overline{(\forall\{\texttt{I}\}).\texttt{s(I): Int}}^f}^m$)))) .

The original goal is transformed in the subgoals 1.1 and 1.2 as follows.

result StrategyExpression:
rewInWith(
  $\overline{\texttt{ITP}}^m$,
  'goalSet[
    ('proveinInitial[
      ('intString['1, '1]),
      $\overline{\overline{\texttt{EINT} \cup \{\texttt{mb s(I): Int .}\}}^m}^m$,
      $\overline{\overline{(\forall\{\texttt{I,J}\}).\texttt{I * J: Int}}^f}^m$,
      $\overline{\texttt{EINTTS}}^m$]),
    ('proveinInitial[
      ('intString['1, '2]),
      $\overline{\overline{\texttt{EINT}}^m}^m$,
      $\overline{\overline{(\forall\{\texttt{I}\}).\texttt{s(I): Int}}^f}^m$,
      $\overline{\texttt{EINTTS}}^m$])],
  nilBindingList,
  idle)

By test set induction on the subgoal 1.2 applied to the variable I of sort Int, we get three new subgoals, one for each of the terms in the corresponding test set.

red rewInWith(
    $\overline{\texttt{ITP}}^m$,

```
    'goalSet[
      ('proveinInitial[
        ('intString['1, '1]),
        EINT ∪ {mb s(I): Int .}^m^m ,
        (∀{I,J}).I * J: Int^f^m ,
        EINTTS^m]),
      ('proveinInitial[
        ('intString['1, '2]),
        EINT^m^m ,
        (∀{I}).s(I): Int^f^m ,
        EINTTS^m])],
    nilBindingList,
    applyWithSubst(
      'testSetInduction,
      (('X <- ''I) ; ('Idx <- ('intString['1, '2]))))) .


result StrategyExpression:
rewInWith(
  ITP^m,
  'goalSet[
    ('proveinInitial[
      ('intString['1, '1]),
      EINT ∪ {mb s(I): Int .}^m^m ,
      (∀{I,J}).I * J: Int^f^m ,
      EINTTS^m]),
    ('proveinInitial[
      ('intString['1, '2, '1]),
      EINT^m^m ,
      s(0):  Int^f^m ,
      EINTTS^m]),
    ('proveinInitial[
      ('intString['1, '2, '2]),
      EINT^m^m ,
      (∀{N}).s(N): Int ⇒ s(s(N)): Int^f^m ,
      EINTTS^m]),
    ('proveinInitial[
      ('intString['1, '2, '3]),
      EINT^m^m ,
      (∀{P}).s(P): Int ⇒ s(p(P)): Int^f^m ,
      EINTTS^m])],
  nilBindingList,
  idle)
```

These new subgoals, according to the notation introduced in Section 4.1, represent the subgoals:

1.2.1	$\mathtt{EINT} \vdash_{\mathrm{ind}} \mathtt{s(0):\ Int}$
1.2.2	$\mathtt{EINT} \vdash_{\mathrm{ind}} (\forall\{\mathtt{N}\}).\mathtt{s(N):\ Int} \Rightarrow \mathtt{s(s(N)):\ Int}$
1.2.3	$\mathtt{EINT} \vdash_{\mathrm{ind}} (\forall\{\mathtt{P}\}).\mathtt{s(P):\ Int} \Rightarrow \mathtt{s(p(P)):\ Int}$

These subgoals are proved by applying to them the strategy `simplify`.

`red rewInWith(`
  $\overline{\mathtt{ITP}}^{m}$,
  `'goalSet[`
    `('proveinInitial[`
      `('intString['1, '1]),`
      $\overline{\overline{\mathtt{EINT} \cup \{\mathtt{mb\ s(I):\ Int\ .}\}}^{m}}^{m}$,
      $\overline{\overline{(\forall\{\mathtt{I,J}\}).\mathtt{I\ *\ J:\ Int}}^{f}}^{m}$,
      $\overline{\mathtt{EINTTS}}^{m}$`]),`
    `('proveinInitial[`
      `('intString['1, '2, '1]),`
      $\overline{\overline{\mathtt{EINT}}^{m}}^{m}$,
      $\overline{\overline{\mathtt{s(0):\ \ Int}}^{f}}^{m}$,
      $\overline{\mathtt{EINTTS}}^{m}$`]),`
    `('proveinInitial[`
      `('intString['1, '2, '2]),`
      $\overline{\overline{\mathtt{EINT}}^{m}}^{m}$,
      $\overline{\overline{(\forall\{\mathtt{N}\}).\mathtt{s(N):\ Int} \Rightarrow \mathtt{s(s(N)):\ Int}}^{f}}^{m}$,
      $\overline{\mathtt{EINTTS}}^{m}$`]),`
    `('proveinInitial[`
      `('intString['1, '2, '3]),`
      $\overline{\overline{\mathtt{EINT}}^{m}}^{m}$,
      $\overline{\overline{(\forall\{\mathtt{P}\}).\mathtt{s(P):\ Int} \Rightarrow \mathtt{s(p(P)):\ Int}}^{f}}^{m}$,
      $\overline{\mathtt{EINTTS}}^{m}$`])],`
  `nilBindingList,`
  `and(simplify('intString['1, '2, '1]),`
  `and(simplify('intString['1, '2, '2]),`
    `simplify('intString['1, '2, '3])))) .`

`result StrategyExpression:`
`rewInWith(`
  $\overline{\mathtt{ITP}}^{m}$,
  `('proveinInitial[`
    `('intString['1, '1]),`
    $\overline{\overline{\mathtt{EINT} \cup \{\mathtt{mb\ s(I):\ Int\ .}\}}^{m}}^{m}$,
    $\overline{\overline{(\forall\{\mathtt{I,J}\}).\mathtt{I\ *\ J:\ Int}}^{f}}^{m}$,
    $\overline{\mathtt{EINTTS}}^{m}$`]),`
  `nilBindingList,`
  `idle)`

Let us now introduce the lemma $(\forall\{\texttt{I}\}).\texttt{p(I): Int}$. To do so we apply to the subgoal `1.1` the rule `lemmaIntroduction`, with variable `Beta` appropriately instantiated. This subgoal is thus transformed into two new ones.

`red rewInWith(`
  $\overline{\texttt{ITP}}^{m}$,
  `('proveinInitial[`
    `('intString['1, '1]),`
    $\overline{\overline{\texttt{EINT} \cup \{\texttt{mb s(I): Int .}\}}^{m}}^{m}$,
    $\overline{\overline{(\forall\{\texttt{I,J}\}).\texttt{I * J: Int}}^{f}}^{m}$,
    $\overline{\texttt{EINTTS}}^{m}$`]),`
  `nilBindingList,`
  `applyWithSubst(`
    `'lemmaIntroduction,`
    `(('Idx <- ('intString['1, '1]));`
     `('Beta <-` $\overline{\overline{(\forall\{\texttt{I}\}).\texttt{p(I): Int}}^{f}}^{m}$ `))),`

`result StrategyExpression:`
`rewInWith(`
  $\overline{\texttt{ITP}}^{m}$,
  `'goalSet[`
    `('proveinInitial[`
      `('intString['1, '1, '1]),`
      $\overline{\overline{\texttt{EINT} \cup \{\texttt{mb s(I): Int .}\} \cup \{\texttt{mb p(I): Int .}\}}^{m}}^{m}$,
      $\overline{\overline{(\forall\{\texttt{I,J}\}).\texttt{I * J: Int}}^{f}}^{m}$,
      $\overline{\texttt{EINTTS}}^{m}$`]),`
    `('proveinInitial[`
      `('intString['1, '1, '2]),`
      $\overline{\overline{\texttt{EINT} \cup \{\texttt{mb s(I): Int .}\}}^{m}}^{m}$,
      $\overline{\overline{(\forall\{\texttt{I}\}).\texttt{p(I): Int}}^{f}}^{m}$,
      $\overline{\texttt{EINTTS}}^{m}$`])],`
  `nilBindingList,`
  `idle)`

By test set induction applied to the subgoal `1.1.2` on variable `I`, of sort `Int`, we get three more subgoals.

`red rewInWith(`
  $\overline{\texttt{ITP}}^{m}$,
  `'goalSet[`
    `('proveinInitial[`
      `('intString['1, '1, '1]),`
      $\overline{\overline{\texttt{EINT} \cup \{\texttt{mb s(I): Int .}\} \cup \{\texttt{mb p(I): Int .}\}}^{m}}^{m}$,
      $\overline{\overline{(\forall\{\texttt{I,J}\}).\texttt{I * J: Int}}^{f}}^{m}$,
      $\overline{\texttt{EINTTS}}^{m}$`]),`
    `('proveinInitial[`

```
            ('intString['1, '1, '2]),
            EINT ∪ {mb s(I): Int .}^m^m ,
            (∀{I}).p(I): Int^f^m ,
            EINTTS^m ])],
        nilBindingList,
        applyWithSubst(
            'testSetInduction,
            (('X <- ''I) ; ('Idx <- ('intString['1, '1, '2])))))) .


result StrategyExpression:
rewInWith(
   ITP^m ,
   'goalSet[
      ('proveinInitial[
         ('intString['1, '1, '1]),
         EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}^m^m ,
         (∀{I,J}).I * J: Int^f^m ,
         EINTTS^m ]),
      ('proveinInitial[
         ('intString['1, '1, '2, '1]),
         EINT ∪ {mb s(I): Int .}^m^m ,
         p(0):  Int^f^m ,
         EINTTS^m ])],
      ('proveinInitial[
         ('intString['1, '1, '2, '2]),
         EINT ∪ {mb s(I): Int .}^m^m ,
         (∀{N}).p(N): Int ⇒ p(s(N)): Int^f^m ,
         EINTTS^m ])],
      ('proveinInitial[
         ('intString['1, '1, '2, '3]),
         EINT ∪ {mb s(I): Int .}^m^m ,
         (∀{P}).p(P): Int ⇒ p(p(P)): Int^f^m ,
         EINTTS^m ])],
   nilBindingList,
   idle)
```

The subgoals `1.1.2.1`, `1.1.2.2` and `1.1.2.3` can then be proved using the strategy `simplify`.

```
red rewInWith(
        ITP^m ,
        'goalSet[
            ('proveinInitial[
             ('intString['1, '1, '1]),
             EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}^m^m ,
```

```
               (∀{I,J}).I * J: Int^f^m ,
               EINTTS^m ]),
             ('proveinInitial[
               ('intString['1, '1, '2, '1]),
               EINT ∪ {mb s(I): Int .}^m^m ,
               p(0): Int^f^m ,
               EINTTS^m ])],
             ('proveinInitial[
               ('intString['1, '1, '2, '2]),
               EINT ∪ {mb s(I): Int .}^m^m ,
               (∀{N}).p(N): Int ⇒ p(s(N)): Int^f^m ,
               EINTTS^m ])],
             ('proveinInitial[
               ('intString['1, '1, '2, '3]),
               EINT ∪ {mb s(I): Int .}^m^m ,
               (∀{P}).p(P): Int ⇒ p(p(P)): Int^f^m ,
               EINTTS^m ])],
        nilBindingList,
        and(simplify('intString['1, '1, '2, '1]),
        and(simplify('intString['1, '1, '2, '2]),
            simplify('intString['1, '1, '2, '3]))),


result StrategyExpression:
rewInWith(
   ITP^m ,
   ('proveinInitial[
      ('intString['1, '1, '1]),
      EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}^m^m ,
      (∀{I,J}).I * J: Int^f^m ,
      EINTTS^m ]),
   nilBindingList,
   idle)
```

Introducing now the lemma (∀{I,J}).I + J: Int we get two new subgoals.

```
red rewInWith(
      ITP^m ,
      ('proveinInitial[
          ('intString['1, '1, '1]),
          EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}^m^m ,
          (∀{I,J}).I * J: Int^f^m ,
          EINTTS^m ]),
      nilBindingList,
      applyWithSubst(
          'lemmaIntroduction,
          (('Idx <- ('intString['1, '1, '1]));
```

```
          ('Beta <- (∀{I,J}).I + J: Int^f^m )))) .
```

```
result StrategyExpression:
rewInWith(
   ITP^m,
   'goalSet[
      ('proveinInitial[
         ('intString['1, '1, '1, '1]),
         EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}
              ∪ {mb I + J: Int .}^m^m ,
         (∀{I,J}).I * J: Int^f^m ,
         EINTTS^m]),
      ('proveinInitial[
         ('intString['1, '1, '1, '2]),
         EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}^m^m ,
         (∀{I,J}).I + J: Int^f^m ,
         EINTTS^m])],
   nilBindingList,
   idle)
```

Let us apply now the rule `testSetInduction` on the variable `J`, of sort `Int`, on the subgoal `1.1.1.2`.

```
red rewInWith(
       ITP^m,
       'goalSet[
            ('proveinInitial[
              ('intString['1, '1, '1, '1]),
              EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}
                   ∪ {mb I + J: Int .}^m^m ,
              (∀{I,J}).I * J: Int^f^m ,
              EINTTS^m]),
            ('proveinInitial[
              ('intString['1, '1, '1, '2]),
              EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}^m^m ,
              (∀{I,J}).I + J: Int^f^m ,
              EINTTS^m])],
       nilBindingList,
       applyWithSubst(
            'testSetInduction,
            (('X <- ''J) ; ('Idx <- ('intString['1, '1, '1, '2]))))) .
```

We get three new subgoals:

```
result StrategyExpression:
```

```
rewInWith(
   ITP^m,
   'goalSet[
       ('proveinInitial[
        ('intString['1, '1, '1, '1]),
        EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}
             ∪ {mb I + J: Int .}^m^m ,
        (∀{I,J}).I * J: Int^f^m ,
        EINTTS^m]),
       ('proveinInitial[
        ('intString['1, '1, '1, '2, '1]),
        EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}^m^m ,
        (∀{I}).I + 0:  Int^f^m ,
        EINTTS^m]),
       ('proveinInitial[
        ('intString['1, '1, '1, '2, '2]),
        EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}^m^m ,
        (∀{N}).((∀{I}).I + N: Int ⇒ I + s(N): Int)^f^m ,
        EINTTS^m]),
       ('proveinInitial[
        ('intString['1, '1, '1, '2, '3]),
        EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}^m^m ,
        (∀{P}).((∀{I}).I + P: Int ⇒ I + p(P): Int)^f^m ,
        EINTTS^m])],
   nilBindingList,
   idle)
```

Again, using the `simplify` strategy we can prove these three subgoals. In the same step we will introduce the last needed lemma, (∀{`I`}).`-I: Int`, getting subgoal `1.1.1.1` transformed into two new subgoals.

```
red rewInWith(
      ITP^m,
      'goalSet[
          ('proveinInitial[
           ('intString['1, '1, '1, '1]),
           EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}S
                ∪ {mb I + J: Int .}^m^m ,
           (∀{I,J}).I * J: Int^f^m ,
           EINTTS^m]),
          ('proveinInitial[
           ('intString['1, '1, '1, '2, '1]),
           EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}^m^m ,
           (∀{I}).I + 0:  Int^f^m ,
```

```
            EINTTS^m]),
          ('proveinInitial[
            ('intString['1, '1, '1, '2, '2]),
            EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}^m^m ,
            (∀{N}).((∀{I}).I + N: Int ⇒ I + s(N): Int)^f^m ,
            EINTTS^m]),
          ('proveinInitial[
            ('intString['1, '1, '1, '2, '3]),
            EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}^m^m ,
            (∀{P}).((∀{I}).I + P: Int ⇒ I + p(P): Int)^f^m ,
            EINTTS^m])],
      nilBindingList,
      and(simplify('intString['1, '1, '1, '2, '1]),
      and(simplify('intString['1, '1, '1, '2, '2]),
      and(simplify('intString['1, '1, '1, '2, '3]),
          applyWithSubst(
            'lemmaIntroduction,
            (('Idx <- ('intString['1, '1, '1, '1]));
             ('Beta <- (∀{I}).-(I): Int^f^m )))))) .

result StrategyExpression:
rewInWith(
   ITP^m,
   'goalSet[
      ('proveinInitial[
         ('intString['1, '1, '1, '1, '1]),
         EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}
              ∪ {mb I + J: Int .} ∪ {mb -I: Int .}^m^m ,
         (∀{I,J}).I * J: Int^f^m ,
         EINTTS^m]),
      ('proveinInitial[
         ('intString['1, '1, '1, '1, '2]),
         EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}
              ∪ {mb I + J: Int .}^m^m ,
         (∀{I}).-I: Int^f^m ,
         EINTTS^m])],
   nilBindingList,
   idle)
```

Applying the rule `testSetInduction` to the subgoal `1.1.1.1.2` we obtain three new subgoals, which can be proved by simplification, i.e., by applying to them the `simplify` strategy.

```
red rewInWith(
      ITP^m,
      'goalSet[
```

```
      ('proveinInitial[
         ('intString['1, '1, '1, '1, '1]),
         EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}
              ∪ {mb I + J: Int .} ∪ {mb -I: Int .}^m^m ,
         (∀{I,J}).I * J: Int^f^m ,
         EINTTS^m]),
      ('proveinInitial[
         ('intString['1, '1, '1, '1, '2]),
         EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}
              ∪ {mb I + J: Int .}^m^m ,
         (∀{I}).-I: Int^f^m ,
         EINTTS^m])],
   nilBindingList,
   applyWithSubst(
         'testSetInduction,
         (('X <- ''I) ;
          ('Idx <- ('intString['1, '1, '1, '1, '2]))))) .

result StrategyExpression:
rewInWith(
   ITP^m ,
   'goalSet[
      ('proveinInitial[
         ('intString['1, '1, '1, '1, '1]),
         EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}
              ∪ {mb I + J: Int .} ∪ {mb -I: Int .}^m^m ,
         (∀{I,J}).I * J: Int^f^m ,
         EINTTS^m]),
      ('proveinInitial[
         ('intString['1, '1, '1, '1, '2, '1]),
         EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}
              ∪ {mb I + J: Int .}^m^m ,
         -0: Int^f^m ,
         EINTTS^m])],
      ('proveinInitial[
         ('intString['1, '1, '1, '1, '2, '2]),
         EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}
              ∪ {mb I + J: Int .}^m^m ,
         (∀{N}).-N: Int ⇒ -s(N): Int^f^m ,
         EINTTS^m])],
      ('proveinInitial[
         ('intString['1, '1, '1, '1, '2, '3]),
         EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}
              ∪ {mb I + J: Int .}^m^m ,
```

```
        (∀{P}).-P: Int ⇒ -p(P): Int^f^m ,
        EINTTS^m])],
   nilBindingList,
   idle)
```

Simplifying again:

```
red rewInWith(
       ITP^m,
       'goalSet[
            ('proveinInitial[
              ('intString['1, '1, '1, '1, '1]),
              EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}
                   ∪ {mb I + J: Int .} ∪ {mb -I: Int .}^m^m ,
              (∀{I,J}).I * J: Int^f^m ,
              EINTTS^m]),
            ('proveinInitial[
              ('intString['1, '1, '1, '1, '2, '1]),
              EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}
                   ∪ {mb I + J: Int .}^m^m ,
              -0: Int^f^m ,
              EINTTS^m]),
            ('proveinInitial[
              ('intString['1, '1, '1, '1, '2, '2]),
              EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}
                   ∪ {mb I + J: Int .}^m^m ,
              (∀{N}).-N: Int ⇒ -s(N): Int^f^m ,
              EINTTS^m]),
            ('proveinInitial[
              ('intString['1, '1, '1, '1, '2, '3]),
              EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}
                   ∪ {mb I + J: Int .}^m^m ,
              (∀{P}).-P: Int ⇒ -p(P): Int^f^m ,
              EINTTS^m])],
       nilBindingList,
       and(simplify('intString['1, '1, '1, '1, '2, '1]),
       and(simplify('intString['1, '1, '1, '1, '2, '2]),
           simplify('intString['1, '1, '1, '1, '2, '3])))) .
```

The result is:

```
result StrategyExpression:
rewInWith(
   ITP^m,
   ('proveinInitial[
```

```
     ('intString['1, '1, '1, '1, '1]),
     EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}
        ∪ {mb I + J: Int .} ∪ {mb -I: Int .}^m^m ,
     (∀{I,J}).I * J: Int^f^m ,
     EINTTS^m]),
   nilBindingList,
   idle)
```

The only subgoal remaining is `1.1.1.1.1`, which corresponds to

```
1.1.1.1.1    EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}
                   ∪ {mb I + J: Int .} ∪ {mb -I: Int .}
                ⊢ind (∀{I,J}).I * J: Int
```

It can be proved by applying to it the rule `testSetInduction` on variable `J`, and then simplifying the subgoals; or using the strategy `induction` on this variable directly. Here we just apply the strategy `induction`, and with that the proof is completed.

```
red rewInWith(
        ITP^m,
        ('proveinInitial[
            ('intString['1, '1, '1, '1, '1]),
            EINT ∪ {mb s(I): Int .} ∪ {mb p(I): Int .}
                 ∪ {mb I + J: Int .} ∪ {mb -I: Int .}^m^m ,
            (∀{I,J}).I * J: Int^f^m ,
            EINTTS^m]),
        nilBindingList,
        induction(''J)) .

result StrategyExpression:
      rewInWith(ITP, 'emptyGoalSet, nilBindingList, idle)
```

The only goal that remains to be proved is (∀{I}).`square(I): Nat`. Instead of showing the interaction with the prover as for the previous goal, we will do it "in one shot", that is, without showing the intermediate steps.

```
red rewInWith(ITP^m,
        ('proveinInitial[
           '1,
           EINT^m^m ,
           (∀{I}).square(I): Nat^f^m ,
           EINTTS^m]),
        nilBindingList,
        and(applyWithSubst(
              'lemmaIntroduction,
              (('Idx <- ('1));
               ('Beta <- (∀{I}).I * I: Nat^f^m ))),
```

```
        and(simplify(('intString['1, '1])),
        and(applyWithSubst(
              'lemmaIntroduction,
              (('Idx <- ('intString['1, '2]));
               ('Beta <- (∀{N, M}).N + M: Nat ))),
        and(applyWithSubst(
              'testSetInduction,
              (('X <- ''N) ; ('Idx <- ('intString['1, '2, '2])))),
        and(simplify('intString['1, '2, '2, '1]),
        and(simplify('intString['1, '2, '2, '2]),
        and(applyWithSubst(
              'lemmaIntroduction,
              (('Idx <- ('intString['1, '2, '1]));
              ('Beta <- (∀{P}).-(P): Nat ))),
        and(applyWithSubst(
              'testSetInduction,
              (('X <- ''P) ; ('Idx <- ('intString['1, '2, '1, '2])))),
        and(simplify('intString['1, '2, '1, '2, '1]),
        and(simplify('intString['1, '2, '1, '2, '2]),
        and(applyWithSubst(
              'lemmaIntroduction,
              (('Idx <- ('intString['1, '2]));
               ('Beta <- (∀{N, M}).N * M: Nat ))),
        and(applyWithSubst(
              'testSetInduction,
              (('X <- ''N) ; ('Idx <- ('intString['1, '2, '1, '1, '2])))),
        and(simplify('intString['1, '2, '1, '1, '2, '1]),
        and(simplify('intString['1, '2, '1, '1, '2, '2]),
        and(applyWithSubst(
              'lemmaIntroduction,
              (('Idx <- ('intString['1, '2, '1, '1, '1]));
               ('Beta <- (∀{P, Q}).P * Q: Nat ))),
        and(applyWithSubst(
              'testSetInduction,
              (('X <- ''I) ;
               ('Idx <- ('intString['1, '2, '1, '1, '1, '1])))),
        and(simplify('intString['1, '2, '1, '1, '1, '1, '1]),
        and(simplify('intString['1, '2, '1, '1, '1, '1, '2]),
        and(simplify('intString['1, '2, '1, '1, '1, '1, '3]),
        and(applyWithSubst(
              'testSetInduction,
              (('X <- ''Q) ;
               ('Idx <- ('intString['1, '2, '1, '1, '1, '2])))),
        and(simplify('intString['1, '2, '1, '1, '1, '2, '1]),
            simplify('intString['1, '2, '1, '1, '1, '2, '2]))))))))))))))))))))))) .

rewrites:  1079 in 709ms cpu (719ms real) (1519 rewrites/second)
result StrategyExpression:
```

```
rewInWith(ITP^m, 'emptyGoalSet, nilBindingList, idle)
```

4.5 Test Sets Generation

Once we have proved that a subsignature of constructors sufficiently generates the initial algebra of a specification, we are justified in using only those constructors to associate to each sort a test set.

Of course, the choice of what terms to use for a test set is not unique. For example, in a specification of the natural numbers with a sort `NzNat` of nonzero natural numbers and constructors `0 : -> Nat`, and `s : Nat -> NzNat`, we could use as test set for the sort `NzNat` the term

```
s(N)
```

with `N` a variable of sort `Nat`, but we could also have used as test set of sort `NzNat` the terms

```
s(0)      s(s(N))
```

Our test set generation function chooses the first, simpler alternative. That is, it always yields terms of depth zero or one in the test set of each sort. Such terms are obtained by inspecting what constructors have sort smaller or equal to the sort in question; for each such constructor $\mathtt{f} : \mathtt{s}_1 \ldots \mathtt{s}_n \to \mathtt{s}$ a term $\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ with $\mathtt{x}_i\colon \mathtt{s}_i$ is then added to the test set. As an optimization that avoids unnecessary subgoals in inductive proofs, if a constructor has two overloadings, with the rank of one of them smaller or equal to that of the other, and both with coarities smaller or equal to the sort in question, then only the most general constructor is used in generating the test set for that sort.

Given a subsignature of constructors for which sufficient generation has been proved, the function `generateTestSets` takes a module and generates the test sets for each sort in it. To be able to distinguish the constructors in the module passed to this function we have defined a new sort `EModule` (for extended module) as a supersort of `Module`. The constructor for this sort is:

```
op fmod : Qid ImportList SortDecl SubsortDeclSet EOpDeclList
          VarDeclSet MembAxSet EquationSet -> EModule .
```

Where `EOpDeclList` is a new list sort declared as supersort of `EOpDecl`, which is declared as a supersort of `OpDecl`. We declare also the following constructor:

```
op copDecl : Qid QidList Qid AttrSet -> EOpDecl .
```

Then we will have that the declarations of operations is given by a list of elements of sort `EOpDecl`, some of them using the constructor `opDecl` and others `copDecl`. We assume then that constructors in the specification are declared using `copDecl` and non-constructor operations using `opDecl`. To summarize, we have the following declarations:

```
sorts EModule EOpDecl EOpDeclList  .

subsort Module < EModule .
subsort OpDecl < EOpDecl .
subsort EOpDecl < EOpDeclList .
```

```
op copDecl : Qid QidList Qid AttrSet -> EOpDecl .
op nilEOpDeclList : -> EOpDeclList .
op eOpDeclList : EOpDeclList EOpDeclList -> EOpDeclList
      [assoc id: nilEOpDeclList] .

op fmod : Qid ImportList SortDecl SubsortDeclSet EOpDeclList
         VarDeclSet MembAxSet EquationSet -> EModule .
```

Then, for example, we have that (the meta-representation of) a specification for the natural numbers with constructors zero and successor and non-constructor operation addition can be given as:

```
fmod('NAT,
      nilImportList,
      sort(qidSet('Nat, 'NzNat)),
      subsort('NzNat, 'Nat),
      eOpDeclList(
         copDecl('0, nilQidList, 'Nat, emptyAttrSet),
         copDecl('s, 'Nat, 'NzNat, emptyAttrSet),
         opDecl('+, qidList('Nat, 'Nat), 'Nat, emptyAttrSet)),
      varDeclSet(
         varDecl('N, 'Nat), varDecl('M, 'Nat)),
      emptyMembAxSet,
      equationSet(
      eq('+['N, '0], 'N),
      eq('+['N, 's['M]], 's['+['N, 'M]])))
```

The function that generates the test sets is `generateTestSets`, which has been declared as:

```
op generateTestSets : EModule -> TestSetSolution .
```

where `TestSetSolution` is a new sort with constructor:

```
op testSetSolution : EModule TestSet -> TestSetSolution .
```

That is, the function returns a pair consisting of a module of sort `EModule` and the family of test sets generated.

It may be that in the process of generating the test sets new variables have to be added to the original specification. Therefore, the test sets generated have to be used together with this new module. However, note that a module of sort `Module`, and not `EModule`, has to be passed to the theorem prover, so this module must be transformed before it is used in the ITP.

The `generateTestSets` function is located in the module `ITP`, so, given an extended module EM, the way in which it has to be called is the following:

$$\texttt{rew meta-reduce(}\overline{\texttt{ITP}}^{m}\texttt{, 'generateTestSets[}\overline{\overline{EM}^{m}}^{m}\texttt{]) .}$$

For example, the call with the module `NAT` would be:

```
rew meta-reduce(ITP,
  'generateTestSets[
```

```
    ('fmod[''NAT,
       'nilImportList,
       ('sort['qidSet[''Nat, ''NzNat]]),
       ('subsort[''NzNat, ''Nat]),
       ('eOpDeclList[
          ('copDecl[''0, 'nilQidList, ''Nat, 'emptyAttrSet]),
          ('copDecl[''s, ''Nat, ''NzNat, 'emptyAttrSet]),
          ('opDecl[''+, ('qidList[''Nat, ''Nat]), ''Nat, 'emptyAttrSet])]),
       ('varDeclSet[
          ('varDecl[''N, ''Nat]), ('varDecl[''M, ''Nat])]),
       'emptyMembAxSet,
       ('equationSet[
          ('eq[('_[_][''+, ('_,_[''N, ''0])]), ''N]),
          ('eq[('_[_][''+, ('_,_[''N, ('_[_][''s, ''M])])]),
               ('_[_][''s, ('_[_][''+, ('_,_[''N, ''M])])])])])])]) .
```

And the result:

```
rewrites: 83 in 30ms cpu (30ms real) (2766 rewrites/second)
result Term:
'testSetSolution[
   ('fmod[''NAT,
      'nilImportList,
      ('sort['qidSet[''Nat, ''NzNat]]),
      ('subsort[''NzNat, ''Nat]),
      ('eOpDeclList[
         ('copDecl[''0, 'nilQidList, ''Nat, 'emptyAttrSet]),
         ('copDecl[''s, ''Nat, ''NzNat, 'emptyAttrSet]),
         ('opDecl[''+, ('qidList[''Nat, ''Nat]), ''Nat, 'emptyAttrSet])]),
      ('varDeclSet[
         ('varDecl[''N, ''Nat]), ('varDecl[''M, ''Nat])]),
      'emptyMembAxSet,
      ('equationSet[
         ('eq[('_[_][''+, ('_,_[''N, ''0])]), ''N]),
         ('eq[('_[_][''+, ('_,_[''N, ('_[_][''s, ''M])])]),
              ('_[_][''s, ('_[_][''+, ('_,_[''N, ''M])])])])])]),
   ('testSet[
         ('test[''Nat, ('termSet[''0, ('_[_][''s, ''X])])]),
         ('test[''NzNat, ('_[_][''s, ''X])])])]
```

Let us see another example to illustrate the need for introducing new variables. This is a specification of natural numbers with zero and successor:

```
rew meta-reduce(ITP,
    'generateTestSets[
      ('fmod[''NAT,
         'nilImportList,
         ('sort['qidSet[''Nat, ''NzNat]]),
         ('subsort[''NzNat, ''Nat]),
         ('eOpDeclList[
```

```
            ('copDecl[''0, 'nilQidList, ''Nat, 'emptyAttrSet]),
            ('copDecl[''s, ''Nat, ''NzNat, 'emptyAttrSet])]),
         'emptyVarDeclSet,
         'emptyMembAxSet,
         'emptyEquationSet])]) .

rewrites: 73 in 10ms cpu (54ms real) (7300 rewrites/second)
result Term:
'testSetSolution[
       ('fmod[''NAT,
          'nilImportList,
          ('sort['qidSet[''Nat, ''NzNat]]),
          ('subsort[''NzNat, ''Nat]),
          ('eOpDeclList[
             ('copDecl[''0, 'nilQidList, ''Nat, 'emptyAttrSet]),
             ('copDecl[''s, ''Nat, ''NzNat, 'emptyAttrSet])]),
          ('varDecl[''Nat&'0, ''Nat]),
          'emptyMembAxSet,
          'emptyEquationSet]),
       ('testSet[
          ('test[''Nat, ('termSet[''0, ('_[_]['s, ''Nat&'0])])]),
          ('test[''NzNat, ('_[_][''s, ''Nat&'0])])])]
```

Note that a new variable `Nat&'0` of sort `Nat` has been introduced. It is clear that in this case there were not enough variables to generate the test sets.

The specification is as follows:

```
op generateTestSets : EModule -> TestSetSolution .
op generateTestSetsAux : Module QidSet MachineInt EOpDeclList VarDeclSet
       TestSet -> TestSetSolution .
op test2 : Module Qid MachineInt EOpDeclList VarDeclSet TermSet
           -> IndexVarsTests .
op test3 : MachineInt QidList VarDeclSet -> IndexVarsTerms .
op test4 : QidList VarDeclSet IndexVarsTerms -> IndexVarsTerms .
op maximalTermSet : TermSet Module -> TermSet .
op maximalTermSetAux : Term TermSet TermSet Module -> TermSet .
op unExtend : EOpDeclList -> OpDeclList .
op newVariable : Qid MachineInt -> Qid .
op thereIsVariable? : VarDeclSet Qid -> Bool .

eq generateTestSets(fmod(QI, IL, sort(SS), SSDS, EODL, VDS, MAS, ES))
  = generateTestSetsAux(
       fmod(QI, IL, sort(SS), SSDS, unExtend(EODL), VDS, MAS, ES),
       SS, 0, EODL, VDS, emptyTestSet) .

eq generateTestSetsAux(Th, qidSet(S, SS), N, EODL, VDS, TSPS)
  = generateTestSetsAux(
       Th, SS, indexIVTests(test2(Th, S, N, EODL, VDS, emptyTermSet)),
       EODL, variablesIVTests(test2(Th, S, N, EODL, VDS, emptyTermSet)),
       testSet(
```

```
          test(S,
             maximalTermSet(
                testsIVTests(test2(Th, S, N, EODL, VDS, emptyTermSet)),
                setUpVariables(Th,
                   variablesIVTests(
                      test2(Th, S, N, EODL, VDS, emptyTermSet))))),
          TSPS)) .
eq generateTestSetsAux(
      fmod(QI, IL, sort(SS), SSDS, ODL, VDS, MAS, ES),
      emptyQidSet, N, EODL, VDS', TSPS)
  = testSetSolution(fmod(QI, IL, sort(SS), SSDS, EODL, VDS', MAS, ES),
                    TSPS) .

ceq test2(Th, S, N, eOpDeclList(copDecl(F, SL, S', AS), EODL), VDS, TS)
  = if sortLeq(Th, S', S)
    then test2(Th, S, indexIVTerms(test3(N, SL, VDS)), EODL,
               varDeclSet(VDS, variablesIVTerms(test3(N, SL, VDS))),
               termSet(TS, F[termsIVTerms(test3(N, SL, VDS))]))
    else test2(Th, S, N, EODL, VDS, TS)
    fi
    if SL =/= nilQidList .
eq test2(Th, S, N, eOpDeclList(copDecl(F, nilQidList, S', AS), EODL), VDS, TS)
  = if sortLeq(Th, S', S)
    then test2(Th, S, N, EODL, VDS, termSet(TS, F))
    else test2(Th, S, N, EODL, VDS, TS)
    fi .
eq test2(Th, S, N, eOpDeclList(opDecl(F, SL, S', AS), EODL), VDS, TS)
  = test2(Th, S, N, EODL, VDS, TS) .
ceq test2(Th, S, N, copDecl(F, SL, S', AS), VDS, TS)
  = if sortLeq(Th, S', S)
    then iVTests(indexIVTerms(test3(N, SL, VDS)),
                 varDeclSet(VDS, variablesIVTerms(test3(N, SL, VDS))),
                 termSet(TS, F[termsIVTerms(test3(N, SL, VDS))]))
    else iVTests(N, VDS, TS)
    fi
    if SL =/= nilQidList .
eq test2(Th, S, N, copDecl(F, nilQidList, S', AS), VDS, TS)
  = if sortLeq(Th, S', S)
    then iVTests(N, VDS, termSet(TS, F))
    else iVTests(N, VDS, TS)
    fi .
eq test2(Th, S, N, opDecl(F, SL, S', AS), VDS, TS)
  = iVTests(N, VDS, TS) .
eq test2(Th, S, N, nilEOpDeclList, VDS, TS)
  = iVTests(N, VDS, TS) .

eq test3(N, S, varDeclSet(varDecl(X, S), VDS))
  = iVTerms(N, emptyVarDeclSet, X) .
ceq test3(N, S, VDS)
  = iVTerms((N + 1), varDecl(newVariable(S, N), S), newVariable(S, N))
```

```
    if not(thereIsVariable?(VDS, S)) .
eq test3(N, qidList(S, SL), varDeclSet(varDecl(X, S), VDS))
  = test4(SL, VDS, iVTerms(N, emptyVarDeclSet, X)) .
ceq test3(N, qidList(S, SL), VDS)
  = test4(SL, VDS,
          iVTerms((N + 1), varDecl(newVariable(S, N), S),
                  newVariable(S, N)))
    if not(thereIsVariable?(VDS, S)) .

eq test4(qidList(S, SL),
         varDeclSet(varDecl(X, S), VDS),
         iVTerms(N, VDS', TL))
  = test4(SL, VDS, iVTerms(N, VDS', (TL, X))) .
ceq test4(qidList(S, SL), VDS, iVTerms(N, VDS', TL))
  = test4(SL, VDS,
          iVTerms((N + 1),
                  varDeclSet(varDecl(newVariable(S, N), S), VDS'),
                  (TL, newVariable(S, N))))
    if not(thereIsVariable?(VDS, S)) .
eq test4(S, varDeclSet(varDecl(X, S), VDS),
         iVTerms(N, VDS', TL))
  = iVTerms(N, VDS', (TL, X)) .
ceq test4(S, VDS, iVTerms(N, VDS', TL))
  = iVTerms((N + 1),
            varDeclSet(varDecl(newVariable(S, N), S), VDS'),
            (TL, newVariable(S, N)))
    if not(thereIsVariable?(VDS, S)) .
eq test4(nilQidList, VDS, iVTerms(N, VDS', TL))
  = iVTerms(N, VDS', TL) .

ceq unExtend(eOpDeclList(OD, EODL))
  = opDeclList(OD, unExtend(EODL))
    if EODL =/= nilEOpDeclList .
ceq unExtend(eOpDeclList(copDecl(F, SL, S, AS), EODL))
  = opDeclList(opDecl(F, SL, S, AS), unExtend(EODL))
    if EODL =/= nilEOpDeclList .
eq unExtend(eOpDeclList(OD, nilEOpDeclList))
  = OD .
eq unExtend(eOpDeclList(copDecl(F, SL, S, AS), nilEOpDeclList))
  = opDecl(F, SL, S, AS) .
eq unExtend(nilEOpDeclList)
  = nilOpDeclList .

eq newVariable(S, N)
  = conc(S, index('&', N)) .

eq thereIsVariable?(varDeclSet(varDecl(X, S), VDS), S')
  = orElse(S == S', thereIsVariable?(VDS, S')) .
eq thereIsVariable?(emptyVarDeclSet, S)
  = false .
```

5 The Church-Rosser Checker

The goal of the Church-Rosser checker tool is to allow the user to check that his/her specification is "correct" and that yields a "computable" abstract data type by rewriting. The main idea we had in mind in the development of the tool was that the specification should be kept as given by the user, that is, as the user understands it. In our opinion transforming the user's specification into an equivalent large set of hard-to-understand equations is not an acceptable output. Our approach is then to give as output a set of non reducible critical pairs and a set of membership proof obligations. Then the user, with this information, can decide how to either prove such proof obligations or to modify his specification. Methodological suggestions for how a user can use the output of the Church-Rosser checker to modify his/her specification are given in Sections 5.5 and 5.6.

The most common situation for large examples is that, although the user's specification implements correctly a decidable data type by rewriting, the tool will output a nonempty set of proof obligations. This is because, typically, the user's specification will be *ground* Church-Rosser, that is, Church-Rosser for ground terms. But this can often happen without the equations being confluent or sort-decreasing for arbitrary terms with variables.

We discuss in this section the design and implementation of the tool and illustrate its use with examples. Its inputs are conditional order-sorted specifications in which some operators may be commutative, and for which a proof of termination of their equations as rewrite rules has already been performed. The corresponding output is a set of critical pairs and a set of membership assertions that must (respectively) be shown ground-joinable, and ground-rewritable to a term with the required sort.

The need to have the input modules completely defined equationally, with explicit definitions for all the operations, sorts and sort relations, force us to forbid the use of any built-in sort or operation in the specification given as an input to the Church-Rosser checker. The user cannot use any of the built-ins, including the sort `Bool` and any other sort or operation not explicitly defined in his/her specification. The actual state of development of the Maude system does not support the use of ad-hoc overloading, and so, the user cannot use the reserved names `Bool`, `true` and `false`. The use of quotes inside the identifiers is also prohibited. In addition to this, the input module has to be a flat module, that is, the importation of any other module, either built-in or user-defined, is forbidden. We plan to study the way of handling built-ins and module importations in the user's module in the future, but we lack of a solution at this moment.

5.1 Some Definitions

Given an index set S, an S-sorted set A is a family of sets A_s for each $s \in S$; we will denote such an S-sorted set by $\{A_s | s \in S\}$. Similarly, given S-sorted sets A and B, an S-sorted function $f\colon A \to B$ is an S-sorted family $f = \{f_s\colon A_s \to B_s | s \in S\}$.

Definition 1 *(Goguen and Meseguer [9]) A many-sorted signature is a pair (Σ, S), where S is a sort set and Σ is an $S^* \times S$-indexed family of operation symbols $\{\Sigma_{w,s} | w \in S^*, s \in S\}$. An order-sorted signature is a triple $(S, \leq, \Sigma)$ such that (S, Σ) is a many-sorted signature and $(S, \leq)$ is a poset.*

Definition 2 *(Goguen and Meseguer [9]) Given an order-sorted signature $(S, \leq, \Sigma)$, we construct the order-sorted Σ-term algebra $\mathcal{T}_\Sigma$ as the least family $\{\mathcal{T}_{\Sigma,s} | s \in S\}$ of* *sets satisfying the following conditions:*

- $\Sigma_{\lambda,s} \subseteq \mathcal{T}_{\Sigma,s}$ *for* $s \in S$*;*
- $\mathcal{T}_{\Sigma,s'} \subseteq \mathcal{T}_{\Sigma,s}$ *if* $s' \leq s'$*;*
- *if* $\sigma \in \Sigma_{w,s}$ *and if* $t_i \in \mathcal{T}_{\Sigma,s_i}$ *where* $w = s_1 \ldots s_n \neq \lambda$*, then (the string)* $\sigma(t_1 \ldots t_n) \in \mathcal{T}_{\Sigma,s}$*, and*
- *for* $\sigma \in \Sigma_{w,s}$ *and if* $t_i \in \mathcal{T}_{\Sigma,s_i}$ *let* $\mathcal{T}_\sigma : \mathcal{T}_w \to \mathcal{T}_s$ *send* $t_1, \ldots, t_n$ *to (the string)* $\sigma(t_1 \ldots t_n)$*.*

Then, an order-sorted signature Σ *is* preregular *if for every* $t \in \mathcal{T}_\Sigma$ *there is a least* $s \in S$*, called the* least sort *of* t *and denoted* $LS(t)$*, such that* $t \in \mathcal{T}_{\Sigma,s}$*.*

We restrict ourselves to the class of preregular signatures.

The set of variables occurring in a term t is denoted $vars(t)$.

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 $t \in \mathcal{T}_\Sigma(X)$ 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 term t with its subterm $t|_p$ replaced by the term t' is denoted $t[t']_p$.

A Σ-substitution σ is a mapping from a finite set of variables $D(\sigma)$ called its *domain* such that for all variable x of sort s in $D(\sigma)$, $\sigma(x)$ is a term of sort s. The image $I(\sigma)$ is defined as $I(\sigma) = \cup_{x \in D(\sigma)} vars(\sigma(x))$.

We denote by $\preceq_E$ the subsumption preorder on $T(\Sigma, X)$ defined by: $t \preceq_E t'$ if and only if $t' =_E \sigma(t')$ for a substitution σ called a *match* from t to t'. Given a subset V of X, we define $\sigma \preceq_E \sigma'[V]$ if and only if $\exists \sigma''$ such that $\forall x \in V,\ \sigma'' \cdot \sigma(x) =_E \sigma'(x)$.

A substitution is uniquely defined by a mapping from variables to terms, and is written out as $\{x_1 \leftarrow s_1, \ldots, x_n \leftarrow s_n\}$.

We say that a term t *overlaps* another term with distinct variables t' if there is a non-variable subterm $t'|_p$ of t' for some position p such that t and $t'|_p$ have a common upper bound with respect to subsumption.

Definition 3 *Given two conditional rewrite rules* $L \to R$ `if` $\bar{U} \downarrow \bar{V}$ *and* $L' \to R'$ `if` $\bar{U'} \downarrow \bar{V'}$ *such that* $vars(L) \cap vars(L') = \emptyset$ *and* $L|_p\sigma = L'\sigma$*, for some non-variable position* $p \in \mathcal{P}_M(L)$ *and unifier* σ*, then* $ccp(L\sigma[R'\sigma]_p,\ R\sigma,\ \bar{U} \downarrow \bar{V} \wedge \bar{U'} \downarrow \bar{V'})$ *is a critical pair.*

Note that we use order-sorted unification with commutative axioms, considering all the unifiers, not only the most general ones. Note also that we accumulate the conditions of both rules they originated as in [1].

A critical pair $ccp(t, t', c)$ is more general than another critical pair $ccp(u, u', c')$ if there exists a substitution σ such that $t\sigma = u$, $t'\sigma = u'$ and $c\sigma = c'$.

Then, given a specification $\mathcal{S}$, let $CCP(\mathcal{S})$ denote the set of most general critical pairs between rules in $\mathcal{S}$ that, after simplifying both sides of the pair using the equations in $\mathcal{S}$ are not identical critical pairs of the form $ccp(t, t, c)$. To determine the overlaps of the lefthand sides of the rules the variables in them have to be renamed somehow in order to get disjoint sets of variables appearing in each of them.

Given an order-sorted signature $(S, \leq, \Sigma)$, a sorted set of variables X can be viewed as a pair $(\bar{X}, \mu)$ where $\bar{X}$ is a set of variable names and μ is a sort assignment $\mu \colon \bar{X} \to S$. Thus, a *sort assignment* μ for X is a function mapping the names of the variables in $\bar{X}$ to their sorts. The ordering $\leq$ on S is extended to sort assignments by

$$\mu \leq \mu' \Leftrightarrow \forall x \in \bar{X}, \mu(x) \leq \mu'(x).$$

We say then that μ' *specializes* to μ, via the substitution $\rho : (x : \mu'(x)) \leftarrow (x : \mu(x))$ called a *specialization* of $X = (\bar{X}, \mu')$ into $\rho(X) = (\bar{X}, \mu)$.

The notion of specialization can then be extended to axioms and rewrite rules. A specialization of an axiom $(\forall X, l = r)$ is another axiom $(\forall \rho(X), \rho(l) = \rho(r))$ where ρ is a specialization of X. A specialization of a rule $(\forall X, l \to r)$ is a rule $(\forall \rho(X), \rho(l) \to \rho(r))$ where ρ is a specialization of X.

If the set of sorts is finite, or if each sort has only a finite number of sorts below it, then a finite sorted set of variables has a finite number of specializations. This allows deciding the sort-decreasingness property [13, 6].

A rule $(\forall X, l \to r)$ is *sort-decreasing* if and only if for any specialization ρ of X, the least sort of $\rho(l)$ is greater or equal than the least sort of $\rho(r)$.

Let $\mathcal{S}$ be a specification. Then, the set of membership constraints for a conditional equation $t = t'$ `if` c is defined as

$$\begin{aligned} CMC(t = t' \texttt{ if } c) \quad = \{\, t'\theta : LS(t\theta) \texttt{ if } c\theta \mid \quad & \theta \text{ is a specialization of } vars(t) \\ & \text{and } LS(t'\theta \downarrow_{\mathcal{S}}) \not\leq LS(t\theta) \,\} \end{aligned}$$

A membership constraint $t : s$ `if` c is more general than another membership constraint $t' : s'$ `if` c' if there exists a substitution σ such that $t\sigma = t'$, $s \leq s'$ and $c\sigma = c'$.

Let $CMC(\mathcal{S})$ be the set of most general membership constraints of each of the equations in the specification $\mathcal{S}$.

Then, given a specification $\mathcal{S}$, the tool gives

$$checking(\mathcal{S}) = \langle\, CCP(\mathcal{S}),\ CMC(\mathcal{S}) \,\rangle.$$

Absence of critical pairs and of membership constraints in such an output is a sufficient condition for a terminating specification $\mathcal{S}$ to be *Church-Rosser* in the precise sense that:

- it is confluent, and
- the least sort of the canonical form of a term—assuming preregularity of the signature—is smaller or equal than the least sort of any of its equivalent terms.

However, for specifications with an initial algebra semantics we need the property to hold only for ground terms, that is, we only need to check that $\mathcal{S}$ is *ground Church-Rosser*. In practice it is perfectly possible to obtain nonempty sets of critical pairs or of membership constraints for specifications that are in fact ground Church-Rosser. In other cases, it may in fact be a matter of some error in the user specification that the tool uncovers. In any case, the user has complete control on how to modify his specification using the proof obligations in the output of the tool as a guide. In fact, several possibilities exist. He can prove inductively that the critical pairs are ground-joinable, and that the membership constraints are ground-rewritable to a term with the required sort; however, at present the methods available for such proofs are quite limited. He can instead modify the specification with the purpose of either correcting a found mistake, or modifying the already correct specification into a variant that the tool will hopefully certify Church-Rosser when resubmitted.

5.2 Auxiliary Functions

The tool has been implemented using the reflective capabilities of Maude. We use the definitions in the built-in module META-LEVEL. We start by introducing the variables of sorts defined there that we will use.

```
vars B B' : Bool .
vars N M : MachineInt .
vars QI QI' F G X Y S1 S2 S S' S'' S''' L L1 L2 : Qid .
vars QIL SL SL' : QidList .
vars QIS SS SS' : QidSet .
var SD : SortDecl .
vars T T' T'' T''' T1 T1' T1'' T2 T2' T2'' : Term .
vars TL TL' TL'' TL''' : TermList .
vars Subst Subst' Subst'' Subst''' : Substitution .
var IL : ImportList .
var SSDS : SubsortDeclSet .
var ODL : OpDeclList .
vars VDS VDS' VDS'' : VarDeclSet .
vars VD VD' : VarDecl .
var MAS MAS' : MembAxSet .
var MA MA' : MembAx .
vars ES ES' ES'' ES''' : EquationSet .
vars E E' : Equation .
vars Th Th' : Module .
vars A A' : Attr .
var AS AS' : AttrSet .
```

The input to the tool is given by the specification module we want to check that, as already explained cannot import any submodules or contain any built-in sorts or functions. At the present state of development of the Maude system, this module has to be given in its meta-representation. As we have already said, only the order-sorted conditional case with commutativity axioms is considered. The syntax for the conditional equations is as presented in Section 4.3.

The output is given as a 3-tuple of sort `CheckingSolution`, consisting of a set of critical pairs, a set of membership axioms (corresponding to sort-decreasingness proof obligations) and the set of variables appearing in them. We declare sorts for critical pairs (`CritPair`) and for sets of critical pairs (`CritPairSet`) and constructors for them. The constructor for a critical pair (`cp`) has two arguments and the constructor for the conditional critical pair (`ccp`) has four. The two last arguments in a conditional critical pair correspond to the condition, like for conditional equations. The checking for confluence and for sort-decreasingness is carried out by two different functions, `confluenceCheck` and `sortDecrCheck`. The modules passed to these functions are previously *prepared* in order to allow conditional equations and the variables needed in the process are added. The declarations are then given as follows:

```
sorts CritPair CritPairSet CheckingSolution .
subsort CritPair < CritPairSet .

vars CP CP' : CritPair .
var CPS CPS' : CritPairSet .
```

```
op cp : Term Term -> CritPair .
op ccp : Term Term Term Term -> CritPair .
op emptyCritPairSet : -> CritPairSet .
op critPairSet : CritPairSet CritPairSet -> CritPairSet
        [assoc comm id: emptyCritPairSet] .

op checkingSolution :
        VarDeclSet CritPairSet MembAxSet -> CheckingSolution .

op checking : Module -> CheckingSolution .

op confluenceCheck : Module Module -> CritPairSet .
op sortDecrCheck : Module -> MembAxSet .

eq checking(Th)
  = checkingSolution(
        eliminateDuplicateVarDecl(
            varDeclSet(
               varDeclSetInCritPairs(
                  prepareModule(Th),
                  confluenceCheck(prepareModule(Th),
                      addAnd(prepareModule(Th)))),
               varDeclSetInMembAxs(
                  prepareModule(Th),
                  sortDecrCheck(
                      addAnd(prepareModule(Th)))))),
        confluenceCheck(prepareModule(Th),
            addAnd(prepareModule(Th))),
        sortDecrCheck(
            addAnd(prepareModule(Th)))) .
```

As we have seen in Section 2, at the meta-level a user-defined sort has to be a quoted identifier, that is, has to be of sort `Qid`. However, the module `META-LEVEL` has also a sort `Sort` defined to be a supersort of `Qid`. The function `leastSort` can return a sort not defined by the user, for example one of the "error sorts" added by the system to complete the connected components. This makes also necessary the definition of the functions `sortLeq`, `lesserSorts` and `glbSorts` in the module `META-LEVEL` on sort `Sort`.

To use appropriately these functions we need to add the following new constructors for membership axioms:

```
op mb : Term Sort -> MembAx .
op cmb : Term Sort Term Term -> MembAx .
```

We will also need the following variables of sort `Sort`:

```
vars So So1 So2 : Sort .
```

Given a module M with subsort relation $\leq_M$, and sorts $s, s' \in S$, where S is the set of sorts in M, `sortLeq(`M,s,s'`)` if and only if $s \leq_M s'$. We will use the extension of the ordering on S to a partial ordering on strings of equal length in S^* by $s_1 \ldots s_n \leq_M s'_1 \ldots s'_n$ if and only if $s_i \leq_M s'_i$ for $1 \leq i \leq n$.

Note that although the declaration of function `sortLeq` is

```
op sortLeq :  Module Sort Sort -> Bool .
```

we only need to have this extension for lists of sorts defined by the user, that is, on lists of sort `QidList`.

```
op qidListLeq : Module QidList QidList -> Bool .
eq qidListLeq(Th, qidList(S, SL), qidList(S', SL'))
   = andThen(sortLeq(Th, S, S'), qidListLeq(Th, SL, SL')) .
eq qidListLeq(Th, S, S')
   = sortLeq(Th, S, S') .
eq qidListLeq(Th, nilQidList, nilQidList)
   = true .
```

5.2.1 Module Transformation

The function `prepareModule` adds to the module the variables needed in the checking. For each variable V of sort S in the original module a variable V`@'`—used in the generation of critical pairs—and variables V`@'`S' and V`@'@'`S' for each subsort S' of S—used in the sort-decreasingness checking—are created.

```
op prepareModule : Module -> Module .
op prepareVars : Module VarDeclSet -> VarDeclSet .
op prepareVarsSubsorts : Qid QidSet -> VarDeclSet .

eq prepareModule(fmod(QI, IL, SD, SSDS, ODL, VDS, MAS, ES))
  = fmod(QI, IL, SD, SSDS, ODL,
         prepareVars(fmod(QI, IL, SD, SSDS, ODL, VDS, MAS, ES), VDS),
         MAS, ES) .

eq prepareVars(Th, varDeclSet(varDecl(X, S), VDS))
  = varDeclSet(
       varDecl(X, S), varDecl(conc(X, '@'), S),
       prepareVarsSubsorts(X, lesserSorts(Th, S)),
       prepareVars(Th, VDS)) .
eq prepareVars(Th, emptyVarDeclSet)
  = emptyVarDeclSet .

eq prepareVarsSubsorts(X, qidSet(S, SS))
  = varDeclSet(varDecl(conc(X, conc('@', S)), S),
               varDecl(conc(conc(X, '@'@'), S), S),
               prepareVarsSubsorts(X, SS)) .
eq prepareVarsSubsorts(X, emptyQidSet)
  = emptyVarDeclSet .
```

The function `addAnd` adds the definitions necessary for the conjunction used in the conditions of the equations in order to be possible to use it when trying to reduce some term.

```
op addAnd : Module -> Module .
eq addAnd(fmod(QI, IL, SD, SSDS, ODL, VDS, MAS, ES))
  = fmod(QI, IL, SD, SSDS,
         opDeclList(opDecl('and, qidList('Bool, 'Bool), 'Bool,
                           attrSet(comm, assoc)), ODL),
         varDeclSet(varDecl('B@'Bool, 'Bool), VDS), MAS,
         equationSet(eq('and['false, 'B@'Bool], 'false),
                     eq('and['true, 'B@'Bool], 'B@'Bool), ES)) .
```

5.2.2 Boolean Operations

As already mentioned, in the built-in module `META-LEVEL` we have a sort `Bool` and constants `true` and `false`, but we need to define some other operations.

```
op and : Bool Bool -> Bool [assoc comm] .
op andThen : Bool Bool -> Bool [strat (1 0 2 0)] .
op or : Bool Bool -> Bool [assoc comm] .
op orElse : Bool Bool -> Bool [strat (1 0 2 0)] .
op not : Bool -> Bool .

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

eq or(true, B) = true .
eq or(false, B) = B .
eq orElse(true, B) = true .
eq orElse(false, B) = B .

eq not(true) = false .
eq not(false) = true .
```

5.2.3 Some Infrastructure for the Order-Sorted Unification Algorithm

The equations of sort `Equation` are oriented. For the implementation of the order-sorted unification algorithm that will be presented in Section 5.2.6 we will need unoriented equations. Thus, we define sorts `CommEq` and `CommEqSet` with the following declarations.

```
sorts CommEq CommEqSet .

var CES : CommEqSet .
var CE : CommEq .

subsort CommEq < CommEqSet .

op commEq : Term Term -> CommEq [comm] .
op emptyCommEqSet : -> CommEqSet .
```

```
op commEqSet : CommEqSet CommEqSet -> CommEqSet
        [assoc comm id: emptyCommEqSet] .
```

The solution for a unification problem will be given as a set of substitutions, and so, we need the following declarations:

```
sorts SubstitutionSet .
subsort Substitution < SubstitutionSet .

vars SubstS SubstS' : SubstitutionSet .

op emptySubstitutionSet : -> SubstitutionSet .
op substitutionSet :
        SubstitutionSet SubstitutionSet -> SubstitutionSet
        [assoc comm id: emptySubstitutionSet] .
```

We give below declarations for sets of lists of identifiers. These declarations will be used to keep sets of arities of operations, that is, lists of user-defined sorts.

```
sorts QidListSet .

vars SLS SLS' : QidListSet .

subsort QidList < QidListSet .

op emptyQidListSet : -> QidListSet .
op qidListSet : QidListSet QidListSet -> QidListSet
        [assoc comm id: emptyQidListSet] .
```

The function `occurs?` takes an identifier corresponding to a variable and a term and checks if the variable occurs in the term. It is declared on lists of terms just to be defined recursively on each of the arguments of the given term.

```
op occurs? : Qid TermList -> Bool .
eq occurs?(X, F)
   = X == F .
eq occurs?(X, F[TL])
   = occurs?(X, TL) .
eq occurs?(X, (T, TL))
   = orElse(occurs?(X, T), occurs?(X, TL)) .
```

In the equations corresponding to the *merging* inference rule we will need an ordering on equations. Let $|t|$ denote the *size* of the term t as the total number of its function symbols. We define a well-founded ordering $\succ$ on equations as follows: $u = v \succ s = t$ if $max(|u|, |v|) > max(|s|, |t|)$, or else $max(|u|, |v|) = max(|s|, |t|)$ and $max(|u|, |v|) - min(|u|, |v|) > max(|s|, |t|) - min(|s|, |t|)$.

```
op greaterCommEq : Module CommEq CommEq -> Bool .
op size : Module TermList -> MachineInt .
op max : MachineInt MachineInt -> MachineInt .
```

```
op min : MachineInt MachineInt -> MachineInt .

eq greaterCommEq(Th, commEq(T, T'), commEq(T1, T1'))
  = orElse(
       max(size(Th, T), size(Th, T'))
           > max(size(Th, T1), size(Th, T1')),
       andThen(max(size(Th, T), size(Th, T'))
                 == max(size(Th, T1), size(Th, T1')),
               (max(size(Th, T), size(Th, T'))
                   - min(size(Th, T), size(Th, T')))
                  > (max(size(Th, T1), size(Th, T1'))
                        - min(size(Th, T1), size(Th, T1'))))) .

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

5.2.4 Manipulation of Module Information

We will start this section introducing some basic functions to calculate the length of different lists (`termListLength` and `qidListLength`), to check if an element is in a given set (`inQidSet?`, `inSubstitutionSet?`, `inVarDeclSet?` and `inAttrSet?`), to check if there is an assignment for a particular variable (`substitutionFor?`), and to reverse a list of identifiers (`reverseQidList`).

```
op termListLength : TermList -> MachineInt .
eq termListLength((T, TL))
  = 1 + termListLength(TL) .
eq termListLength(T)
  = 1 .

op qidListLength : QidList -> MachineInt .
eq qidListLength(qidList(QI, QIL))
  = 1 + qidListLength(QIL) .
eq qidListLength(nilQidList)
  = 0 .

op inQidSet? : Qid QidSet -> Bool .
eq inQidSet?(QI, qidSet(QI', QIS))
  = orElse(QI == QI', inQidSet?(QI, QIS)) .
eq inQidSet?(QI, emptyQidSet)
  = false .
```

```
op inSubstitutionSet? : Substitution SubstitutionSet  -> Bool .
eq inSubstitutionSet?(Subst, substitutionSet(Subst', SubstS))
  = orElse(Subst == Subst', inSubstitutionSet?(Subst, SubstS)) .
eq inSubstitutionSet?(Subst, emptySubstitutionSet)
  = false .

op inVarDeclSet? : VarDecl VarDeclSet  -> Bool .
eq inVarDeclSet?(VD, varDeclSet(VD', VDS))
  = orElse(VD == VD', inVarDeclSet?(VD, VDS)) .
eq inVarDeclSet?(VD, emptyVarDeclSet)
  = false .

op inAttrSet? : Attr AttrSet -> Bool .
eq inAttrSet?(A, attrSet(A', AS))
  = orElse(A == A', inAttrSet?(A, AS)) .
eq inAttrSet?(A, emptyAttrSet)
  = false .

op substitutionFor? : Substitution Qid  -> Bool .
eq substitutionFor?(((X <- T); Subst), Y)
  = orElse(X == Y, substitutionFor?(Subst, Y)) .
eq substitutionFor?(emptySubstitution, X)
  = false .

op reverseQidList : QidList -> QidList .
eq reverseQidList(qidList(QI, QIL))
  = qidList(reverseQidList(QIL), QI) .
eq reverseQidList(nilQidList)
  = nilQidList .
```

We will define now some more basic operations: `commOp?` to check whether an operator has been declared commutative in the specification, `isVar?` to check if an identifier corresponds to a variable, functions `sorts`, `subsortRelation`, `operations`, `variables` and `equations` to access the information in the module, and functions `variableSort`, `constantSort` and `functionSort` to get the sort of a variable, a constant and an operation declaration, respectively.

```
op commOp? : Module Qid -> Bool .
eq commOp?(
     fmod(QI, IL, SD, SSDS,
          opDeclList(opDecl(F, SL, S, AS), ODL), VDS, MAS, ES), G)
  = if andThen(F == G, qidListLength(SL) == 2)
    then inAttrSet?(comm, AS)
    else commOp?(fmod(QI, IL, SD, SSDS, ODL, VDS, MAS, ES), G)
    fi .
eq commOp?(
       fmod(QI, IL, SD, SSDS, nilOpDeclList, VDS, MAS, ES), G)
  = false .
```

```
op isVar? : Module Term -> Bool .
op isVarInVarDeclSet? : VarDeclSet Term -> Bool .
eq isVar?(fmod(QI, IL, SD, SSDS, ODL, VDS, MAS, ES), T)
  = isVarInVarDeclSet?(VDS, T) .
eq isVarInVarDeclSet?(emptyVarDeclSet, T)
  = false .
eq isVarInVarDeclSet?(varDeclSet(varDecl(X, S), VDS), F)
  = orElse(X == F, isVarInVarDeclSet?(VDS, F) ) .
eq isVarInVarDeclSet?(varDeclSet(varDecl(X, S), VDS), F[TL])
  = false .

op sorts : Module -> SortDecl .
op subsortRelation : Module -> ESubsortDeclSet .
op operations : Module -> OpDeclList .
op variables : Module -> VarDeclSet .
op equations : Module -> EquationSet .

eq sorts(fmod(QI, IL, SD, SSDS, ODL, VDS, MAS, ES))
  = SD .
eq subsortRelation(fmod(QI, IL, SD, SSDS, ODL, VDS, MAS, ES))
  = SSDS .
eq operations(fmod(QI, IL, SD, SSDS, ODL, VDS, MAS, ES))
  = ODL .
eq variables(fmod(QI, IL, SD, SSDS, ODL, VDS, MAS, ES))
  = VDS .
eq equations(fmod(QI, IL, SD, SSDS, ODL, VDS, MAS, ES))
  = ES .

op variableSort : Module Qid -> Qid .
op constantSort : Module Qid -> Qid .

eq variableSort(
      fmod(QI, IL, SD, SSDS, ODL, varDeclSet(varDecl(X, S), VDS),
           MAS, ES), X)
  = S .
eq constantSort(
     fmod(QI, IL, SD, SSDS,
          opDeclList(opDecl(F, SL, S, AS), ODL), VDS, MAS, ES), G)
  = if or(G == 'true, G == 'false)
    then 'Bool
    else if and(F == G, SL == nilQidList)
         then S
         else constantSort(
                 fmod(QI, IL, SD, SSDS, ODL, VDS, MAS, ES), G)
         fi
    fi .
```

5.2.5 General Operations on Terms

In our specification, `pos` is the constructor symbol for positions, and `emptyPos` for the empty sequence of positions.

We include in this section the definitions of operations to manipulate terms: definitions for positions as given in Section 5.1, to get the subterm of a given term at a given position, to replace the subterm of a term at a given position by another term, to get all the non-variable positions in a term and to apply a substitution.

```
sorts Position PositionSet .

var P : Position .
var PS : PositionSet .

subsort MachineInt < Position < PositionSet .

op emptyPos : -> Position .
op pos : Position Position -> Position [assoc] .
op nullPos : -> Position .
op emptyPosSet : -> PositionSet .
op posSet : PositionSet PositionSet -> PositionSet
        [assoc comm id: emptyPosSet] .
```

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

```
op getSubterm : Term Position -> Term .
op getSubterm1 : TermList MachineInt -> Term .

eq getSubterm(T, emptyPos)
  = T .
eq getSubterm(F, N)
  = error* .
eq getSubterm(F[TL], N)
  = getSubterm1(TL, N) .
eq getSubterm(F, pos(N, P))
  = error* .
eq getSubterm(F[TL], pos(N, P))
  = getSubterm(getSubterm1(TL, N), P) .

eq getSubterm1((T, TL), N)
  = if N == 1
    then T
    else getSubterm1(TL, (N - 1))
    fi .
eq getSubterm1(T, N)
  = if N == 1
    then T
    else error*
    fi .
```

The function `replace` takes terms t and t' and a position p as input and returns the term $t[t']_p$.

```
op replace : Term Term Position -> Term .
op replace1 : TermList Term Position -> Term .

eq replace(T, T', emptyPos)
  = T' .
eq replace(F, T', N)
  = error* .
eq replace(F[TL], T', N)
  = F[replace1(TL, T', N)] .

eq replace1((T, TL), T', N)
  = if N == 1
    then (T', TL)
    else (T, replace1(TL, T', (N - 1)))
    fi .
eq replace1(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[replace1(TL, T', pos(N, P))] .

eq replace1((T, TL), T', pos(N, P))
  = if N == 1
    then (replace(T, T', P), TL)
    else (T, replace1(TL, T', pos((N - 1), P)))
    fi .
eq replace1(T, T', pos(N, P))
  = if N == 1
    then replace(T, T', P)
    else error*
    fi .
```

The function `allNonVarPos` takes a module M and a term t and returns the set $\mathcal{P}_M(t)$ of nonvariable positions of t: $\{p \in \mathcal{P} \mid t|_p \notin vars(M)\}$.

```
op allNonVarPos : Module Term -> PositionSet .
op allNonVarPos1 :
        Module Term Position PositionSet -> PositionSet .
op nextPosition : Term Position -> Position .
op nextPositionUp : Term Position -> Position .

eq allNonVarPos(Th, T)
```

```
  = posSet(emptyPos,
           allNonVarPos1(Th, T, emptyPos, emptyPosSet)) .

eq allNonVarPos1(Th, T, P, PS)
  = if nextPosition(T, P) == nullPos
    then PS
    else if isVar?(Th, getSubterm(T, nextPosition(T, P)))
         then allNonVarPos1(Th, T, nextPosition(T, P), PS)
         else allNonVarPos1(Th, 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 .
```

The function `substitute` takes a term t and a substitution σ and returns $t\sigma$. This function is extended to sets of (commutative and non commutative) equations, sets of membership axioms and to substitutions with functions `substituteCommEqSet`, `substituteEquationSet`, `substMembAxSet` and `substituteSubstitution`.

```
op substitute : TermList Substitution -> TermList .
op substituteCommEqSet : CommEqSet Substitution -> CommEqSet .
op substituteEquationSet :
        EquationSet Substitution -> EquationSet .
op substituteSubstitution :
        Substitution Substitution  -> Substitution .
op substMembAxSet : Module VarDeclSet -> Substitution .

eq substitute(T, emptySubstitution)
  = T .
```

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

eq substituteCommEqSet(commEqSet(commEq(T, T'), CES), Subst)
  = commEqSet(commEq(substitute(T, Subst), substitute(T', Subst)),
              substituteCommEqSet(CES, Subst)) .
eq substituteCommEqSet(emptyCommEqSet, Subst)
  = emptyCommEqSet .

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

eq substituteEquationSet(equationSet(eq(T, T'), ES), Subst)
  = equationSet(eq(substitute(T, Subst), substitute(T', Subst)),
                   substituteEquationSet(ES, Subst)) .
eq substituteEquationSet(emptyEquationSet, Subst)
  = emptyEquationSet .

eq substMembAxSet(Th, varDeclSet(varDecl(X, S), VDS))
  = if variableSort(Th, X) == S
    then substMembAxSet(Th, VDS)
    else ((X <- conc(X, conc('@', S))) ; substMembAxSet(Th, VDS))
    fi .
eq substMembAxSet(Th, emptyVarDeclSet)
  = emptySubstitution .

op varDeclSetInTerm : Module TermList -> VarDeclSet .
ceq varDeclSetInTerm(Th, F)
  = emptyVarDeclSet
    if not(isVar?(Th, F)) .
eq varDeclSetInTerm(
      fmod(QI, IL, SD, SSDS, ODL,
           varDeclSet(varDecl(X, S), VDS), MAS, ES), X)
  = varDecl(X, S) .
eq varDeclSetInTerm(Th, F[TL])
  = varDeclSetInTerm(Th, TL) .
eq varDeclSetInTerm(Th, (T, TL))
  = varDeclSet(varDeclSetInTerm(Th, T),
               varDeclSetInTerm(Th, TL)) .
```

We now define functions to collect the multiset of variables in a set of critical pairs (`varDeclSetInCritPairs`) and from a set of membership axioms (`varDeclSetInMembAxs`). We also define a function to eliminate duplicated elements in a multiset of variables (eliminateDuplicateVarDecl).

```
op eliminateDuplicateVarDecl : VarDeclSet -> VarDeclSet .
eq eliminateDuplicateVarDecl(varDeclSet(VD, VDS))
  = if inVarDeclSet?(VD, VDS)
    then eliminateDuplicateVarDecl(VDS)
    else varDeclSet(VD, eliminateDuplicateVarDecl(VDS))
    fi .
eq eliminateDuplicateVarDecl(emptyVarDeclSet)
  = emptyVarDeclSet .

op varDeclSetInCritPairs : Module CritPairSet -> VarDeclSet .
eq varDeclSetInCritPairs(Th, critPairSet(cp(T, T'), CPS))
  = varDeclSet(varDeclSetInTerm(Th, T),
               varDeclSetInTerm(Th, T'),
               varDeclSetInCritPairs(Th, CPS)) .
eq varDeclSetInCritPairs(
      Th, critPairSet(ccp(T, T', T'', 'true), CPS))
  = varDeclSet(varDeclSetInTerm(Th, T),
               varDeclSetInTerm(Th, T'),
               varDeclSetInTerm(Th, T''),
               varDeclSetInCritPairs(Th, CPS)) .
eq varDeclSetInCritPairs(Th, emptyCritPairSet)
  = emptyVarDeclSet .

op varDeclSetInMembAxs : Module MembAxSet -> VarDeclSet .
eq varDeclSetInMembAxs(Th, membAxSet(mb(T, So), MAS))
  = varDeclSet(varDeclSetInTerm(Th, T),
               varDeclSetInMembAxs(Th, MAS)) .
eq varDeclSetInMembAxs(Th, membAxSet(cmb(T, So, T', 'true), MAS))
  = varDeclSet(varDeclSetInTerm(Th, T),
               varDeclSetInTerm(Th, T'),
               varDeclSetInMembAxs(Th, MAS)) .
eq varDeclSetInMembAxs(Th, emptyMembAxSet)
  = emptyVarDeclSet .
```

Finally, we will define the function `rename`, which takes a term and renames all the variables appearing in it. The renaming is done by searching variables and replacing them for another variable with its name concatenated with the string `@'`, that is, each variable V is substituted by V`@'`.

```
op rename : Module TermList -> TermList .

eq rename(Th, F[TL])
  = F[rename(Th, TL)] .
eq rename(Th, (T, TL))
  = (rename(Th, T), rename(Th, TL)) .
eq rename(Th, X)
  = if isVar?(Th, X)
    then conc(X, '@')
```

```
else X
fi .
```

5.2.6 Order-Sorted Unification With Commutativity Axioms

In this section we sketch the unification algorithm implemented. Following [12], we unify using the sort information as soon as possible in order to quickly discard failure. Then, we complete the simplification process and push the constraints of the sorts on the solutions that have been found. A similar point of view holds for order-sorted matching.

Any order-sorted unification algorithm for solving equations (pairs of the form $t =^?_c t'$) can be described as the result of a simplification step followed by a solving step. The simplification step is a sequence of decomposition, merging and mutation steps, transforming the initial unification problem into an equivalent disjunction of systems of fully decomposed equations of the form $x =^?_c t$, where x is a variable appearing only once in the system. The solving step consist in finding the finite set of solutions for the simplified system.

Thus, a unification problem is a set of equations, also called a system, denoted $(e_1, \ldots, e_n)$ or a set of systems, also called a disjunction of systems and written $S_1 \vee \ldots \vee S_n$. The set of variables occurring in a unification problem U is denoted $vars(U)$.

Given a specification $\mathcal{S} = (\Sigma, E)$, a substitution σ is $\mathcal{S}$-solution of the equation $t =^?_E t'$ if and only if $\sigma(t) =_E \sigma(t')$. The set of $\mathcal{S}$-solutions of an equation $t =^?_E t'$ is denoted $U(t, t', \mathcal{S})$.

A set of substitutions Φ is a complete set of $\mathcal{S}$-solutions of the equation $t =^?_E t'$ away from the set of variables W such that $vars(t) \cup vars(t') \subseteq W$ if and only if

- $\forall \sigma \in \Phi, D(\sigma) \subseteq vars(t) \cup vars(t')$ and $I(\sigma) \cap W = \emptyset$;
- $\forall \sigma \in \Phi, \sigma \in U(t, t', \mathcal{S})$; and
- $\forall \sigma \in U(t, t', \mathcal{S}), \exists \alpha \in \Phi$ such that $\alpha \lesssim_E \sigma[vars(t) \cup vars(t')]$.

Note that as mentioned in Section 5.1 we do not compute the minimal set of unifiers. Note also that it would be easy to deal with ad-hoc overloading following this set of inference rules, but because of the current state of the implementation of the Maude system this is not considered. With subsort overloading we assume that an operator symbol has always the same attributes.

Let the following set of inference rules operating on 3-tuples $\langle V;\ E;\ \sigma\rangle$ with V a set of variables, E a set of equations and σ a substitution and on 4-tuples $[V;\ C;\ \sigma;\ \theta]$ with V a set of variables, C a set of membership constraints and σ and θ substitutions. The rules operating on the first kind of tuples correspond to the first stage of the process, which is quite similar to syntactic unification. The main differences are in the rules **Check** and **Eliminate** in which the sort information is used to try to quickly discard failure. In the second stage the constraints on the solutions are checked.

Deletion of Trivial Equations

$$\frac{\langle\ V;\ \{t =^?_c t,\ E\};\ \sigma\ \rangle}{\langle\ V;\ E;\ \sigma\ \rangle}$$

Decomposition

$$\frac{\langle\ V;\ \{f(t_1,\ \ldots,\ t_n) =^?_c f(t'_1,\ \ldots,\ t'_n),\ E\};\ \sigma\ \rangle}{\langle\ V;\ \{t_1 =^?_c t'_1,\ \ldots,\ t_n =^?_c t'_n,\ E\};\ \sigma\ \rangle}$$

if $n \neq 2$ or f non commutative

$$\frac{\langle\ V;\ \{f(t_1,t_2) =^?_c f(t'_1,t'_2),\ E\};\ \sigma\ \rangle}{\langle\ V;\ \{t_1 =^?_c t'_1,\ t_2 =^?_c t'_2,\ E\};\ \sigma\ \rangle \vee \langle\ V;\ \{t_1 =^?_c t'_2,\ t_2 =^?_c t'_1,\ E\};\ \sigma\ \rangle}$$

if f commutative

Clash of Symbols

$$\frac{\langle\ V;\ \{f(t_1,\ \ldots,\ t_n) =^?_c g(t'_1,\ \ldots,\ t'_m),\ E\};\ \sigma\ \rangle}{\mathit{failure}}$$

if f, g operators and ($n \neq m$ or $f \neq g$)

Merging

$$\frac{\langle\ \{x : s,\ V\};\ \{x =^?_c t,\ x =^?_c t',\ E\};\ \sigma\ \rangle}{\langle\ \{x : s,\ V\};\ \{x =^?_c t,\ t =^?_c t',\ E\};\ \sigma\ \rangle}$$

if $x = t \succ t = t'$

Check

$$\frac{\langle\ \{x : s,\ V\};\ \{x =^?_c t,\ E\};\ \sigma\ \rangle}{\mathit{failure}}$$

if $x \neq t$ and (x occurs in t or $s \cap LS(t) = \emptyset$)

Eliminate

$$\frac{\langle\ \{x : s,\ V\};\ \{x =^?_c t,\ E\};\ \sigma\ \rangle}{\langle\ \{x : s,\ V\};\ E\theta;\ \{\sigma\theta, \theta\}\ \rangle}$$

with $\theta = \{x \leftarrow t\}$

if x does not occur in t and $s \cap LS(t) \neq \emptyset$)

Transition

$$\frac{\langle\ V;\ \emptyset;\ \sigma\ \rangle}{[\ V;\ \emptyset;\ \sigma;\ \sigma\ ]}$$

Solving $(x \leftarrow t)$

$$\frac{[\ \{x : s,\ V\};\ C;\ \{x \leftarrow t,\ \sigma\};\ \theta\ ]}{[\ V;\ \{(t : s),\ C\};\ \sigma;\ \theta\ ]}$$

Solving $(x : s)$

$$\frac{[\ \{x : s,\ V\};\ \{x : s',\ C\};\ \emptyset;\ \theta\ ]}{\bigvee_{s'' \in s \wedge s'} [\ \{x : s'',\ V\};\ C;\ \emptyset;\ \theta\ ]}$$

Solving $(f(t_1, \ldots, t_n) : s)$

$$\frac{[\ V;\ \{f(t_1, \ldots, t_n) : s,\ C\};\ \emptyset;\ \theta\ ]}{\bigvee_{\substack{f : s_1 \ldots s_n \to s' \\ s' \leq s \\ s_1 \ldots s_n\, maximal}} [\ V;\ \{t_1 : s_1,\ \ldots,\ t_n : s_n,\ C\};\ \emptyset;\ \theta\ ]}$$

These inference rules are mapped into Maude specifications almost without modification. But in order to do that we need some previous declarations. Because we need to have access to the information in the module we define the equations on elements of sort `UnifPair`, which has a constructor for pairs formed by a module (of sort `Module`) and a disjunction of tuples. The sort `Disjunction` is then defined as a set of unification tuples.

```
sorts UnifPair UnifTuple Disjunction .

subsort UnifTuple < Disjunction .

var D : Disjunction .
var UT : UnifTuple .

op unify : Module Term Term -> SubstitutionSet .
op getUnifSolution : UnifPair -> SubstitutionSet .

op failure : -> Disjunction .
op disjunction : Disjunction Disjunction -> Disjunction
        [assoc comm id: failure] .
op unifPair : Module Disjunction -> UnifPair .
op unifTuple : VarDeclSet CommEqSet Substitution -> UnifTuple .
op unifTuple2 : VarDeclSet MembAxSet Substitution Substitution
        -> UnifTuple .
```

Given the previous rules and the way in which the functions are defined, each of the tuples in the disjunction in `unifPair` terminates when the set of constraints for the variables is empty, that is, in a state of the form `unifTuple2(`V`,`$\emptyset$`,`σ`)` for some variables V and some substitution σ. The function `getUnifSolution` makes use of this information to extract the solutions and generates the set of unifiers from them.

```
eq unify(Th, T, T')
  = getUnifSolution(
       unifPair(Th,
                unifTuple(variables(Th), commEq(T, T'),
                          emptySubstitution))) .

eq getUnifSolution(
      unifPair(Th,
         disjunction(unifTuple2(VDS, emptyMembAxSet,
                                emptySubstitution, Subst), D)))
  = substitutionSet(
        (substituteSubstitution(Subst,
           substMembAxSet(Th, VDS)) ; substMembAxSet(Th, VDS)),
        getUnifSolution(unifPair(Th, D))) .
eq getUnifSolution(unifPair(Th, failure))
  = emptySubstitutionSet .
```

We will introduce the corresponding definitions for each one of the rules. Let us start with the rule **Deletion of Trivial Equations**.

```
eq unifPair(Th,
            disjunction(
              unifTuple(VDS, commEqSet(commEq(T, T), CES), Subst),
              D))
  = unifPair(Th, disjunction(unifTuple(VDS, CES, Subst), D)) .
```

For the **Decomposition** rules we need two auxiliary functions: one to generate the tuples corresponding to the commutative case (`commUnifTupleSet`), and another one to generate a set of equations corresponding to the decomposition of the unification problem into smaller ones for the subterms (`commEqBreak`).

```
op commUnifTupleSet :
        VarDeclSet Qid TermList TermList CommEqSet
        Substitution -> Disjunction .
eq commUnifTupleSet(VDS, F, (T, T'), (T'', T'''), CES, Subst)
 = disjunction(
     unifTuple(VDS,
       commEqSet(commEq(T, T''), commEq(T', T'''), CES), Subst),
     unifTuple(VDS,
       commEqSet(commEq(T, T'''), commEq(T', T''), CES), Subst)) .

op commEqBreak : TermList TermList -> CommEqSet .
eq commEqBreak(T, T')
  = commEq(T, T') .
eq commEqBreak((T, TL), (T', TL'))
  = commEqSet(commEq(T, T'), commEqBreak(TL, TL')) .
```

Then, the two **Decomposition** rules are combined as follows. The case for the **Clash** rule in which non-constant operations with different symbols appear is also included in this equation.

```
eq unifPair(
    Th,
    disjunction(
     unifTuple(VDS, commEqSet(commEq(F[TL], G[TL']), CES), Subst),
     D))
 = if orElse(F =/= G, termListLength(TL) =/= termListLength(TL'))
   then unifPair(Th, D)
   else if andThen(termListLength(TL) == 2, commOp?(Th, F))
        then unifPair(
                Th,
                disjunction(
                   commUnifTupleSet(VDS, F, TL, TL', CES, Subst),
                   D))
        else unifPair(
                Th,
                disjunction(
                  unifTuple(VDS,
                     commEqSet(commEqBreak(TL, TL'), CES), Subst),
                  D))
```

```
      fi
   fi .
```

We need two more equations for the **Clash** rule, corresponding to the cases in which two constants or a constant and a non-constant operator appear.

```
ceq unifPair(Th,
      disjunction(
          unifTuple(VDS, commEqSet(commEq(F, G), CES), Subst), D))
  = unifPair(Th, D)
    if andThen(F =/= G,
               not(orElse(isVar?(Th, F), isVar?(Th, G)))) .

ceq unifPair(Th,
     disjunction(
      unifTuple(VDS, commEqSet(commEq(G, F[TL]), CES), Subst), D))
  = unifPair(Th, D)
    if not(isVar?(Th, G)) .
```

With the order on equations defined in Section 5.2.3 the equation for the **Merging** rule is as follows.

```
ceq unifPair(Th,
       disjunction(
           unifTuple(varDeclSet(varDecl(X, S), VDS),
                     commEqSet(commEq(X, T), commEq(X, T'), CES),
                     Subst), D))
  = unifPair(Th,
       disjunction(
           unifTuple(varDeclSet(varDecl(X, S), VDS),
                     commEqSet(commEq(X, T), commEq(T, T'), CES),
                     Subst), D))
    if greaterCommEq(Th, commEq(X, T'), commEq(T, T')) .
```

The **Check** and **Eliminate** rules are combined as follows.

```
ceq unifPair(Th,
       disjunction(
           unifTuple(varDeclSet(varDecl(X, S), VDS),
                     commEqSet(commEq(X, T), CES), Subst), D))
  = if occurs?(X, T)
    then unifPair(Th, D)
    else if glbSorts(Th, leastSort(Th, T), S) == emptyQidSet
         then unifPair(Th, D)
         else unifPair(Th,
                 disjunction(
                    unifTuple(varDeclSet(varDecl(X, S), VDS),
                              substituteCommEqSet(CES, (X <- T)),
                              (substituteSubstitution(
                                 Subst, (X <- T)); (X <- T))), D))
         fi
```

```
    fi
    if X =/= T .
```

The equations for the **Transition** and **Solving** $(\mathbf{x} \leftarrow \mathbf{t})$ rules are written as follows.

```
eq unifPair(Th,
         disjunction(unifTuple(VDS, emptyCommEqSet, Subst), D))
  = unifPair(Th,
      disjunction(
         unifTuple2(VDS, emptyMembAxSet, Subst, Subst), D)) .

eq unifPair(Th,
      disjunction(
         unifTuple2(varDeclSet(varDecl(X, S), VDS), MAS,
                               ((X <- T) ; Subst), Subst'), D))
  = unifPair(Th,
      disjunction(
         unifTuple2(VDS, membAxSet(mb(T, S), MAS), Subst, Subst'),
         D)) .
```

For the equations for the **Solving** $(\boldsymbol{x} : \boldsymbol{s})$ and **Solving** $(\boldsymbol{f(t1, ..., tn)} : \boldsymbol{s})$ rules we need some other functions.

In the **Solving** $(\boldsymbol{x} : \boldsymbol{s})$ rule, given a variable x with (possibly constrained) sort s and membership constraint s', we have to create a tuple for each of the maximal sorts in the intersection of the sets of subsorts for each of them. The greatest lower bound is given by the built-in function `glbSorts` introduced in Section 2.

```
op unifTuplesVar : Module Qid Qid Qid UnifTuple -> Disjunction .
op unifTuplesVar1 : QidSet Qid UnifTuple -> Disjunction .

eq unifTuplesVar(Th, X, So, S, UT)
  = unifTuplesVar1(glbSorts(Th, So, S), X, UT) .

eq unifTuplesVar1(qidSet(S, SS),
                X, unifTuple2(VDS, MAS, Subst, Subst'))
  = disjunction(
      unifTuple2(varDeclSet(varDecl(X, S), VDS), MAS, Subst, Subst'),
      unifTuplesVar1(SS, X, unifTuple2(VDS, MAS, Subst, Subst'))) .
eq unifTuplesVar1(emptyQidSet, X, UT)
  = failure .
```

Then, the equation corresponding to the rule can be given as

```
eq unifPair(Th,
      disjunction(unifTuple2(varDeclSet(varDecl(X, S), VDS),
                            membAxSet(mb(X, So), MAS),
                            emptySubstitution, Subst), D))
  = unifPair(Th,
      disjunction(
         unifTuplesVar(Th, X, So, S,
            unifTuple2(VDS, MAS, emptySubstitution, Subst)), D)) .
```

The last rule is a bit more complicated. In it, given a constraint of the form $f(t_1, \ldots, t_n) : s$ we need to generate a tuple for each of the operation declarations with maximal arity among those of sort less or equal than s.

```
op unifTuplesNonVar : Module MembAx UnifTuple -> Disjunction .
op unifTuplesNonVar1 :
        QidListSet TermList UnifTuple -> Disjunction .
op unifTuplesNonVar2 : QidList TermList UnifTuple -> UnifTuple .
op getMaximalArities : Module Qid MachineInt Sort -> QidListSet .
op arityOfFunctions : Module OpDeclList Qid MachineInt Sort -> QidListSet .
op maximalSortListSet : Module QidListSet -> QidListSet .
op maximalSortListSet1 :
        Module QidListSet QidListSet -> QidListSet .
op maximalSortList? : Module QidList QidListSet -> Bool .

eq unifTuplesNonVar(Th, mb(F[TL], So), UT)
  = unifTuplesNonVar1(
       getMaximalArities(Th, F, termListLength(TL), So), TL, UT) .

eq unifTuplesNonVar1(qidListSet(SL, SLS), TL, UT)
  = disjunction(unifTuplesNonVar2(SL, TL, UT),
                unifTuplesNonVar1(SLS, TL, UT)) .
eq unifTuplesNonVar1(emptyQidListSet, TL, UT)
  = failure .

eq unifTuplesNonVar2(
      qidList(S, SL), (T, TL), unifTuple2(VDS, MAS, Subst, Subst'))
  = unifTuplesNonVar2(SL, TL,
      unifTuple2(VDS, membAxSet(mb(T, S), MAS), Subst, Subst')) .
eq unifTuplesNonVar2(S, T, unifTuple2(VDS, MAS, Subst, Subst'))
  = unifTuple2(VDS, membAxSet(mb(T, S), MAS), Subst, Subst') .

eq getMaximalArities(Th, F, N, So)
  = maximalSortListSet(Th, arityOfFunctions(Th, operations(Th), F, N, So)) .

eq arityOfFunctions(Th, opDeclList(opDecl(G, SL, S, AS), ODL), F, N, So)
  = if F == G
    then if qidListLength(SL) == N
         then if sortLeq(Th, S, So)
              then if inAttrSet?(comm, AS)
                   then qidListSet(SL, reverseQidList(SL),
                           arityOfFunctions(Th, ODL, F, N, So))
                   else qidListSet(SL,
                           arityOfFunctions(Th, ODL, F, N, So))
                   fi
              else arityOfFunctions(Th, ODL, F, N, So)
              fi
         else arityOfFunctions(Th, ODL, F, N, So)
         fi
    else arityOfFunctions(Th, ODL, F, N, So)
```

```
    fi .
eq arityOfFunctions(Th, nilOpDeclList, F, N, So)
  = emptyQidListSet .

eq maximalSortListSet(Th, SLS)
  = maximalSortListSet1(Th, SLS, SLS) .

eq maximalSortListSet1(Th, qidListSet(SL, SLS), SLS')
  = if maximalSortList?(Th, SL, SLS')
    then qidListSet(SL, maximalSortListSet1(Th, SLS, SLS'))
    else maximalSortListSet1(Th, SLS, SLS')
    fi .
eq maximalSortListSet1(Th, emptyQidListSet, SLS)
  = emptyQidListSet .

eq maximalSortList?(Th, SL, qidListSet(SL', SLS))
  = if SL == SL'
    then maximalSortList?(Th, SL, SLS)
    else andThen(not(qidListLeq(Th, SL, SL')),
                 maximalSortList?(Th, SL, SLS))
    fi .
eq maximalSortList?(Th, SL, emptyQidListSet)
  = true .
```

Given these functions, the rule can be specified by the following two equations.

```
ceq unifPair(Th,
       disjunction(
          unifTuple2(VDS,
                     membAxSet(mb(F, So), MAS),
                     emptySubstitution, Subst), D))
  = if sortLeq(Th, constantSort(Th, F), So)
    then unifPair(Th,
           disjunction(
              unifTuple2(VDS, MAS, emptySubstitution, Subst), D))
    else unifPair(Th, D)
    fi
    if not(isVar?(Th, F)) .
eq unifPair(Th,
      disjunction(unifTuple2(VDS, membAxSet(mb(F[TL], So), MAS),
                             emptySubstitution, Subst), D))
  = unifPair(Th,
       disjunction(
          unifTuplesNonVar(
             Th, mb(F[TL], So),
             unifTuple2(VDS, MAS, emptySubstitution, Subst)), D)) .
```

5.2.7 Order-Sorted Matching With Commutativity Axioms

The specification for the order-sorted matching is quite similar to the one previously given for unification. The main differences come from the fact that now we have to consider non-commutative equations, to find a match from a term t to another term t' is to find a substitution σ such that $t' =_c t\sigma$. The checks for the correct sorting of an equation in solved form $x =_c^? t$ is reduced to compare the sort of the variable s with the least sort of the term t being assigned. For our tool we do not need the solution of the matching, just to know if there is match or not. As for the unification case we consider the order-sorted matching with the commutative axiom, assuming no ad-hoc overloading and where subsort overloading that the symbols have the same attributes.

The other difference is that now we have to consider a new case for the **Clash** rule, corresponding to the case in which we try to match from a term of the form $f(t_1, \ldots, t_n)$ to a variable.

```
sort MatchTuple .

subsort MatchTuple < Disjunction .

var MT : MatchTuple .

op match? : Module EquationSet -> Bool .
op getMatchSolution : UnifPair -> SubstitutionSet .
op matchTuple :
        VarDeclSet EquationSet Substitution -> MatchTuple .
op eqBreak : TermList TermList -> EquationSet .
op greaterEq : Module Equation Equation -> Bool .
op commMatchTupleSet :
        VarDeclSet Qid TermList TermList EquationSet
        Substitution -> Disjunction .

eq match?(Th, ES)
  = emptySubstitutionSet
       =/= getMatchSolution(
              unifPair(Th, matchTuple(variables(Th), ES,
                       emptySubstitution))) .
```

Deletion of trivial equation.

```
eq unifPair(Th,
      disjunction(
          matchTuple(VDS, equationSet(eq(T, T), ES), Subst), D))
  = unifPair(Th, disjunction(matchTuple(VDS, ES, Subst), D)) .
```

Decomposition / Clash.

```
eq unifPair(Th,
     disjunction(
       matchTuple(VDS, equationSet(eq(F[TL], G[TL']), ES), Subst),
       D))
  = if orElse(F =/= G, termListLength(TL) =/= termListLength(TL'))
    then unifPair(Th, D)
    else if andThen(termListLength(TL) == 2, commOp?(Th, F))
         then unifPair(Th,
                disjunction(
```

```
                 commMatchTupleSet(VDS, F, TL, TL', ES, Subst),
                 D))
         else unifPair(Th,
               disjunction(
                matchTuple(VDS, equationSet(eqBreak(TL, TL'), ES),
                           Subst), D))
         fi
    fi .

eq commMatchTupleSet(VDS, F, (T, T'), (T'', T'''), ES, Subst)
 = disjunction(
      matchTuple(
         VDS, equationSet(eq(T, T''), eq(T', T'''), ES), Subst),
      matchTuple(
         VDS, equationSet(eq(T, T'''), eq(T', T''), ES), Subst)) .
```

Clash of symbols.

```
ceq unifPair(Th,
      disjunction(
        matchTuple(VDS, equationSet(eq(F, G), ES), Subst), D))
  = unifPair(Th, D)
    if andThen(F =/= G, not(isVar?(Th, F))) .

ceq unifPair(Th,
      disjunction(
        matchTuple(VDS, equationSet(eq(G, F[TL]), ES), Subst), D))
  = unifPair(Th, D)
    if not(isVar?(Th, G)) .

eq unifPair(Th,
     disjunction(
       matchTuple(VDS, equationSet(eq(F[TL], G), ES), Subst), D))
  = unifPair(Th, D) .
```

Merging.

```
ceq unifPair(Th,
      disjunction(matchTuple(varDeclSet(varDecl(X, S), VDS),
                             equationSet(eq(X, T), eq(X, T'), ES),
                             Subst), D))
  = unifPair(Th,
      disjunction(matchTuple(varDeclSet(varDecl(X, S), VDS),
                             equationSet(eq(X, T), eq(T, T'), ES),
                             Subst), D))
    if greaterEq(Th, eq(X, T'), eq(T, T')) .
```

Check / Eliminate.

```
ceq unifPair(Th,
      disjunction(
```

```
      matchTuple(varDeclSet(varDecl(X, S), VDS),
                 equationSet(eq(X, T), ES), Subst), D))
  = if occurs?(X, T)
    then unifPair(Th, D)
    else if sortLeq(Th, leastSort(Th, T), S)
         then unifPair(Th,
                disjunction(
                  matchTuple(varDeclSet(varDecl(X, S), VDS),
                             substituteEquationSet(ES, (X <- T)),
                             (substituteSubstitution(
                                 Subst, (X <- T));
                              (X <- T))), D))
         else unifPair(Th, D)
         fi
    fi
    if X =/= T .
```

The auxiliary functions:

```
eq getMatchSolution(
      unifPair(Th,
         disjunction(
            matchTuple(VDS, emptyEquationSet, Subst),
            D)))
  = substitutionSet(
        (substituteSubstitution(Subst, substMembAxSet(Th, VDS)) ;
         substMembAxSet(Th, VDS)),
        getMatchSolution(unifPair(Th, D))) .
eq getMatchSolution(unifPair(Th, failure))
  = emptySubstitutionSet .

eq eqBreak(T, T')
  = eq(T, T') .
eq eqBreak((T, TL), (T', TL'))
  = equationSet(eq(T, T'), eqBreak(TL, TL')) .

eq greaterEq(Th, eq(T, T'), eq(T1, T1'))
  = orElse(
       max(size(Th, T), size(Th, T'))
            > max(size(Th, T1), size(Th, T1')),
       andThen(max(size(Th, T), size(Th, T'))
                 == max(size(Th, T1), size(Th, T1')),
               (max(size(Th, T), size(Th, T'))
                   - min(size(Th, T), size(Th, T')))
                > (max(size(Th, T1), size(Th, T1'))
                    - min(size(Th, T1), size(Th, T1'))))) .
```

5.3 Confluence

We are ready now to give the implementation of the function `confluenceCheck`, whose declaration already appeared in Section 5.2.

Given a specification $\mathcal{S}$, `critPairs` finds all the critical pairs between the equations in $\mathcal{S}$ considered as rules oriented from left to right. Once all the critical pairs have been generated the trivial ones are removed by the function `delete`, which is applied again to the set of critical pairs resulting from the simplification process. This simplification is achieved by the function `simplify` just reducing both sides of the critical pairs to their normal forms in the given specification. Finally, we apply the function `maximalCPSet` to get the set of maximal critical pairs among those remaining. Note that the function `confluenceCheck` takes two arguments of sorts `Module`; one of them is used to calculate the critical pairs and the other, in which has been introduced the operator `and` (see Section 5.2.1), is used to rewrite.

```
op critPairs : Module -> CritPairSet .
op delete : CritPairSet -> CritPairSet .
op simplify : CritPairSet Module -> CritPairSet .
op maximalCPSet : CritPairSet Module -> CritPairSet .
op maximalCPSetAux :
       CritPair CritPairSet CritPairSet Module -> CritPairSet .
op moreGeneralCP? : CritPair CritPair Module -> Bool .

eq confluenceCheck(Th, Th')
  = maximalCPSet(delete(simplify(delete(critPairs(Th)), Th')), Th) .
```

The function `delete` removes trivial critical pairs.

```
eq delete(critPairSet(cp(T, T'), CPS))
  = if T == T'
    then delete(CPS)
    else critPairSet(cp(T, T'), delete(CPS))
    fi .
eq delete(critPairSet(ccp(T, T', T'', 'true), CPS))
  = if T == T'
    then delete(CPS)
    else critPairSet(ccp(T, T', T'', 'true), delete(CPS))
    fi .
eq delete(emptyCritPairSet)
  = emptyCritPairSet .
```

The function `simplify` rewrites both sides of a critical pair; for conditional critical pairs, it leaves the condition untouched. Note that the module in which the critical pairs are simplified is the one to which was applied the function `addAnd`.

```
eq simplify(critPairSet(cp(T, T'), CPS), Th)
  = critPairSet(cp(meta-reduce(Th, T), meta-reduce(Th, T')),
                simplify(CPS, Th)) .
eq simplify(critPairSet(ccp(T, T', T'', 'true), CPS), Th)
  = critPairSet(ccp(meta-reduce(Th, T),
                    meta-reduce(Th, T'), T'', 'true),
                simplify(CPS, Th)) .
eq simplify(emptyCritPairSet, Th)
  = emptyCritPairSet .
```

In order to get the minimal set of critical pairs we not only have to eliminate trivial ones, but we also need to take care of those that are repeated or less general than some other critical pair. The function maximalCPSet returns the set of most general critical pairs.

```
eq maximalCPSet(critPairSet(CP, CPS), Th)
  = maximalCPSetAux(CP, CPS, emptyCritPairSet, Th) .
eq maximalCPSet(emptyCritPairSet, Th)
  = emptyCritPairSet  .

eq maximalCPSetAux(CP, critPairSet(CP', CPS), CPS', Th)
  = if moreGeneralCP?(CP, CP', Th)
    then maximalCPSetAux(CP, CPS, CPS', Th)
    else if moreGeneralCP?(CP', CP, Th)
         then maximalCPSetAux(
                CP', critPairSet(CPS, CPS'), emptyCritPairSet, Th)
         else maximalCPSetAux(CP, CPS, critPairSet(CP', CPS'), Th)
         fi
    fi .
eq maximalCPSetAux(CP, emptyCritPairSet, critPairSet(CP', CPS), Th)
  = critPairSet(CP, maximalCPSetAux(CP', CPS, emptyCritPairSet, Th)) .
eq maximalCPSetAux(CP, emptyCritPairSet, emptyCritPairSet, Th)
  = CP .

eq moreGeneralCP?(cp(T1, T1'), cp(T2, T2'), Th)
  = orElse(match?(Th, equationSet(eq(T1, T2), eq(T1', T2'))),
           match?(Th, equationSet(eq(T1, T2'), eq(T1', T2)))) .
eq moreGeneralCP?(cp(T1, T1'),
                  ccp(T2, T2', T2'', 'true), Th)
  = orElse(match?(Th, equationSet(eq(T1, T2), eq(T1', T2'))),
           match?(Th, equationSet(eq(T1, T2'), eq(T1', T2)))) .
eq moreGeneralCP?(ccp(T1, T1', T1'', 'true),
                  cp(T2, T2'), Th)
  = orElse(match?(Th,
              equationSet(eq(T1, T2), eq(T1', T2'), eq(T1'', 'true))),
           match?(Th,
              equationSet(eq(T1, T2'), eq(T1', T2), eq(T1'', 'true')))) .
eq moreGeneralCP?(ccp(T1, T1', T1'', 'true),
                  ccp(T2, T2', T2'', 'true), Th)
  = orElse(match?(Th,
              equationSet(eq(T1, T2), eq(T1', T2'), eq(T1'', T2''))),
           match?(Th,
              equationSet(eq(T1, T2'), eq(T1', T2), eq(T1'', T2'')))) .
```

One critical pair is generated for each unifier for every one of the possible non-variable overlappings of the lefthand sides of any two equations of the module.

The critical pairs are calculated by finding all the possible ones for each of the equations in the module (critPairs1) with each one of the other equations in the module including itself (critPairs2). For each pair of equations, their left sides are unified at any nonvarible position of the term of the first equation (critPairs3), and then a critical pair is built up for each one of the solutions of the unification problem (critPairs4).

```
op critPairs1 : Module EquationSet EquationSet EquationSet
        -> CritPairSet .
op critPairs2 : Module Equation EquationSet -> CritPairSet .
op critPairs3 : Module Equation Equation PositionSet -> CritPairSet .
op critPairs4 : Module Equation Equation Position SubstitutionSet
        -> CritPairSet .

eq critPairs(Th)
  = critPairs1(Th, equations(Th), equations(Th), equations(Th)) .

eq critPairs1(Th, equationSet(eq(T, T'), ES), ES', ES'')
  = critPairSet(
      critPairs1(Th, ES, ES', ES''),
      critPairs2(Th, eq(rename(Th, T), rename(Th, T')), ES')) .
eq critPairs1(Th,
             equationSet(ceq(T, T', T'', 'true), ES), ES', ES'')
  = critPairSet(
      critPairs1(Th, ES, ES', ES''),
      critPairs2(Th, ceq(rename(Th, T), rename(Th, T'),
                         rename(Th, T''), 'true), ES')) .
eq critPairs1(Th, emptyEquationSet, ES, ES')
  = emptyCritPairSet .

eq critPairs2(Th, eq(T, T'), equationSet(E, ES))
  = critPairSet(
      critPairs2(Th, eq(T, T'), ES),
      critPairs3(Th, eq(T, T'), E, allNonVarPos(Th, T))) .
eq critPairs2(Th, ceq(T, T', T'', 'true), equationSet(E, ES))
  = critPairSet(
      critPairs2(Th, ceq(T, T', T'', 'true), ES),
      critPairs3(Th, ceq(T, T', T'', 'true),
                 E, allNonVarPos(Th, T))) .
eq critPairs2(Th, E, emptyEquationSet)
  = emptyCritPairSet .

eq critPairs3(Th, eq(T1, T1'), eq(T2, T2'), posSet(P, PS))
  = critPairSet(
      critPairs3(Th, eq(T1, T1'), eq(T2, T2'), PS),
      critPairs4(Th, eq(T1, T1'), eq(T2, T2'),
                 P, unify(Th, getSubterm(T1, P), T2))) .
eq critPairs3(Th, eq(T1, T1'),
             ceq(T2, T2', T2'', 'true), posSet(P, PS))
  = critPairSet(
      critPairs3(Th, eq(T1, T1'), ceq(T2, T2', T2'', 'true), PS),
      critPairs4(Th, eq(T1, T1'), ceq(T2, T2', T2'', 'true),
                 P, unify(Th, getSubterm(T1, P), T2))) .
eq critPairs3(Th, ceq(T1, T1', T1'', 'true),
```

```
              eq(T2, T2'), posSet(P, PS))
  = critPairSet(
      critPairs3(Th, ceq(T1, T1', T1'', 'true), eq(T2, T2'), PS),
      critPairs4(Th, ceq(T1, T1', T1'', 'true), eq(T2, T2'),
                 P, unify(Th, getSubterm(T1, P), T2))) .
eq critPairs3(Th, ceq(T1, T1', T1'', 'true),
              ceq(T2, T2', T2'', 'true), posSet(P, PS))
  = critPairSet(
      critPairs3(Th, ceq(T1, T1', T1'', 'true),
                 ceq(T2, T2', T2'', 'true), PS),
      critPairs4(Th, ceq(T1, T1', T1'', 'true),
                 ceq(T2, T2', T2'', 'true),
                 P, unify(Th, getSubterm(T1, P), T2))) .
eq critPairs3(Th, E, E', emptyPosSet)
  = emptyCritPairSet .

eq critPairs4(Th, eq(T1, T1'), eq(T2, T2'), P,
              substitutionSet(Subst, SubstS))
  = critPairSet(
      cp(substitute(T1', Subst), replace(substitute(T1, Subst),
         substitute(T2', Subst), P)),
      critPairs4(Th, eq(T1, T1'), eq(T2, T2'), P, SubstS)) .
eq critPairs4(Th, ceq(T1, T1', T1'', 'true), eq(T2, T2'), P,
              substitutionSet(Subst, SubstS))
  = critPairSet(
      ccp(substitute(T1', Subst),
          replace(substitute(T1, Subst),
                  substitute(T2', Subst), P),
          substitute(T1'', Subst), 'true),
      critPairs4(Th, ceq(T1, T1', T1'', 'true),
                 eq(T2, T2'), P, SubstS)) .
eq critPairs4(Th, eq(T1, T1'), ceq(T2, T2', T2'', 'true), P,
              substitutionSet(Subst, SubstS))
  = critPairSet(
      ccp(substitute(T1', Subst),
          replace(substitute(T1, Subst),
                  substitute(T2', Subst), P),
          substitute(T2'', Subst), 'true),
      critPairs4(Th, eq(T1, T1'),
                 ceq(T2, T2', T2'', 'true), P, SubstS)) .
eq critPairs4(Th, ceq(T1, T1', T1'', 'true),
              ceq(T2, T2', T2'', 'true), P,
              substitutionSet(Subst, SubstS))
  = critPairSet(
      ccp(substitute(T1', Subst),
          replace(substitute(T1, Subst),
                  substitute(T2', Subst), P),
          'and[substitute(T1'', Subst), substitute(T2'', Subst)],
          'true),
      critPairs4(Th, ceq(T1, T1', T1'', 'true),
```

```
                ceq(T2, T2', T2'', 'true), P, SubstS)) .
eq critPairs4(Th, E, E', P, emptySubstitutionSet)
  = emptyCritPairSet .
```

5.3.1 An Example: The Natural Numbers

Let us start our set of examples with the very simple specification NAT for the natural numbers already presented in Section 4.4.1 to see how the set of maximal critical pairs CCP(NAT) is generated.

```
mod NAT is
  sorts Zero Nat .
  subsort Zero < Nat .
  op 0 : -> Zero .
  op s : Nat -> Nat .
  op (_+_) (_*_) : Nat Nat -> Nat [comm] .
  vars N M : Nat .
  eq 0 + N = N .
  eq s(N) + M = s(N + M) .
  eq 0 * N = 0 .
  eq s(N) * M = M + (N * M) .
endm
```

This specification is perfectly reasonable. Its intended model is $\mathbb{N}$, with the sum and product functions. However it is not confluent. One of the solutions for the unification in the overlapping of the last of the equations with itself at the top generates one nonconfluent critical pair. Let us see in detail how the tool works in this example.

The module cannot be given as such as an input. The tool expects as input the module at the meta-level, i.e., its meta-representation. The input to the tool is then given as:

```
red checking(
  fmod(
     'NAT,
     nilImportList,
     sort(qidSet('Zero, 'Nat)),
     subsort('Zero, 'Nat),
     opDeclList(
        opDecl('0, nilQidList, 'Zero, emptyAttrSet),
        opDecl('s, 'Nat, 'Nat, emptyAttrSet),
        opDecl('+, qidList('Nat, 'Nat), 'Nat, comm),
        opDecl('*, qidList('Nat, 'Nat), 'Nat, comm)),
     varDeclSet(
        varDecl('N, 'Nat), varDecl('M, 'Nat)),
     emptyMembAxSet,
     equationSet(
        eq('+['0, 'N], 'N ),
        eq('+['s['N], 'M], 's['+['N, 'M]]),
        eq('*['0, 'N], '0),
        eq('*['s['N], 'M], '+['M, '*['N, 'M]])))) .
```

The information contained in this module is first of all *prepared* in order to get all the variables that may be needed in the checking process. For example, in checking sort-decreasingness of an equation, we must consider all substitution instances of each equation with variables of smaller sorts. The declarations of these variables have to be introduced in the module. We introduce them in advance. The result of this process is the following module:

```
fmod(
   'NAT,
   nilImportList,
   sort(qidSet('Zero, 'Nat)),
   subsort('Zero, 'Nat),
   opDeclList(
      opDecl('0, nilQidList, 'Zero, emptyAttrSet),
      opDecl('s, 'Nat, 'Nat, emptyAttrSet),
      opDecl('+, qidList('Nat, 'Nat), 'Nat, comm),
      opDecl('*, qidList('Nat, 'Nat), 'Nat, comm)),
   varDeclSet(
      varDecl('N, 'Nat), varDecl('M, 'Nat),
      varDecl('N@', 'Nat), varDecl('M@', 'Nat),
      varDecl('M@'Zero, 'Zero), varDecl('M@'@'Zero, 'Zero),
      varDecl('N@'Zero, 'Zero), varDecl('N@'@'Zero, 'Zero)),
   emptyMembAxSet,
   equationSet(
      eq('+['0, 'N], 'N),
      eq('+[('s['N]), 'M], 's['+['N, 'M]]),
      eq('*['0, 'N], '0),
      eq('*[('s['N]), 'M], '+['M, ('*['N, 'M])])))
```

In this term we can see how the changes affect the variable declarations. Note that for each variable V of sort S in the original module a variable V`@'`, and variables V`@'`S' and V`@'@'`S' for each subsort S' of S, are created.

After these variables have been created, the algorithm proceeds by generating all the critical pairs and then eliminating those in which both sides rewrite to the same term.

One critical pair is generated for each solution of the unification of the terms resulting from the overlapping of the lefthand sides of each pair of equations in all possible non variable subterm positions.

The complete set of critical pairs for this example is:

```
critPairSet(
   cp('0, '0),
   cp('0, '0),
   cp('0, '0),
   cp('0, '+['0, ('*['N, '0])]),
   cp('N@', 'N@'),
   cp('s['N], 's['+['N, '0]]),
   cp('s['+['N@', '0]], 's['N@']),
   cp('s['+['N@', 'M@']], 's['+['N@', 'M@']]),
   cp('s['+['N@', ('s['N])]], 's['+['N, ('s['N@'])]]),
```

```
  cp('+['0, ('*['N@', '0])], '0),
  cp('+['M@', ('*['N@', 'M@'])], '+['M@', ('*['N@', 'M@'])]),
  cp('+[('s['N]), ('*['N@', ('s['N])])],
     '+[('s['N@']), ('*['N, ('s['N@'])])]))
```

In a first step, all the trivial critical pairs are discarded by the `delete` function, leaving the set:

```
critPairSet(
   cp('0, '+['0, ('*['N, '0])]),
   cp('s['N], 's['+['N, '0]]),
   cp('s['+['N@', '0]], 's['N@']),
   cp('s['+['N@', ('s['N])]], 's['+['N, ('s['N@'])]]]),
   cp('+['0, ('*['N@', '0])], '0),
   cp('+[('s['N]), ('*['N@', ('s['N])])],
      '+[('s['N@']), ('*['N, ('s['N@'])])]))
```

Then, each of these critical pairs is simplified. This simplification is done by reducing both terms to their normal forms in the specification using the built-in function `meta-reduce`. The result of this process is the following:

```
critPairSet(
   cp('0, '0),
   cp('0, '0),
   cp('s['N], 's['N]),
   cp('s['N@'], 's['N@']),
   cp('s['s['+['N, 'N@']]], 's['s['+['N, 'N@']]]),
   cp('s['+['N, ('+['N@', ('*['N, 'N@'])])]],
      's['+['N@', ('+['N, ('*['N, 'N@'])])]]))
```

After removing the trivial pairs again, we are left with one critical pair:

```
cp('s['+['N, ('+['N@', ('*['N, 'N@'])])]],
   's['+['N@', ('+['N, ('*['N, 'N@'])])]])
```

We can see how this critical pair came from one of the solutions of the unification corresponding to overlapping at the top the lefthand sides of the equation

```
eq('*['s['N], 'M], '+['M, '*['N, 'M]])
```

and a renamed copy of itself

```
eq('*['s['N@'], 'M@'], '+['M@', '*['N@', 'M@']])
```

The unification of both lefthand sides gives solutions

```
substitutionSet(
   (('N <- 'N@'); ('M <- 'M@')),
   (('M <- ('s['N@'])); ('M@' <- ('s['N]))))
```

One critical pair is generated for each of these substitutions:

```
   critPairSet(
      cp('+['M@', ('*['N@', 'M@'])],
         '+['M@', ('*['N@', 'M@'])]),
      cp('+[('s['N]), ('*['N@', ('s['N])])],
         '+[('s['N@']), ('*['N, ('s['N@'])])]))
```

The first one is one of the trivial critical pairs, and so is eliminated. Reducing each of the terms of the other one in the specification we obtain the critical pair showed above. Note that these nonconfluent critical pairs generated by the tool can be used by the user to detect possible errors in the specification or to try to complete it manually. This last possibility is similar to the process used in automatic completion tools, with the difference that the equations added in these tools are usually hard to understand and that such automatic completion processes often diverge. Here, the user can decide if they are really needed and then the way in which they are going to be added or can be replaced by other equations.

The output of the system is given as follows.

```
rewrites: 7612 in 89ms cpu (99ms real) (84578 rewrites/second)
result CheckingSolution:
   checkingSolution(
      varDeclSet(
         varDecl('N, 'Nat),
         varDecl('N@', 'Nat)),
      cp('s['+['N, ('+['N@', ('*['N, 'N@'])])]],
         's['+['N@', ('+['N, ('*['N, 'N@'])])]]),
      emptyMembAxSet)
```

In this case the set of membership axioms is empty. This means that all the equations are sort-decreasing. We will study the sort-decreasingness checking in Section 5.4. Regarding the set of variables, all the variables in the critical pairs and in the membership axioms are collected and included as the first component of the output triple.

5.4 Sort-decreasingness

Given a specification $\mathcal{S}$ the function `sortDecrCheck` returns $CMB(\mathcal{S})$, that is, the set of maximal membership constraints among those generated for all the specializations of the equations not satisfying the sort-decreasing condition. For each one of the possible specializations of each of the equations in the module (generated by `eqInstanceSet`) the least sort of the term in the lefthand side is compared with the least sort of the term in the righthand side reduced to its normal form. Thus, for each equation $(\forall X, l = r)$ that considered as a rewrite rule fails to satisfy the sort-decreasingness condition, that is, $LS(r\downarrow) \not\leq LS(l)$, a membership axiom of the form mb(r,$LS(l)$) is generated. Similarly, for a conditional equation $(\forall X, l = r$ `if` $c)$ failing to satisfy sort-decreasingness a *conditional* membership axiom of the form cmb(r,$LS(l)$,c) is generated. The function `maximalMASet` returns the set of maximals in the set of membership axioms that takes as input. `moreGeneralMA?` compares two membership axioms.

It is not enough comparing the least sort for the sides of each equation. To check that an equation is sort-decreasing we have to generate all the possible instances of it obtained for each of the possible specializations of its variables. An equation is sort-decreasing if in each of these instances the least sort of the term in the lefthand side is greater or equal than the least sort of the term in the righthand side.

```
op sortDecrCheck1 : Module EquationSet -> MembAxSet .
op eqInstanceSet : Module EquationSet -> EquationSet .
op instanceSet :
        Module Equation SubstitutionSet -> EquationSet .
op genSubstSet :
        Module VarDeclSet Substitution -> SubstitutionSet .
op genSubstSet1 : Module Qid QidSet VarDeclSet Substitution
        -> SubstitutionSet .
op maximalMASet : MembAxSet Module -> MembAxSet .
op maximalMASetAux : MembAx MembAxSet MembAxSet Module -> MembAxSet .
op moreGeneralMA? : MembAx MembAx Module -> Bool .

eq sortDecrCheck(Th)
 = maximalMASet(
       sortDecrCheck1(Th, eqInstanceSet(Th, equations(Th))),
       Th) .

eq sortDecrCheck1(Th, equationSet(eq(T, T'), ES))
  = if sortLeq(Th,
               leastSort(Th, meta-reduce(Th, T')),
               leastSort(Th, T))
    then sortDecrCheck1(Th, ES)
    else membAxSet(mb(T', leastSort(Th, T)),
                   sortDecrCheck1(Th, ES))
    fi .
eq sortDecrCheck1(Th, equationSet(ceq(T, T', T'', 'true), ES))
  = if sortLeq(Th,
               leastSort(Th, meta-reduce(Th, T')),
               leastSort(Th, T))
    then sortDecrCheck1(Th, ES)
    else membAxSet(cmb(T', leastSort(Th, T), T'', 'true),
                   sortDecrCheck1(Th, ES))
    fi .
eq sortDecrCheck1(Th, emptyEquationSet)
  = emptyMembAxSet .

eq eqInstanceSet(Th, equationSet(eq(T, T'), ES))
  = equationSet(
       instanceSet(Th, eq(T, T'),
           genSubstSet(Th, varDeclSetInTerm(Th, T),
                       emptySubstitution)),
        eqInstanceSet(Th, ES)) .
eq eqInstanceSet(Th, equationSet(ceq(T, T', T'', 'true), ES))
  = equationSet(
        instanceSet(
           Th, ceq(T, T', T'', 'true),
           genSubstSet(Th, varDeclSetInTerm(Th, T),
```

```
                       emptySubstitution)),
        eqInstanceSet(Th, ES)) .
eq eqInstanceSet(Th, emptyEquationSet)
  = emptyEquationSet .

eq instanceSet(Th, eq(T, T'), substitutionSet(Subst, SubstS))
  = equationSet(eq(substitute(T, Subst), substitute(T', Subst)),
                instanceSet(Th, eq(T, T'), SubstS)) .
eq instanceSet(Th, ceq(T, T', T'', 'true),
               substitutionSet(Subst, SubstS))
  = equationSet(ceq(substitute(T, Subst), substitute(T', Subst),
                    substitute(T'', Subst), 'true),
                instanceSet(Th, ceq(T, T', T'', 'true), SubstS)) .
eq instanceSet(Th, E, emptySubstitutionSet)
  = emptyEquationSet .

eq genSubstSet(Th, varDeclSet(varDecl(X, S), VDS), Subst)
  = genSubstSet1(Th, X, qidSet(S, lesserSorts(Th, S)), VDS, Subst) .
eq genSubstSet(Th, emptyVarDeclSet, Subst)
  = Subst .

eq genSubstSet1(Th, X, qidSet(S, SS), VDS, Subst)
 = if variableSort(Th, X) == S
   then substitutionSet(
           genSubstSet(Th, VDS, Subst),
           genSubstSet1(Th, X, SS, VDS, Subst))
   else substitutionSet(
           genSubstSet(Th, VDS,
                       ((X <- conc(X, conc('@', S))); Subst)),
           genSubstSet1(Th, X, SS, VDS, Subst))
   fi .
eq genSubstSet1(Th, X, emptyQidSet, VDS, Subst)
  = emptySubstitutionSet .

eq maximalMASet(membAxSet(MA, MAS), Th)
  = maximalMASetAux(MA, MAS, emptyMembAxSet, Th) .
eq maximalMASet(emptyMembAxSet, Th)
  = emptyMembAxSet .

eq maximalMASetAux(MA, membAxSet(MA', MAS), MAS', Th)
  = if moreGeneralMA?(MA, MA', Th)
    then maximalMASetAux(MA, MAS, MAS', Th)
    else if moreGeneralMA?(MA', MA, Th)
         then maximalMASetAux(
                MA', membAxSet(MAS, MAS'), emptyMembAxSet, Th)
         else maximalMASetAux(MA, MAS, membAxSet(MA', MAS'), Th)
         fi
    fi .
eq maximalMASetAux(MA, emptyMembAxSet, membAxSet(MA', MAS), Th)
  = membAxSet(MA, maximalMASetAux(MA', MAS, emptyMembAxSet, Th)) .
```

```
eq maximalMASetAux(MA, emptyMembAxSet, emptyMembAxSet, Th)
  = MA .

eq moreGeneralMA?(mb(T1, So1), mb(T2, So2), Th)
  = andThen(match?(Th, eq(T1, T2)), sortLeq(Th, So1, So2)) .
eq moreGeneralMA?(mb(T1, So1), cmb(T2, So2, T2', 'true), Th)
  = andThen(sortLeq(Th, So1, So2), match?(Th, eq(T1, T2))) .
eq moreGeneralMA?(cmb(T1, So1, T1', 'true), mb(T2, So2), Th)
  = andThen(T1' == 'true,
    andThen(sortLeq(Th, So1, So2), match?(Th, eq(T1, T2)))) .
eq moreGeneralMA?(cmb(T1, So1, T1', 'true),
                  cmb(T2, So2, T2', 'true), Th)
  = andThen(T1' == 'true,
    andThen(sortLeq(Th, So1, So2),
            match?(Th, equationSet(eq(T1, T2), eq(T1', T2))))) .
```

The checking of sort-decreasingness with the above equations is illustrated in detail in the following section with a module INT for integers.

5.4.1 Another Example: The Integers

Let us continue with the specification of the integers introduced in Section 4.4.2, which was presented in the Cafe Workshop in March 1997 by José Meseguer to illustrate the way in which the sort-decreasingness check is accomplished. Let NAT be the module defined in Section 5.3.1.

```
mod INT is
   sorts Int Neg .
   protecting NAT .
   subsorts Zero < Neg < Int .
   subsort Nat < Int .
   ops s p : Int -> Int .
   op p : Neg -> Neg .
   ops (_+_) (_*_) : Int Int -> Int [comm] .
   op -_ : Int -> Int .
   op square : Int -> Int .
   vars N M : Nat .
   vars I J : Int .
   vars P Q : Neg .
   eq s(p(I)) = I .
   eq p(s(I)) = I .
   eq I + 0 = I .
   eq I + s(N) = s(I + N) .
   eq I + p(P) = p(I + P) .
   eq I * 0 = 0 .
   eq I * s(N) = (I * N) + I .
   eq I * p(Q) = (I * Q) + (- I) .
   eq - 0 = 0 .
   eq - s(N) = p(- N) .
   eq - p(P) = s(- P) .
```

```
  eq square(I) = I * I .
endm
```

As pointed out by José Meseguer in his slides, this specification of INT is perfectly reasonable. Its intended model is the ring of the integers with the square function. However:

- It is not confluent. For example, the equations

  ```
  eq I * s(N) = (I * N) + I .
  eq I * p(P) = (I * P) + (- I) .
  ```

  yield an unjoinable critical pair.

$$\begin{tikzcd} & \texttt{s(N)*p(P)} \arrow[dl, "*"'] \arrow[dr, "*"] & \\ \texttt{p(((N*P)+(-N))+P)} & & \texttt{p(((N*P)+P)+(-N))} \end{tikzcd}$$

- It is not sort-decreasing. The equation

  ```
  eq square(I) = I * I .
  ```

  rewrites up. The least sort of the term square(I) is Nat, but it is Int for the term in the righthand side.

As stated above, the input module has to be meta-represented. Then, the input to the tool with the meta-representation of the module INT, with the module NAT already included, is the following:

```
red checking(
  fmod(
    'INT,
    nilImportList,
    sort(qidSet('Zero, 'Nat, 'Neg, 'Int)),
    subsortDeclSet(
      subsort('Zero, 'Nat),
      subsort('Nat, 'Int),
      subsort('Zero, 'Neg),
      subsort('Neg, 'Int)),
    opDeclList(
      opDecl('0, nilQidList, 'Zero, emptyAttrSet),
      opDecl('s, 'Nat, 'Nat, emptyAttrSet),
      opDecl('+, qidList('Nat, 'Nat), 'Nat, comm),
      opDecl('*, qidList('Nat, 'Nat), 'Nat, comm),
      opDecl('s, 'Int, 'Int, emptyAttrSet),
      opDecl('p, 'Int, 'Int, emptyAttrSet),
      opDecl('p, 'Neg, 'Neg, emptyAttrSet),
      opDecl('+, qidList('Int, 'Int), 'Int, comm),
      opDecl('*, qidList('Int, 'Int), 'Int, comm),
      opDecl('-, 'Int, 'Int, emptyAttrSet),
      opDecl('square, 'Int, 'Nat, emptyAttrSet)),
    varDeclSet(
```

```
      varDecl('N, 'Nat), varDecl('M, 'Nat),
      varDecl('I, 'Int), varDecl('J, 'Int),
      varDecl('P, 'Neg), varDecl('Q, 'Neg)),
   emptyMembAxSet,
   equationSet(
      eq('+['0, 'N], 'N ),
      eq('+['s['N], 'M], 's['+['N, 'M]]),
      eq('*['0, 'N], '0),
      eq('*['s['N], 'M], '+['M, '*['N, 'M]]),
      eq('s['p['I]], 'I),
      eq('p['s['I]], 'I),
      eq('+['I, '0], 'I ),
      eq('+['I, 's['N]], 's['+['I, 'N]]),
      eq('+['I, 'p['Q]], 'p['+['I, 'Q]]),
      eq('*['I, '0], '0),
      eq('*['I, 's['N]], '+['*['I, 'N], 'I]),
      eq('*['I, 'p['Q]], '+['*['I, 'Q], '-['I]]),
      eq('-['0], '0),
      eq('-['s['N]], 'p['-['N]]),
      eq('-['p['Q]], 's['-['Q]]),
      eq('square['I], '*['I, 'I])))) .
```

To check sort-decreasingness, all the instances specializing the variables to smaller sorts are generated for each equation. Let us consider for example the instances for the equation

```
   eq square(I) = I * I .
```

Note that for a variable `I` of sort `Int` the *preparation step* creates variables `I@'Zero`, `I@'Nat` and `I@'Neg`, of sorts `Zero`, `Nat` and `Neg`, respectively. Then, we have that the following equations are generated:

```
equationSet(
  eq('square['I], '*['I, 'I]),
  eq('square['I@'Zero], '*['I@'Zero, 'I@'Zero]),
  eq('square['I@'Nat], '*['I@'Nat, 'I@'Nat]),
  eq('square['I@'Neg], '*['I@'Neg, 'I@'Neg]))
```

For each of these instances the least sort of the term in the lefthand side is compared with the least sort of the term in the righthand side reduced to its normal form. The set of instances for which the sort-decreasing condition is not satisfied is:

```
equationSet(
  eq('square['I], '*['I, 'I]),
  eq('square['I@'Neg], '*['I@'Neg, 'I@'Neg]))
```

The proof obligations generated are:

```
membAxSet(
  mb('*['I, 'I], 'Nat),
  mb('*['I@'Neg, 'I@'Neg], 'Nat))
```

It is easy to see that the first membership axiom is more general than the second. Therefore, only the first is included in the output of the tool, together with three critical pairs.

```
rewrites: 139041 in 1659ms cpu (1669ms real) (83759 rewrites/second)
result CheckingSolution:
checkingSolution(
   varDeclSet(
      varDecl('N, 'Nat), varDecl('I, 'Int),
      varDecl('N@', 'Nat), varDecl('Q, 'Neg),
      varDecl('Q@', 'Neg)),
   critPairSet(
      cp('s['+['N, ('+['N@', ('*['N, 'N@'])])]],
         's['+['N@', ('+['N, ('*['N, 'N@'])])]]),
      cp('p['+['Q, ('+[('-['N@']), ('*['N@', 'Q])])]],
         '+[('p['-['N@']]), ('+['Q, ('*['N@', 'Q])])]),
      cp('+[('s['-['Q]]), ('+[('-['Q@']), ('*['Q, 'Q@'])])],
         '+[('s['-['Q@']]), ('+[('-['Q]), ('*['Q, 'Q@'])])])),
   mb('*['I, 'I], 'Nat))
```

Regarding the critical pairs, the first one comes from the overlapping of the equation

```
eq I * s(N) = (I * N) + I .
```

with a renamed copy of itself at the top. The critical pair generated is

```
cp('+[('*[('s['N]), 'N@']), ('s['N])],
   '+[('*[('s['N@']), 'N]), ('s['N@'])]),
```

which is reduced to

```
cp('s['+['N, ('+['N@', ('*['N, 'N@'])])]],
   's['+['N@', ('+['N, ('*['N, 'N@'])])]]).
```

The same critical pair is also generated in the overlapping of the equation

```
eq s(N) * M = M + (N * M) .
```

with itself at the top. Nevertheless, only one of them is taken; the other is eliminated in the last step of the process.

The overlapping at the top of equations

```
eq I * s(N) = (I * N) + I .
eq I * p(Q) = (I * Q) + (- I) .
```

generates two critical pairs:

```
cp('+[('*[('s['N]), 'Q@']), ('-['s['N]])],
   '+[('*[('p['Q@']), 'N]), ('p['Q@'])]),
cp('+[('*[('p['Q]), 'N@']), ('p['Q])],
   '+[('*[('s['N@']), 'Q]), ('-['s['N@']])]),
```

These critical pairs are the same one actually. So, after reducing them we take only one of them.

And finally, the last of these critical pairs comes from the overlapping of the equation

```
eq I * p(Q) = (I * Q) + (- I) .
```

with a renamed copy of itself at the top, generating the critical pair

```
cp('+[('*[('p['Q]), 'Q@']), ('-['p['Q]])],
   '+[('*[('p['Q@']), 'Q]), ('-['p['Q@']])]),
```

which is reduced to

```
cp('+[('s['-['Q]]), ('+[('-['Q@']), ('*['Q, 'Q@'])])],
   '+[('s['-['Q@']]), ('+[('-['Q]), ('*['Q, 'Q@'])])]).
```

5.5 How to Use the Church-Rosser Checker

This section and Section 5.6 illustrate with examples the use of the Church-Rosser checker tool, and some methods that—using the feedback provided by the tool—can help the user establish that his/her specification is ground Church-Rosser.

We assume a context of use very different from the usual context for completing an equational theory. The starting point for completing a theory, say the theory of groups, is an equational theory that is *not* Church-Rosser. A Knuth-Bendix-like completion process then attempts to make it so by *automatically adding* new oriented equations.

In our case, however, we assume that the user has developed an *executable specification* of his/her intended system with an initial algebra semantics, and that this specification has already been *tested* with examples, so that the user is in fact confident that the specification is *ground Church-Rosser*, and wants only to check this property with the tool.

Of course, the tool can only guarantee success when the user's specification is Church-Rosser, and not just ground Church-Rosser. That is, not generating any proof obligations is only a *sufficient* condition. But in many cases of interest—particularly for specifications with sophisticated sort orderings such as, for example, the specifications of the number hierarchy discussed in this section and in Section 5.6—the specification will typically be *ground* Church-Rosser, but not Church-Rosser, so that a collection of critical pairs and of membership axioms will be returned by the tool as proof obligations.

The important methodological question is what to do, or not do, with these proof obligations. As the examples that we discuss illustrate, what should *not* be done is to let an automatic completion process add new equations to the user's specification in a mindless way. In some cases this is even impossible, because the critical pair in question cannot be oriented. In many cases it will certainly lead to a nonterminating process. In any case, it will modify the user's specification in ways that can make difficult for the user in question to recognize the final result, if any, as intuitively equivalent to the original specification.

The feedback of the tool should instead be used as a guide for *careful thought* about one's specification. As several of the examples discussed in detail indicate, by analyzing the critical pairs returned, the user can understand why they could not be joined. It may be a mistake that must be corrected. More often, however, it is not a matter of a mistake, but of an axiom that is either *too general*—so that its very generality makes joining the critical pair impossible, because no more equations can apply to it—or *amenable to an equivalent formulation* that is unproblematic—for example, by reordering the parentheses for an operator that is ground-associative—or both. In any case, it is the user himself/herself who must study where the problem comes from, and how to fix it by modifying the specification. Interaction with the tool then provides a way of modifying the original specification and ascertaining that the new version passes the test or is a good step towards that goal.

Since the user's specification has an *initial* algebra semantics and the property of interest is checking that it is *ground* Church-Rosser, the proof obligations returned by the tool are *inductive* proof obligations. Therefore, after having introduced some modifications that may already eliminate some of the critical pairs and memberships generated by the tool, the user may be left with proof obligations for which the best approach is not any further modification of the specification, but, instead, an inductive proof. This is illustrated in Section 5.5.2 with an specification of the integers with a function `square : Int -> Nat` for which a membership proof obligation

$$(\forall \mathtt{I} : \mathtt{Int})\ \mathtt{I} * \mathtt{I} : \mathtt{Nat}$$

is generated by the tool. Provided that no equational lemmas are used in the proof, the inductive theorem prover presented in Section 4 can be used for this purpose, and in fact succeeds in proving this proof obligation.

Inductive proof of the joinability of critical pairs is a thornier issue, for which we lack at present good methods. The problem is that, if we have a critical pair $cp(t, t')$ generated by the tool for a module M, proving inductively

$$M \vdash_{\mathrm{ind}} t = t'$$

is *not* sufficient for ensuring joinability. This is because such a proof can add new equations as lemmas that may destroy the Church-Rosser property, so that joinability with those new equations does not guarantee joinability with the original equations. What must instead be proved inductively is

$$M \vdash_{\mathrm{ind}} t \downarrow t'$$

but this requires new inductive methods that, as far as we know, have not yet been developed.

A related unresolved methodological issue is what to do with *conditional* critical pairs or memberships whose conditions are *unsatisfiable*. For example, the condition may be of the form

$$\mathtt{tt} = \mathtt{ff}$$

where `tt`, `ff` are constants in a user-defined Boolean sort. If we already *knew* that the specification was Church-Rosser, we could conclude from the fact that `tt` and `ff` are irreducible that they are different, so that the condition cannot be satisfied. But this is precisely what we need to prove. It is quite possible that a modular/hierarchical approach could be used, in conjunction with new inductive proof methods, to establish the unsatisfiability of such conditions and then discard the corresponding proof obligations. But such an approach has still to be developed.

5.5.1 Making the Naturals Church-Rosser

In Section 5.3.1 we studied the confluence of a simple specification for the natural numbers. Let us reconsider it and analyze in more detail why does the checker give that output, and how can we make it Church-Rosser. The specification is the following:

```
mod NAT is
  sorts Zero Nat .
  subsort Zero < Nat .
  op 0 : -> Zero .
  op s : Nat -> Nat .
  op (_+_) (_*_) : Nat Nat -> Nat [comm] .
  vars N M : Nat .
  eq 0 + N = N .
  eq s(N) + M = s(N + M) .
  eq 0 * N = 0 .
  eq s(N) * M = M + (N * M) .
endm
```

As indicated in Section 5.3.1, giving to the checker the input:

```
red checking(
  fmod(
     'NAT,
     nilImportList,
     sort(qidSet('Zero, 'Nat)),
     subsort('Zero, 'Nat),
     opDeclList(
        opDecl('0, nilQidList, 'Zero, emptyAttrSet),
        opDecl('s, 'Nat, 'Nat, emptyAttrSet),
        opDecl('+, qidList('Nat, 'Nat), 'Nat, comm),
        opDecl('*, qidList('Nat, 'Nat), 'Nat, comm)),
     varDeclSet(
        varDecl('N, 'Nat), varDecl('M, 'Nat)),
     emptyMembAxSet,
     equationSet(
        eq('+['0, 'N], 'N ),
        eq('+['s['N], 'M], 's['+['N, 'M]]),
        eq('*['0, 'N], '0),
        eq('*['s['N], 'M], '+['M, '*['N, 'M]])))) .
```

we get as output:

```
rewrites: 7612 in 89ms cpu (99ms real) (84578 rewrites/second)
result CheckingSolution:
   checkingSolution(
      varDeclSet(
         varDecl('N, 'Nat),
         varDecl('N@', 'Nat)),
      cp('s['+['N, ('+['N@', ('*['N, 'N@'])])]],
         's['+['N@', ('+['N, ('*['N, 'N@'])])]]),
      emptyMembAxSet)
```

The approach of classical completion systems consists in taking this critical pair and add it to the specification after orienting it somehow. The user of the Church-Rosser checker tool can in some cases succeed by orienting a critical pair, adding it to the specification, checking termination, and resubmitting the modified specification to the checker. However, in this case it is clear that this critical pair cannot be oriented, since, however we were to orient it, its addition would turn the specification into a non terminating one. The way to handle this with our approach consists in studing the specification, and, in particular, the equations generating the critical pairs, and trying to find a "smart" solution modifying the specification in a way as intuitive and as clear as possible, since it is the user who decides the modifications to apply. The key is then to give the user the information needed to allow him to proceed as he thinks best.

In this case we have that the critical pair comes from the overlap of the equation

```
eq('*['s['N], 'M], '+['M, '*['N, 'M]])
```

with a renamed copy of itself at the top, that is, its overlapping with this other one:

```
eq('*['s['N@'], 'M@'], '+['M@', '*['N@', 'M@']])
```

In this overlapping we have several solutions for the unification problem. The one generating the critical pair is $\{$`('M` $\leftarrow$ `'s['N@'])`, `('M@'` $\leftarrow$ `'s['N])`$\}$. This means that one term of the form `s(N) * s(M)` can be rewriting to terms `s(N + (M + (N * M)))` and `s(M + (N + (N * M)))`. Note that this terms cannot be further reduced, but they would be if they were ground, that is, the result of the tool does not contradict the specification's ground confluence.

In many cases, if the specification is ground confluent, we can eliminate the critical pairs just by writing the equations in some other way, or perhaps by adding some new equations. As already mentioned, in other cases the best possibility may be to take the critical pair and add it as an equation after orienting it somehow, but, as we have explained, this critical pair cannot be oriented. In this case, what we can do is to rewrite the "problematic" equation as

```
eq('*['s['N], 's['M]], 's['+['+['N, 'M], '*['N, 'M]]])
```

Replacing the original equation by this other one in the input the result obtained is as follows.

```
rewrites: 7174 in 89ms cpu (99ms real) (79711 rewrites/second)
result CheckingSolution:
   checkingSolution(emptyVarDeclSet, emptyCritPairSet, emptyMembAxSet)
```

This means that, once termination has been checked, we are guaranteed that the modified specification is confluent. Note the way in which we have associated the variables in the righthand side. If instead we were to modify the equation as

```
eq('*['s['N], 's['M]], 's['+['N, '+['M, '*['N, 'M]]]])
```

we would be in exactly the same situation, and the same critical pair is generated. The point is that associativity of addition has not been declared as an attribute, and therefore the order of the parentheses is crucial to achieve confluence of the critical pair generated.

Since we do not have any proof obligation for sort-decreasingness we can conclude that the specification is Church-Rosser.

5.5.2 Making the Integers Church-Rosser

Let us continue now with the specification for the integers presented in Section 5.4.1. There, we discussed in detail from which equations each of the critical pairs were coming. Let us now apply our technique to eliminate them. We will see also how to take care of the membership proof obligation generated.

In Section 5.4.1 we saw that the output given by the tool for the specification was the following:

```
rewrites: 156159 in 1809ms cpu (1819ms real) (86275 rewrites/second)
result CheckingSolution:
checkingSolution(
   varDeclSet(
      varDecl('N, 'Nat), varDecl('I, 'Int),
      varDecl('N@', 'Nat), varDecl('Q, 'Neg),
      varDecl('Q@', 'Neg)),
   critPairSet(
      cp('s['+['N, ('+['N@', ('*['N, 'N@'])])]],
         's['+['N@', ('+['N, ('*['N, 'N@'])])]]),
      cp('p['+['Q, ('+[('-['N@']), ('*['N@', 'Q])])]],
         '+[('p['-['N@']]), ('+['Q, ('*['N@', 'Q])])]),
      cp('+[('s['-['Q]]), ('+[('-['Q@']), ('*['Q, 'Q@'])])],
         '+[('s['-['Q@']]), ('+[('-['Q]), ('*['Q, 'Q@'])])])),
   mb('*['I, 'I], 'Nat))
```

And we studied there also in detail that the critical pairs were produced by the equations

```
        eq s(N) * M = M + (N * M) .
        eq I * s(N) = (I * N) + I .
        eq I * p(Q) = (I * Q) + (- I) .
```

in different ways. The first thing we should do is to apply the results in a modular way, that is, we should apply to the specifications the changes that have taken place in specifications being imported. In this case, we should incorporate the changes making the specification of the natural numbers confluent in the previous section. Observing the results and the considerations of Section 5.4.1, we realize that these critical pairs are generated for exactly the same reasons that generated the critical pair in the specification for the natural numbers in Section 5.5.1, and that we can make similar modifications on these ones. We need the following equations:

```
        eq s(N) * s(M) = s((N + M) + (N * M)) .
        eq p(P) * s(N) = s((P + - N) + (P * N)) .
        eq p(P) * p(Q) = s((- P + - Q) + (P * Q)) .
```

With this new equations, we have:

```
red checking(
  fmod(
     'INT,
     nilImportList,
     sort(qidSet('Zero, 'Nat, 'Neg, 'Int)),
     subsortDeclSet(
```

```
        subsort('Zero, 'Nat),
        subsort('Nat, 'Int),
        subsort('Zero, 'Neg),
        subsort('Neg, 'Int)),
     opDeclList(
        opDecl('0, nilQidList, 'Zero, emptyAttrSet),
        opDecl('s, 'Nat, 'Nat, emptyAttrSet),
        opDecl('+, qidList('Nat, 'Nat), 'Nat, comm),
        opDecl('*, qidList('Nat, 'Nat), 'Nat, comm),
        opDecl('s, 'Int, 'Int, emptyAttrSet),
        opDecl('p, 'Int, 'Int, emptyAttrSet),
        opDecl('p, 'Neg, 'Neg, emptyAttrSet),
        opDecl('+, qidList('Int, 'Int), 'Int, comm),
        opDecl('*, qidList('Int, 'Int), 'Int, comm),
        opDecl('-, 'Int, 'Int, emptyAttrSet),
        opDecl('square, 'Int, 'Nat, emptyAttrSet)),
     varDeclSet(
        varDecl('N, 'Nat), varDecl('M, 'Nat),
        varDecl('I, 'Int), varDecl('J, 'Int),
        varDecl('P, 'Neg), varDecl('Q, 'Neg)),
     emptyMembAxSet,
     equationSet(
        eq('+['0, 'N], 'N ),
        eq('+['s['N], 'M], 's['+['N, 'M]]),
        eq('*['0, 'N], '0),
        eq('*['s['N], 's['M]], 's['+['+['M, 'N], '*['N, 'M]]]),
        eq('s['p['I]], 'I),
        eq('p['s['I]], 'I),
        eq('+['I, '0], 'I ),
        eq('+['I, 's['N]], 's['+['I, 'N]]),
        eq('+['I, 'p['Q]], 'p['+['I, 'Q]]),
        eq('*['I, '0], '0),
        eq('*['s['N], 's['M]], 's['+['*['N, 'M], '+['N, 'M]]]),
        eq('*['p['P], 's['N]], 's['+['*['P, 'N], '+['P, '-['N]]]]),
        eq('*['p['P], 'p['Q]], 's['+['*['P, 'Q], '+['-['P], '-['Q]]]]),
        eq('-['0], '0),
        eq('-['s['N]], 'p['-['N]]),
        eq('-['p['Q]], 's['-['Q]]),
        eq('square['I], '*['I, 'I])))) .

rewrites: 114757 in 1359ms cpu (1369ms real) (84380 rewrites/second)
result CheckingSolution:
  checkingSolution(
     varDecl('I, 'Int),
     emptyCritPairSet,
     mb('*['I, 'I], 'Nat))
```

Regarding sort-decreasingness, we need to prove inductively the membership constraint given. That is, we have to treat it as the proof obligation that has to be satisfied in order to be able to assert that the specification is sort decreasing. In this case, we have to prove INT $\vdash_{\text{ind}}$ `(∀I) I * I: Nat`. This can be done using the theorem prover presented in Section 4. In fact, in Section 4.4.2 we proved EINT $\vdash_{\text{ind}}$ `(∀I) square(I): Nat`. without any use of equational lemmas. There, it was introduced as a Lemma to prove the main goal EINT $\vdash_{\text{ind}}$ `(∀I) square(I): Nat`. Notice that EINT $\vdash_{\text{ind}}$ `(∀I) I * I: Nat` implies INT $\vdash_{\text{ind}}$ `(∀I) I * I: Nat` because this is, somehow, a weaker assertion. Therefore, we have successfully transformed our original INT specification into one that is Church-Rosser.

5.6 Additional Examples

We present here several additional examples to illustrate the behavior of the tool. No attempt is made to study each of them in full detail, although we will make some comments about them. Also, our intention is not to give a large collection of examples, just a representative collection. We will start with new specifications for natural numbers and integers, rational numbers, arrays, lists and stacks.

5.6.1 Natural Numbers (II)

We give now a specification for the natural numbers quite similar to the one presented in the manual of OBJ ([10], page 77), although with some modifications.

```
fmod NAT is
  sorts Nat NzNat Zero .
  subsorts Zero NzNat < Nat .
  op 0 : -> Zero .
  op s_ : Nat -> NzNat .
  op p_ : NzNat -> Nat .
  op _+_ : Nat Nat -> Nat [comm] .
  op _*_ : Nat Nat -> Nat [comm] .
  op _*_ : NzNat NzNat -> NzNat [comm] .
  op _>_ : Nat Nat -> Bool .
  op d : Nat Nat -> Nat [comm] .
  op quot : Nat NzNat -> Nat .
  op gcd : NzNat NzNat -> NzNat [comm] .
  vars N M : Nat . vars N' M' : NzNat .
  eq p s N = N .
  eq N + 0 = N .
  eq (s N) + (s M) = s s (N + M) .
  eq N * 0 = 0 .
  eq (s N) * (s M) = s (N + (M + (N * M))) .
  eq 0 > M = false .
  eq N' > 0 = true .
  eq s N > s M = N > M .
  eq d(0,N) = N .
  eq d(s N, s M) = d(N,M) .
  ceq quot(N,M') = s quot(d(N,M'),M') if N > M' .
  eq quot(M',M') = s 0 .
  ceq quot(N,M') = 0 if M' > N .
  eq gcd(N',N') = N' .
  ceq gcd(N',M') =  gcd(d(N',M'),M') if N' > M' .
```

```
endfm
```

Here is the corresponding input to the tool:

```
red checking(
     fmod('NAT,
          nilImportList,
          sort(qidSet('Zero, 'NzNat, 'Nat, 'Boolean)),
          subsortDeclSet(subsort('Zero, 'Nat),
                         subsort('NzNat, 'Nat)),
          opDeclList(
             opDecl('T, nilQidList, 'Boolean, emptyAttrSet),
             opDecl('F, nilQidList, 'Boolean, emptyAttrSet),
             opDecl('0, nilQidList, 'Zero, emptyAttrSet),
             opDecl('s, 'Nat, 'NzNat, emptyAttrSet),
             opDecl('p, 'NzNat, 'Nat, emptyAttrSet),
             opDecl('+, qidList('Nat, 'Nat), 'Nat, comm),
             opDecl('*, qidList('Nat, 'Nat), 'Nat, comm),
             opDecl('*, qidList('NzNat, 'NzNat), 'NzNat, comm),
             opDecl('greater, qidList('Nat, 'Nat), 'Boolean, emptyAttrSet),
             opDecl('d, qidList('Nat, 'Nat), 'Nat, comm),
             opDecl('quot, qidList('Nat, 'NzNat), 'Nat, emptyAttrSet),
             opDecl('gcd, qidList('NzNat, 'NzNat), 'NzNat, comm)),
          varDeclSet(varDecl('B, 'Boolean),
                     varDecl('N, 'Nat), varDecl('M, 'Nat), varDecl('R, 'Nat),
                     varDecl('NN, 'NzNat), varDecl('NM, 'NzNat)),
          emptyMembAxSet,
          equationSet(
             eq('p['s['N]], 'N),
             eq('+['N, '0], 'N ),
             eq('+['s['N], 's['M]], 's['s['+['N, 'M]]]),
             eq('*['N, '0], '0),
             eq('*['s['N], 's['M]], 's['+['N, '+['M, '*['N, 'M]]]]),
             eq('greater['0, 'M], 'F),
             eq('greater['NN, '0], 'T),
             eq('greater['s['N], 's['M]], 'greater['N, 'M]),
             eq('d['0, 'N], 'N),
             eq('d['s['N], 's['M]], 'd['N, 'M]),
             ceq('quot['N, 'NM], 's['quot['d['N,'NM], 'NM]],
                 '_==_['greater['N, 'NM], 'T], 'true),
             eq('quot['NM, 'NM], 's['0]),
             ceq('quot['N, 'NM], '0, '_==_['greater['NM, 'N], 'T], 'true),
             eq('gcd['NN, 'NN], 'NN),
             ceq('gcd['NN,'NM], 'gcd['d['NN, 'NM], 'NM],
                 '_==_['greater['NN, 'NM], 'T], 'true)))) .
```

The output is:

```
rewrites: 90946 in 929ms cpu (939ms real) (97791 rewrites/second)
result CheckingSolution:
```

```
checkingSolution(
   varDeclSet(
      varDecl('NN, 'NzNat), varDecl('NM, 'NzNat),
      varDecl('N@', 'Nat), varDecl('M@', 'Nat),
      varDecl('NN@', 'NzNat), varDecl('NM@', 'NzNat)),
   critPairSet(
      cp('s['+['N@', ('+['M@', ('*['N@', 'M@'])])]],
         's['+['M@', ('+['N@', ('*['N@', 'M@'])])]]),
      ccp('0, 's['0], '_==_[('greater['NM@', 'NM@']), 'T], 'true),
      ccp('0,
          's['quot[('d['N@', 'NM@']), 'NM@']],
          'and[('_==_[('greater['NM@', 'N@']), 'T]),
               ('_==_[('greater['N@', 'NM@']), 'T])],
          'true),
      ccp('NN@',
          'gcd['NN@', ('d['NN@', 'NN@'])],
          '_==_[('greater['NN@', 'NN@']), 'T],
          'true),
      ccp('s['0],
          's['quot[('d['NM@', 'NM@']), 'NM@']],
          '_==_[('greater['NM@', 'NM@']), 'T],
          'true),
      ccp('gcd['NM@', ('d['NN@', 'NM@'])],
          'gcd['NN@', ('d['NN@', 'NM@'])],
          'and[('_==_[('greater['NN@', 'NM@']), 'T]),
               ('_==_[('greater['NM@', 'NN@']), 'T])],
          'true)),
   cmb('gcd[('d['NN, 'NM]), 'NM], 'NzNat,
       '_==_[('greater['NN, 'NM]), 'T], 'true))
```

We have some critical pairs and one proof obligation for sort-decreasingness. Let us study them.

After a quick overview of the critical pairs generated, intuitively, we realize that we do not have to worry about most of them. Nevertheless, eliminate conditional critical pairs with conditions that cannot be satisfied or even to reduce the conditions if possible is a tricky matter for which we do not have good methods at the moment. The difficult question is how to prove that two terms are not equal if we do not know whether our specification is confluent. Some research needs to be done in this direction before deciding how to handle these cases. For the moment the user will have to decide what to do with the critical pairs generated, and may have to consider a modified specification.

Let us look for example at the critical pair

```
ccp('0, 's['0], '_==_[('greater['NM@', 'NM@']), 'T], 'true),
```

A natural number cannot be greater than itself, and the condition will never be satisfied. Therefore, in an intuitive sense we do not need to worry about this critical pair.

The same happens with the following two critical pairs:

```
ccp('NN@',
```

```
    'gcd['NN@', ('d['NN@', 'NN@'])],
    '_==_[('greater['NN@', 'NN@']), 'T],
    'true),
ccp('s['0],
    's['quot[('d['NM@', 'NM@']), 'NM@']],
    '_==_[('greater['NM@', 'NM@']), 'T],
    'true),
```

The conditions of the following other two conditional critical pairs cannot be satisfied either, although in this case it is because the condition is given as a conjunction of two equalities that cannot be true simultaneously.

```
ccp('0,
    's['quot[('d['N@', 'NM@']), 'NM@']],
    'and[('_==_[('greater['NM@', 'N@']), 'T]),
         ('_==_[('greater['N@', 'NM@']), 'T])],
    'true),
ccp('gcd['NM@', ('d['NN@', 'NM@'])],
    'gcd['NN@', ('d['NN@', 'NM@'])],
    'and[('_==_[('greater['NN@', 'NM@']), 'T]),
         ('_==_[('greater['NM@', 'NN@']), 'T])],
    'true)),
```

There is only one critical pair remaining to be considered, namely,

```
cp('s['+['N@', ('+['M@', ('*['N@', 'M@'])])]],
   's['+['M@', ('+['N@', ('*['N@', 'M@'])])]])
```

Note that this critical pair is the same as the one we obtained for the specification of natural numbers given in Section 5.5.1. Although the equations defining the product function are different here, the problem is the same. In fact, one of the equations for the product, namely,

```
eq('*['s['N], 's['M]], 's['+['N, '+['M, '*['N, 'M]]]])
```

is very similar to the equation that we discarded at the end of that section. We can choose the same solution taken there, that is, replacing this equation by the equation

```
eq('*['s['N], 's['M]], 's['+['+['N, 'M], '*['N, 'M]]]),
```

Or we can try with some other alternative; for example, given the sort hierarchy in this specification, we can write the equation as

```
eq('*['NN, 's['M]], '+['M, '*['NN, 'M]])
```

Note that in this second case the first argument must be of sort `NzNat`, since if it took value zero then we would get again a non confluent specification. But this second change leads us to a critical pair similar to the one we want to eliminate, so it does not seem to be a good option.

Taking the first option we get:

```
red checking(
    fmod('NAT,
         nilImportList,
```

```
          sort(qidSet('Zero, 'NzNat, 'Nat, 'Boolean)),
          subsortDeclSet(subsort('Zero, 'Nat),
                         subsort('NzNat, 'Nat)),
          opDeclList(
             opDecl('T, nilQidList, 'Boolean, emptyAttrSet),
             opDecl('F, nilQidList, 'Boolean, emptyAttrSet),
             opDecl('0, nilQidList, 'Zero, emptyAttrSet),
             opDecl('s, 'Nat, 'NzNat, emptyAttrSet),
             opDecl('p, 'NzNat, 'Nat, emptyAttrSet),
             opDecl('+, qidList('Nat, 'Nat), 'Nat, comm),
             opDecl('*, qidList('Nat, 'Nat), 'Nat, comm),
             opDecl('*, qidList('NzNat, 'NzNat), 'NzNat, comm),
             opDecl('greater, qidList('Nat, 'Nat), 'Boolean, emptyAttrSet),
             opDecl('d, qidList('Nat, 'Nat), 'Nat, comm),
             opDecl('quot, qidList('Nat, 'NzNat), 'Nat, emptyAttrSet),
             opDecl('gcd, qidList('NzNat, 'NzNat), 'NzNat, comm)),
          varDeclSet(
             varDecl('B, 'Boolean),
             varDecl('N, 'Nat), varDecl('M, 'Nat), varDecl('R, 'Nat),
             varDecl('NN, 'NzNat), varDecl('NM, 'NzNat)),
          emptyMembAxSet,
          equationSet(
             eq('p['s['N]], 'N),
             eq('+['N, '0], 'N ),
             eq('+['s['N], 's['M]], 's['s['+['N, 'M]]]),
             eq('*['N, '0], '0),
             eq('*['s['N], 's['M]], 's['+['+['N, 'M], '*['N, 'M]]]),
             eq('greater['0, 'M], 'F),
             eq('greater['NN, '0], 'T),
             eq('greater['s['N], 's['M]], 'greater['N, 'M]),
             eq('d['0, 'N], 'N),
             eq('d['s['N], 's['M]], 'd['N, 'M]),
             ceq('quot['N, 'NM], 's['quot['d['N,'NM], 'NM]],
                 '_==_['greater['N, 'NM], 'T], 'true),
             eq('quot['NM, 'NM], 's['0]),
             ceq('quot['N, 'NM], '0, '_==_['greater['NM, 'N], 'T], 'true),
             eq('gcd['NN, 'NN], 'NN),
             ceq('gcd['NN,'NM], 'gcd['d['NN, 'NM], 'NM],
                 '_==_['greater['NN, 'NM], 'T], 'true)))) .

rewrites: 89434 in 919ms cpu (929ms real) (97210 rewrites/second)
result CheckingSolution:
checkingSolution(
   varDeclSet(
      varDecl('NN, 'NzNat), varDecl('NM, 'NzNat),
      varDecl('N@', 'Nat), varDecl('NN@', 'NzNat),
      varDecl('NM@', 'NzNat)),
   critPairSet(
      ccp('0, 's['0], '_==_[('greater['NM@', 'NM@']), 'T], 'true),
      ccp('0,
```

```
          's['quot[('d['N@', 'NM@']), 'NM@']],
          'and[('_==_[('greater['NM@', 'N@']), 'T]),
               ('_==_[('greater['N@', 'NM@']), 'T])],
         'true),
     ccp('NN@',
          'gcd['NN@', ('d['NN@', 'NN@'])],
          '_==_[('greater['NN@', 'NN@']), 'T],
          'true),
     ccp('s['0],
          's['quot[('d['NM@', 'NM@']), 'NM@']],
          '_==_[('greater['NM@', 'NM@']), 'T],
          'true),
     ccp('gcd['NM@', ('d['NN@', 'NM@'])],
          'gcd['NN@', ('d['NN@', 'NM@'])],
          'and[('_==_[('greater['NN@', 'NM@']), 'T]),
          ('_==_[('greater['NM@', 'NN@']), 'T])], 'true)),
 cmb('gcd[('d['NN, 'NM]), 'NM], 'NzNat,
      '_==_[('greater['NN, 'NM]), 'T], 'true))
```

We can see that none of the conditions in the critical pairs can be satisfied. Therefore, assuming that we have checked the termination of the modified specification, we can gain an intuitive assurance that our specification is ground confluent. But in order to have a full proof we still need to find proof methods to check the unsatisfiability of the condition in each of the critical pairs.

Let us study now the membership proof obligation

```
cmb('gcd[('d['NN, 'NM]), 'NM], 'NzNat,
    '_==_[('greater['NN, 'NM]), 'T], 'true))
```

It comes from the equation

```
ceq('gcd['NN,'NM],
    'gcd['d['NN, 'NM], 'NM],
    '_==_['greater['NN, 'NM], 'T], 'true)
```

Since `gcd` has been declared as

```
opDecl('gcd, qidList('NzNat, 'NzNat), 'NzNat, comm),
```

and the function `d` has been declared as

```
opDecl('d, qidList('Nat, 'Nat), 'Nat, comm),
```

the term `'gcd['d['NN, 'NM], 'NM]` does not have sort `NzNat`, which then gives rise to the proof obligation above. This membership axiom has to be proved to guarantee the sort decreasingness of the specification. Unfortunately, the present inductive theorem prover does not handle proofs of conditional memberships, and, in addition, we also lack at the present methods to handle the confluence assumption hidden in the equality of the condition.

5.6.2 Integers (II)

We give now a specification for the integers also similar to that presented in the manual of OBJ ([10, page 77]). Let us suppose that the module NAT being imported is the one presented in the previous section, but instead of considering the original one, let us consider the one resulting from the modifications introduced there.

```
fmod INT is
  sorts Int NzInt .
  protecting NAT .
  subsort Nat < Int .
  subsorts NzNat < NzInt < Int .
  op -_ : Int -> Int .
  op -_ : NzInt -> NzInt .
  op _+_ : Int Int -> Int [comm] .
  op _*_ : Int Int -> Int [comm] .
  op _*_ : NzInt NzInt -> NzInt [comm] .
  op quot : Int NzInt -> Int .
  op gcd : NzInt NzInt -> NzNat [comm] .
  vars I J : Int . vars I' J' : NzInt .
  vars N' M' : NzNat .
  eq - - I = I .
  eq - 0 = 0 .
  eq I + 0 = I .
  eq M' + (- M') = 0 .
  ceq M' + (- N') = - d(N',M') if N' > M' .
  ceq M' + (- N') = d(N',M') if M' > N' .
  eq (- I) + (- J) = - (I + J) .
  eq I * 0 = 0 .
  eq I * (- J) = - (I * J) .
  eq quot(0,I') = 0 .
  eq quot(- I',J') = - quot(I',J') .
  eq quot(I',- J') = - quot(I',J') .
  eq gcd(- I',J') = gcd(I',J') .
endfm
```

The corresponding input to the tool is as follows.

```
red checking(
     fmod('INT,
          nilImportList,
          sort(qidSet('Zero, 'NzNat, 'Nat, 'Int, 'NzInt, 'Boolean)),
          subsortDeclSet(
             subsort('Zero, 'Nat), subsort('NzNat, 'Nat),
             subsort('Nat, 'Int), subsort('NzNat, 'NzInt),
             subsort('NzInt, 'Int)),
          opDeclList(
             opDecl('T, nilQidList, 'Boolean, emptyAttrSet),
             opDecl('F, nilQidList, 'Boolean, emptyAttrSet),
             opDecl('0, nilQidList, 'Zero, emptyAttrSet),
             opDecl('s, 'Nat, 'NzNat, emptyAttrSet),
```

```
   opDecl('p, 'NzNat, 'Nat, emptyAttrSet),
   opDecl('+, qidList('Nat, 'Nat), 'Nat, comm),
   opDecl('*, qidList('Nat, 'Nat), 'Nat, comm),
   opDecl('*, qidList('NzNat, 'NzNat), 'NzNat, comm),
   opDecl('greater, qidList('Nat, 'Nat), 'Boolean, emptyAttrSet),
   opDecl('d, qidList('Nat, 'Nat), 'Nat, comm),
   opDecl('quot, qidList('Nat, 'NzNat), 'Nat, emptyAttrSet),
   opDecl('gcd, qidList('NzNat, 'NzNat), 'NzNat, comm),
   opDecl('-, 'Int, 'Int, emptyAttrSet),
   opDecl('-, 'NzInt, 'NzInt, emptyAttrSet),
   opDecl('+, qidList('Int, 'Int), 'Int, comm),
   opDecl('*, qidList('Int, 'Int), 'Int, comm),
   opDecl('*, qidList('NzInt, 'NzInt), 'NzInt, comm),
   opDecl('quot, qidList('Int, 'NzInt), 'Int, emptyAttrSet),
   opDecl('gcd, qidList('NzInt, 'NzInt), 'NzNat, comm)),
varDeclSet(
   varDecl('B, 'Boolean),
   varDecl('N, 'Nat), varDecl('M, 'Nat), varDecl('R, 'Nat),
   varDecl('NN, 'NzNat), varDecl('NM, 'NzNat),
   varDecl('I, 'Int), varDecl('J, 'Int),
   varDecl('NI, 'NzInt), varDecl('NJ, 'NzInt)),
emptyMembAxSet,
equationSet(
   eq('p['s['N]], 'N),
   eq('+['N, '0], 'N ),
   eq('+['s['N], 's['M]], 's['s['+['N, 'M]]]),
   eq('*['N, '0], '0),
   eq('*['s['N], 's['M]], 's['+['+['N, 'M], '*['N, 'M]]]),
   eq('greater['0, 'M], 'F),
   eq('greater['NN, '0], 'T),
   eq('greater['s['N], 's['M]], 'greater['N, 'M]),
   eq('d['0, 'N], 'N),
   eq('d['s['N], 's['M]], 'd['N, 'M]),
   ceq('quot['N, 'NM], 's['quot['d['N,'NM], 'NM]],
       '_==_['greater['N, 'NM], 'T], 'true),
   eq('quot['NM, 'NM], 's['0]),
   ceq('quot['N, 'NM], '0,
       '_==_['greater['NM, 'N], 'T], 'true),
   eq('gcd['NN, 'NN], 'NN),
   ceq('gcd['NN,'NM],
       'gcd['d['NN, 'NM], 'NM],
       '_==_['greater['NN, 'NM], 'T], 'true),
   eq('-['-['I]], 'I),
   eq('-['0], '0),
   eq('+['I, '0], 'I),
   eq('+['NM, '-['NM]], '0),
   ceq('+['NM, '-['NN]], '-['d['NN, 'NM]],
       '_==_['greater['NN, 'NM], 'T], 'true),
   ceq('+['NM, '-['NN]], 'd['NN, 'NM],
       '_==_['greater['NM, 'NN], 'T], 'true),
```

```
          eq('+['-['I], '-['J]], '-['+['I, 'J]]),
          eq('*['I, '0], '0),
          eq('*['I, '-['J]], '-['*['I, 'J]]),
          eq('quot['0, 'NI], '0),
          eq('quot['-['NI], 'NJ], '-['quot['NI, 'NJ]]),
          eq('gcd['-['NI], 'NJ], 'gcd['NI, 'NJ])))) .
```

Here is the output:

```
rewrites: 400567 in 5799ms cpu (5809ms real) (69063 rewrites/second)
result CheckingSolution:
checkingSolution(
   varDeclSet(
      varDecl('NN, 'NzNat), varDecl('NM, 'NzNat),
      varDecl('I, 'Int), varDecl('N@', 'Nat),
      varDecl('NN@', 'NzNat), varDecl('NM@', 'NzNat),
      varDecl('I@', 'Int), varDecl('NI@'@'NzNat, 'NzNat)),
   critPairSet(
      cp('-['+['I@', ('-['I])]], '+['I, ('-['I@'])]),
      ccp('0, 's['0],
          '_==_[('greater['NM@', 'NM@']), 'T], 'true),
      ccp('0, 's['s['0]],
          '_==_[('greater['0, 'NI@'@'NzNat]), 'T], 'true),
      ccp('0, 's['quot[('d['N@', 'NM@']), 'NM@']],
          'and[('_==_[('greater['NM@', 'N@']), 'T]),
               ('_==_[('greater['N@', 'NM@']), 'T])], 'true),
      ccp('0, 'd['NM@', 'NM@'],
          '_==_[('greater['NM@', 'NM@']), 'T], 'true),
      ccp('0, '-['d['NM@', 'NM@']],
          '_==_[('greater['NM@', 'NM@']), 'T], 'true),
      ccp('NN@', 'gcd['NN@', ('d['NN@', 'NN@'])],
          '_==_[('greater['NN@', 'NN@']), 'T], 'true),
      ccp('s['0], 's['quot[('d['NM@', 'NM@']), 'NM@']],
          '_==_[('greater['NM@', 'NM@']), 'T], 'true),
      ccp('d['NN@', 'NM@'], '-['d['NN@', 'NM@']],
          'and[('_==_[('greater['NM@', 'NN@']), 'T]),
               ('_==_[('greater['NN@', 'NM@']), 'T])], 'true),
      ccp('gcd['NM@', ('d['NN@', 'NM@'])],
          'gcd['NN@', ('d['NN@', 'NM@'])],
          'and[('_==_[('greater['NN@', 'NM@']), 'T]),
               ('_==_[('greater['NM@', 'NN@']), 'T])], 'true)),
   cmb('gcd[('d['NN, 'NM]), 'NM], 'NzNat,
       '_==_[('greater['NN, 'NM]), 'T], 'true))
```

We have some critical pairs, among which only one is non conditional, and one proof obligation for sort decreasingness. Let us study them.

None of the conditions in the conditional critical pairs can be satisfied, so—in an intuitive sense, but with the remaining proof obligation of showing that, indeed, those conditions are unsatisfiable—we only need to worry about the non-conditional critical pair.

```
cp('-['+['I@', ('-['I])]], '+['I, ('-['I@'])]),
```

A classical completion systems would add the equation

```
eq('-['+['I, ('-['J])]], '+['J, ('-['I])])
```

But this would introduce more critical pairs, and the process would have to be repeated several times without any guarantee of success or even of termination. But looking at the critical pair we can see easily that the problem is that there is no way to reduce a sum where one of the arguments is a negative number. And if we try to proceed adding the critical pair oriented, we obtain critical pairs in which the same situation appears. The question is that a term of the form `I + (-J)` cannot be reduced if `J` is of sort `Int`. But if `J` is a non-zero natural number then some equation could be applied, and if it is lesser than zero then it is of the form `-NN` for some `NN` of sort `NzNat`. But then `-(-NN)` can be reduced to `NN`. The only thing we can say then is that probably the specification is confluent, although nothing can be concluded from these critical pairs. But we can in fact modify the specification not just to reduce the critical pairs but in fact to avoid generating them. This can be easily done, just by changing in the equations the terms of the form `-I` by others of the form `-NN`. Making such modifications we then get:

```
red checking(
     fmod('INT,
          nilImportList,
          sort(qidSet('Zero, 'NzNat, 'Nat, 'Int, 'NzInt, 'Boolean)),
          subsortDeclSet(
             subsort('Zero, 'Nat), subsort('NzNat, 'Nat),
             subsort('Nat, 'Int), subsort('NzNat, 'NzInt),
             subsort('NzInt, 'Int)),
          opDeclList(
             opDecl('T, nilQidList, 'Boolean, emptyAttrSet),
             opDecl('F, nilQidList, 'Boolean, emptyAttrSet),
             opDecl('0, nilQidList, 'Zero, emptyAttrSet),
             opDecl('s, 'Nat, 'NzNat, emptyAttrSet),
             opDecl('p, 'NzNat, 'Nat, emptyAttrSet),
             opDecl('+, qidList('Nat, 'Nat), 'Nat, comm),
             opDecl('*, qidList('Nat, 'Nat), 'Nat, comm),
             opDecl('*, qidList('NzNat, 'NzNat), 'NzNat, comm),
             opDecl('greater, qidList('Nat, 'Nat), 'Boolean, emptyAttrSet),
             opDecl('d, qidList('Nat, 'Nat), 'Nat, comm),
             opDecl('quot, qidList('Nat, 'NzNat), 'Nat, emptyAttrSet),
             opDecl('gcd, qidList('NzNat, 'NzNat), 'NzNat, comm),
             opDecl('-, 'Int, 'Int, emptyAttrSet),
             opDecl('-, 'NzInt, 'NzInt, emptyAttrSet),
             opDecl('+, qidList('Int, 'Int), 'Int, comm),
             opDecl('*, qidList('Int, 'Int), 'Int, comm),
             opDecl('*, qidList('NzInt, 'NzInt), 'NzInt, comm),
             opDecl('quot, qidList('Int, 'NzInt), 'Int, emptyAttrSet),
             opDecl('gcd, qidList('NzInt, 'NzInt), 'NzNat, comm)),
          varDeclSet(
             varDecl('B, 'Boolean),
```

```
          varDecl('N, 'Nat), varDecl('M, 'Nat), varDecl('R, 'Nat),
          varDecl('NN, 'NzNat), varDecl('NM, 'NzNat),
          varDecl('I, 'Int), varDecl('J, 'Int),
          varDecl('NI, 'NzInt), varDecl('NJ, 'NzInt)),
       emptyMembAxSet,
       equationSet(
          eq('p['s['N]], 'N),
          eq('+['N, '0], 'N ),
          eq('+['s['N], 's['M]], 's['s['+['N, 'M]]]),
          eq('*['N, '0], '0),
          eq('*['s['N], 's['M]], 's['+['+['N, 'M], '*['N, 'M]]]),
          eq('greater['0, 'M], 'F),
          eq('greater['NN, '0], 'T),
          eq('greater['s['N], 's['M]], 'greater['N, 'M]),
          eq('d['0, 'N], 'N),
          eq('d['s['N], 's['M]], 'd['N, 'M]),
          ceq('quot['N, 'NM], 's['quot['d['N,'NM], 'NM]],
              '_==_['greater['N, 'NM], 'T], 'true),
          eq('quot['NM, 'NM], 's['0]),
          ceq('quot['N, 'NM], '0, '_==_['greater['NM, 'N], 'T], 'true),
          eq('gcd['NN, 'NN], 'NN),
          ceq('gcd['NN,'NM], 'gcd['d['NN, 'NM], 'NM],
              '_==_['greater['NN, 'NM], 'T], 'true),
          eq('-['-['I]], 'I),
          eq('-['0], '0),
          eq('+['I, '0], 'I),
          eq('+['NM, '-['NM]], '0),
          ceq('+['NM, '-['NN]], '-['d['NN, 'NM]],
              '_==_['greater['NN, 'NM], 'T], 'true),
          ceq('+['NM, '-['NN]], 'd['NN, 'NM],
              '_==_['greater['NM, 'NN], 'T], 'true),
          eq('+['-['NN], '-['NM]], '-['+['NN, 'NM]]),
          eq('*['I, '0], '0),
          eq('*['I, '-['NN]], '-['*['I, 'NN]]),
          eq('quot['0, 'NI], '0),
          eq('quot['-['NN], 'NI], '-['quot['NN, 'NI]]),
          eq('gcd['-['NI], 'NJ], 'gcd['NI, 'NJ])))) .

rewrites: 391368 in 5689ms cpu (5699ms real) (68781 rewrites/second)
result CheckingSolution:
checkingSolution(
   varDeclSet(
      varDecl('NN, 'NzNat), varDecl('NM, 'NzNat),
      varDecl('N@', 'Nat), varDecl('NN@', 'NzNat),
      varDecl('NM@', 'NzNat), varDecl('NI@'@'NzNat, 'NzNat)),
   critPairSet(
      ccp('0, 's['0],
          '_==_[('greater['NM@', 'NM@']), 'T], 'true),
      ccp('0, 's['s['0]],
          '_==_[('greater['0, 'NI@'@'NzNat]), 'T], 'true),
```

```
    ccp('0, 's['quot[('d['N@', 'NM@']), 'NM@']],
        'and[('_==_[('greater['NM@', 'N@']), 'T]),
             ('_==_[('greater['N@', 'NM@']), 'T])], 'true),
    ccp('0, 'd['NM@', 'NM@'],
        '_==_[('greater['NM@', 'NM@']), 'T], 'true),
    ccp('0, '-['d['NM@', 'NM@']],
        '_==_[('greater['NM@', 'NM@']), 'T], 'true),
    ccp('NN@', 'gcd['NN@', ('d['NN@', 'NN@'])],
        '_==_[('greater['NN@', 'NN@']), 'T], 'true),
    ccp('s['0], 's['quot[('d['NM@', 'NM@']), 'NM@']],
        '_==_[('greater['NM@', 'NM@']), 'T], 'true),
    ccp('d['NN@', 'NM@'], '-['d['NN@', 'NM@']],
        'and[('_==_[('greater['NM@', 'NN@']), 'T]),
             ('_==_[('greater['NN@', 'NM@']), 'T])], 'true),
    ccp('gcd['NM@', ('d['NN@', 'NM@'])],
        'gcd['NN@', ('d['NN@', 'NM@'])],
        'and[('_==_[('greater['NN@', 'NM@']), 'T]),
             ('_==_[('greater['NM@', 'NN@']), 'T])], 'true)),
 cmb('gcd[('d['NN, 'NM]), 'NM], 'NzNat,
     '_==_[('greater['NN, 'NM]), 'T], 'true))
```

All the critical pairs obtained are now conditional, with their conditions unsatisfiable. Therefore, modulo checking termination and the unsatisfiability of the conditions, we can conclude that it is confluent.

The conditional membership proof obligation is the same one we had for the specification of the natural numbers in Section 5.6.1.

5.6.3 Rationals

We give now a specification for the rationals, also similar to that presented in the manual of OBJ ([10, page 78]). We assume the module `INT` as resulted from the modifications introduced there to make the specification Church-Rosser.

```
fmod RAT is
  sorts Rat NzRat .
  protecting INT .
  subsort Int < Rat .
  subsorts NzInt < NzRat < Rat .
  op _/_ : Rat NzRat -> Rat .
  op _/_ : NzRat NzRat -> NzRat .
  op -_  : Rat -> Rat .
  op -_  : NzRat -> NzRat .
  op _+_ : Rat Rat -> Rat [comm] .
  op _*_ : Rat Rat -> Rat [comm] .
  op _*_ : NzRat NzRat -> NzRat [comm] .
  vars I' J' : NzInt . vars R S : Rat .
  vars R' S' : NzRat .
  eq R / (R' / S') = (R * S') / R' .
  eq (R / R') / S' = R / (R' * S') .
  cq J' / I'
    = quot(J', gcd(J', I')) / quot(I', gcd(J', I'))
```

```
      if gcd(J', I') > s 0 .
  eq R / s 0 = R .
  eq 0 / R' = 0 .
  eq R / (- R') = (- R) / R' .
  eq - (R / R') = (- R) / R' .
  eq R + (S / R') = ((R * R') + S) / R' .
  eq R * (S / R') = (R * S) / R' .
endo
```

Its meta-representation:

```
red checking(
     fmod('RAT,
          nilImportList,
          sort(qidSet('Zero, 'NzNat, 'Nat, 'Int,
                      'NzInt, 'Rat, 'NzRat, 'Boolean)),
          subsortDeclSet(
             subsort('Zero, 'Nat), subsort('NzNat, 'Nat),
             subsort('Nat, 'Int), subsort('NzNat, 'NzInt),
             subsort('NzInt, 'Int), subsort('Int, 'Rat),
             subsort('NzInt, 'NzRat), subsort('NzRat, 'Rat)),
          opDeclList(
             opDecl('T, nilQidList, 'Boolean, emptyAttrSet),
             opDecl('F, nilQidList, 'Boolean, emptyAttrSet),
             opDecl('0, nilQidList, 'Zero, emptyAttrSet),
             opDecl('s, 'Nat, 'NzNat, emptyAttrSet),
             opDecl('p, 'NzNat, 'Nat, emptyAttrSet),
             opDecl('+, qidList('Nat, 'Nat), 'Nat, comm),
             opDecl('*, qidList('Nat, 'Nat), 'Nat, comm),
             opDecl('*, qidList('NzNat, 'NzNat), 'NzNat, comm),
             opDecl('greater, qidList('Nat, 'Nat), 'Boolean, emptyAttrSet),
             opDecl('d, qidList('Nat, 'Nat), 'Nat, comm),
             opDecl('quot, qidList('Nat, 'NzNat), 'Nat, emptyAttrSet),
             opDecl('gcd, qidList('NzNat, 'NzNat), 'NzNat, comm),
             opDecl('-, 'Int, 'Int, emptyAttrSet),
             opDecl('-, 'NzInt, 'NzInt, emptyAttrSet),
             opDecl('+, qidList('Int, 'Int), 'Int, comm),
             opDecl('*, qidList('Int, 'Int), 'Int, comm),
             opDecl('*, qidList('NzInt, 'NzInt), 'NzInt, comm),
             opDecl('quot, qidList('Int, 'NzInt), 'Int, emptyAttrSet),
             opDecl('gcd, qidList('NzInt, 'NzInt), 'NzNat, comm),
             opDecl('/, qidList('Rat, 'NzRat), 'Rat, emptyAttrSet),
             opDecl('/, qidList('NzRat, 'NzRat), 'NzRat, emptyAttrSet),
             opDecl('-, 'Rat, 'Rat, emptyAttrSet),
             opDecl('-, 'NzRat, 'NzRat, emptyAttrSet),
             opDecl('+, qidList('Rat, 'Rat), 'Rat, comm),
             opDecl('*, qidList('Rat, 'Rat), 'Rat, comm),
             opDecl('*, qidList('NzRat, 'NzRat), 'NzRat, comm)),
          varDeclSet(
             varDecl('B, 'Boolean),
```

```
   varDecl('N, 'Nat), varDecl('M, 'Nat),
   varDecl('NN, 'NzNat), varDecl('NM, 'NzNat),
   varDecl('I, 'Int), varDecl('J, 'Int),
   varDecl('NI, 'NzInt), varDecl('NJ, 'NzInt),
   varDecl('R, 'Rat), varDecl('S, 'Rat),
   varDecl('R', 'NzRat), varDecl('S', 'NzRat)),
emptyMembAxSet,
equationSet(
   eq('p['s['N]], 'N),
   eq('+['N, '0], 'N ),
   eq('+['s['N], 's['M]], 's['s['+['N, 'M]]]),
   eq('*['N, '0], '0),
   eq('*['s['N], 's['M]], 's['+['+['N, 'M], '*['N, 'M]]]),
   eq('greater['0, 'M], 'F),
   eq('greater['NN, '0], 'T),
   eq('greater['s['N], 's['M]], 'greater['N, 'M]),
   eq('d['0, 'N], 'N),
   eq('d['s['N], 's['M]], 'd['N, 'M]),
   ceq('quot['N, 'NM], 's['quot['d['N,'NM], 'NM]],
       '_==_['greater['N, 'NM], 'T], 'true),
   eq('quot['NM, 'NM], 's['0]),
   ceq('quot['N, 'NM], '0, '_==_['greater['NM, 'N], 'T], 'true),
   eq('gcd['NN, 'NN], 'NN),
   ceq('gcd['NN,'NM], 'gcd['d['NN, 'NM], 'NM],
       '_==_['greater['NN, 'NM], 'T], 'true),
   eq('-['-['I]], 'I),
   eq('-['0], '0),
   eq('+['I, '0], 'I),
   eq('+['NM, '-['NM]], '0),
   ceq('+['NM, '-['NN]], '-['d['NN, 'NM]],
       '_==_['greater['NN, 'NM], 'T], 'true),
   ceq('+['NM, '-['NN]], 'd['NN, 'NM],
       '_==_['greater['NM, 'NN], 'T], 'true),
   eq('+['-['NN], '-['NM]], '-['+['NN, 'NM]]),
   eq('*['I, '0], '0),
   eq('*['I, '-['NN]], '-['*['I, 'NN]]),
   eq('quot['0, 'NI], '0),
   eq('quot['-['NN], 'NI], '-['quot['NN, 'NI]]),
   eq('gcd['-['NI], 'NJ], 'gcd['NI, 'NJ]),
   eq('/['R, '/['R', 'S']], '/['*['R, 'S'], 'R']),
   eq('/['/['R, 'R'], 'S'], '/['R, '*['R', 'S']]),
   ceq('/['NJ, 'NI],
       '/['quot['NJ, 'gcd['NJ, 'NI]],
          'quot['NI, 'gcd['NJ, 'NI]]],
       '_==_['greater['gcd['NJ, 'NI], 's['0]], 'T], 'true),
   eq('/['R, 's['0]], 'R),
   eq('/['0, 'R'], '0),
   eq('/['R, '-['R']], '/['-['R], 'R']),
   eq('-['/['R, 'R']], '/['-['R], 'R']),
   eq('+['R, '/['S, 'R']], '/['+['*['R, 'R'], 'S], 'R']),
```

```
          eq('*['R, '/['S, 'R']], '/['*['R, 'S], 'R'])))) .
```

Yields the result

```
rewrites: 2156488 in 30919ms cpu (30929ms real) (69744 rewrites/second)
result CheckingSolution:
checkingSolution(
   varDeclSet(
      varDecl('S', 'NzRat), varDecl('NN, 'NzNat), varDecl('NM, 'NzNat),
      varDecl('NI, 'NzInt), varDecl('NJ, 'NzInt), varDecl('R, 'Rat),
      varDecl('R', 'NzRat), varDecl('N@', 'Nat), varDecl('S@', 'Rat),
      varDecl('S'@', 'NzRat), varDecl('NN@', 'NzNat),
      varDecl('NM@', 'NzNat), varDecl('NJ@', 'NzInt),
      varDecl('R@', 'Rat), varDecl('R'@', 'NzRat),
      varDecl('R'@'NzInt, 'NzInt), varDecl('R'@'@'NzInt, 'NzInt),
      varDecl('R@'@'NzInt, 'NzInt), varDecl('NJ@'NzNat, 'NzNat),
      varDecl('NI@'NzNat, 'NzNat), varDecl('NI@'@'NzNat, 'NzNat),
      varDecl('I@'NzInt, 'NzInt), varDecl('S'@'@'NzInt, 'NzInt),
      varDecl('S@'@'NzInt, 'NzInt)),
   critPairSet(
1     cp('0, '/[('*['0, 'S']), 'R']),
2     cp('+['S@', ('*['R@', ('s['0])])], '+['S@', 'R@']),
3     cp('/['R, 'R'], '/['R, ('*['R', ('s['0])])]),
4     cp('/['R, ('*['R', ('*['S'@', 'R'@'])])],
         '/['R, ('*['S'@', ('*['R', 'R'@'])])]),
5     cp('/['R@', ('*['S'@', ('-['R'])])],
         '/[('-['R@']), ('*['R', 'S'@'])]),
6     cp('/[('+['0, ('*['R@', 'R'@'])]), 'R'@'], '+['0, 'R@']),
7     cp('/[('+['R, ('*['R', ('*['R@', 'R'@'])])]), ('*['R', 'R'@'])],
         '/[('+['R, ('*['R@', ('*['R', 'R'@'])])]), ('*['R', 'R'@'])]),
8     cp('/[('*['0, 'R@']), 'R'@'], '*['0, 'R@']),
9     cp('/[('*['S', ('+[('*['S', 'S@']), ('*['R', 'R@'])])]),
            ('*['S', 'R'])],
         '/[('+[('*['S', 'S@']), ('*['R', 'R@'])]), 'R']),
10    cp('/[('*['S', ('*['S@', 'R@'])]), 'R'],
         '/[('*['R@', ('*['S', 'S@'])]), 'R']),
11    cp('/[('*['R@', ('s['0])]), 'R'@'], '/['R@', 'R'@']),
12    cp('/[('-['+['S@', ('*['R@', ('-['R'])])]]), 'R'],
         '/[('+[('-['S@']), ('*['R', 'R@'])]), 'R']),
13    cp('/[('-['*['S@', 'R@']]), 'R'], '/[('*['R@', ('-['S@'])]), 'R']),
14    cp('/[('-['-['R@']]), 'I@'NzInt], '/['R@', 'I@'NzInt]),
15    ccp('0, 's['0],
         '_==_[('greater['NM@', 'NM@']), 'T], 'true),
16    ccp('0, 's['s['0]],
         '_==_[('greater['0, 'NI@'@'NzNat]), 'T], 'true),
17    ccp('0, 's['quot[('d['N@', 'NM@']), 'NM@']],
         'and[('_==_[('greater['NM@', 'N@']), 'T]),
              ('_==_[('greater['N@', 'NM@']), 'T])], 'true),
18    ccp('0, 'd['NM@', 'NM@'],
         '_==_[('greater['NM@', 'NM@']), 'T], 'true),
```

```
19    ccp('0, '-['d['NM@', 'NM@']],
          '_==_[('greater['NM@', 'NM@']), 'T], 'true),
20    ccp('NN@', 'gcd['NN@', ('d['NN@', 'NN@'])],
          '_==_[('greater['NN@', 'NN@']), 'T], 'true),
21    ccp('R@'@'NzInt,
          '/[('quot['R@'@'NzInt, ('gcd['R@'@'NzInt, ('s['0])])]),
            ('quot[('s['0]), ('gcd['R@'@'NzInt, ('s['0])])])],
          '_==_[('greater[('gcd['R@'@'NzInt, ('s['0])]), ('s['0])]), 'T],
          'true),
22    ccp('s['0], 's['quot[('d['NM@', 'NM@']), 'NM@']],
          '_==_[('greater['NM@', 'NM@']), 'T], 'true),
23    ccp('d['NN@', 'NM@'], '-['d['NN@', 'NM@']],
          'and[('_==_[('greater['NM@', 'NN@']), 'T]),
               ('_==_[('greater['NN@', 'NM@']), 'T])],
          'true),
24    ccp('gcd['NM@', ('d['NN@', 'NM@'])], 'gcd['NN@', ('d['NN@', 'NM@'])],
          'and[('_==_[('greater['NN@', 'NM@']), 'T]),
               ('_==_[('greater['NM@', 'NN@']), 'T])],
          'true),
25    ccp('/['R@'@'NzInt, ('*['S'@', 'R'@'@'NzInt])],
          '/[('/[('quot['R@'@'NzInt, ('gcd['R'@'@'NzInt, 'R@'@'NzInt])]),
                 ('quot['R'@'@'NzInt, ('gcd['R'@'@'NzInt, 'R@'@'NzInt])])]),
            'S'@'],
          '_==_[('greater[('gcd['R@'@'NzInt, 'R'@'@'NzInt]), ('s['0])]), 'T],
          'true),
26    ccp('/[('+['S@'@'NzInt, ('*['R@', 'R'@'@'NzInt])]),
            'R'@'@'NzInt],
          '+['R@',
            ('/[('quot['S@'@'NzInt, ('gcd['R'@'@'NzInt, 'S@'@'NzInt])]),
                ('quot['R'@'@'NzInt, ('gcd['R'@'@'NzInt, 'S@'@'NzInt])])])],
          '_==_[('greater[('gcd['S@'@'NzInt, 'R'@'@'NzInt]), ('s['0])]), 'T],
          'true),
27    ccp('/[('*['R@', 'S'@'@'NzInt]), 'R'@'@'NzInt],
          '/['R@',
            ('/[('quot['R'@'@'NzInt, ('gcd['R'@'@'NzInt, 'S'@'@'NzInt])]),
                ('quot['S'@'@'NzInt, ('gcd['R'@'@'NzInt, 'S'@'@'NzInt])])])],
          '_==_[('greater[('gcd['R'@'@'NzInt, 'S'@'@'NzInt]), ('s['0])]), 'T],
          'true),
28    ccp('/[('*['R@', 'S@'@'NzInt]), 'R'@'@'NzInt],
          '*['R@',
            ('/[('quot['S@'@'NzInt, ('gcd['R'@'@'NzInt, 'S@'@'NzInt])]),
                ('quot['R'@'@'NzInt, ('gcd['R'@'@'NzInt, 'S@'@'NzInt])])])],
          '_==_[('greater[('gcd['S@'@'NzInt, 'R'@'@'NzInt]), ('s['0])]), 'T],
          'true),
29    ccp('/[('quot['NJ@', ('gcd['NJ@', 'R'@'NzInt])]),
            ('quot[('-['R'@'NzInt]), ('gcd['NJ@', 'R'@'NzInt])])],
          '/[('-['NJ@']), 'R'@'NzInt],
          '_==_[('greater[('gcd['NJ@', ('-['R'@'NzInt])]), ('s['0])]), 'T],
          'true),
30    ccp('/[('-['R@'@'NzInt]), 'R'@'@'NzInt],
```

```
        '-['/[('quot['R@'@'NzInt, ('gcd['R'@'@'NzInt, 'R@'@'NzInt])]),
             ('quot['R'@'@'NzInt, ('gcd['R'@'@'NzInt, 'R@'@'NzInt])])]],
        '_==_[('greater[('gcd['R@'@'NzInt, 'R'@'@'NzInt]), ('s['0])]), 'T],
        'true)),
  membAxSet(
1     cmb('gcd[('d['NN, 'NM]), 'NM], 'NzNat,
        '_==_[('greater['NN, 'NM]), 'T], 'true),
2     cmb('/[('quot['NJ, ('gcd['NJ, 'NI])]),
            ('quot['NI, ('gcd['NJ, 'NI])])],
        'NzRat,
        '_==_[('greater[('gcd['NJ, 'NI]), ('s['0])]), 'T],
        'true),
3     cmb('/[('quot['NJ, ('gcd['NJ, 'NI@'NzNat])]),
            ('quot['NI@'NzNat, ('gcd['NJ, 'NI@'NzNat])])],
        'NzRat,
        '_==_[('greater[('gcd['NJ, 'NI@'NzNat]), ('s['0])]), 'T],
        'true),
4     cmb('/[('quot['NJ@'NzNat, ('gcd['NJ@'NzNat, 'NI])]),
            ('quot['NI, ('gcd['NJ@'NzNat, 'NI])])], 'NzRat,
        '_==_[('greater[('gcd['NJ@'NzNat, 'NI]), ('s['0])]), 'T],
        'true),
5     cmb('/[('quot['NJ@'NzNat, ('gcd['NJ@'NzNat, 'NI@'NzNat])]),
            ('quot['NI@'NzNat, ('gcd['NJ@'NzNat, 'NI@'NzNat])])],
        'NzRat,
        '_==_[('greater[('gcd['NJ@'NzNat, 'NI@'NzNat]), ('s['0])]), 'T],
        'true)))
```

Studing in full detail this specification to make it Church-Rosser is a challenging problem. We are not going to do it here; we just present a general discussion on the methods that can be used for this purpose.

As you can see, we have numbered the critical pairs and the membership proof obligations given by the Church-Rosser Checker to be able to reference them more easily. Among these we have, for example, that the conditional critical pairs 15–24 have non-satisfiable conditions, and so, modulo checking for the unsatisfiability of the conditions, we do not have to worry about them. From 1 and 8 we see that terms of the form `R * 0` cannot be further reduced. The same happens with terms of forms `s(0) * R` and `0 + R`, as can be observed in critical pairs 2, 3 and 11, and 1 and 8, respectively. We can eliminate these critical pairs adding the following equations:

```
eq('*['R, '0], '0)
eq('*['R, 's['0]], 'R)
eq('+['R, '0], 'R)
```

From critical pairs 5 and 14 we also realize the need for the following equations:

```
eq('*['R, '-['S]], '-['*['R, 'S]])
eq('-['-['R]], 'R)
```

After introducing these equations there are still 22 critical pairs and 5 memberships left:

```
rewrites: 1770682 in 38999ms cpu (39009ms real) (45402 rewrites/second)
result CheckingSolution:
checkingSolution(
   varDeclSet(
      varDecl('S', 'NzRat), varDecl('NN, 'NzNat), varDecl('NM, 'NzNat),
      varDecl('NI, 'NzInt), varDecl('NJ, 'NzInt), varDecl('R, 'Rat),
      varDecl('R', 'NzRat), varDecl('N@', 'Nat), varDecl('S@', 'Rat),
      varDecl('S'@', 'NzRat), varDecl('NN@', 'NzNat), varDecl('NM@', 'NzNat),
      varDecl('NJ@', 'NzInt), varDecl('R@', 'Rat), varDecl('R'@', 'NzRat),
      varDecl('R'@'NzInt, 'NzInt), varDecl('R'@'@'NzInt, 'NzInt),
      varDecl('R@'@'NzInt, 'NzInt), varDecl('NJ@'NzNat, 'NzNat),
      varDecl('NI@'NzNat, 'NzNat), varDecl('NI@'@'NzNat, 'NzNat),
      varDecl('S'@'@'NzInt, 'NzInt), varDecl('S@'@'NzInt, 'NzInt)),
   critPairSet(
1     cp('/['R, ('*['R', ('*['S'@', 'R'@'])])],
         '/['R, ('*['S'@', ('*['R', 'R'@'])])]),
2     cp('/[('+['R, ('*['R', ('*['R@', 'R'@'])])]), ('*['R', 'R'@'])],
         '/[('+['R, ('*['R@', ('*['R', 'R'@'])])]), ('*['R', 'R'@'])]),
3     cp('/[('*['S', ('+[('*['S', 'S@']), ('*['R', 'R@'])])]),
            ('*['S', 'R'])],
         '/[('+[('*['S', 'S@']), ('*['R', 'R@'])]), 'R']),
4     cp('/[('*['S', ('*['S@', 'R@'])]), 'R'],
         '/[('*['R@', ('*['S', 'S@'])]), 'R']),
5     cp('/[('*['R@', 'R'@']), 'R'@'], 'R@'),
6     cp('/[('-['+['S@', ('-['*['R', 'R@']])]]), 'R'],
         '/[('+[('-['S@']), ('*['R', 'R@'])]), 'R']),
7     ccp('0, 's['0], '_==_[('greater['NM@', 'NM@']), 'T], 'true),
8     ccp('0, 's['s['0]], '_==_[('greater['0, 'NI@'@'NzNat]), 'T], 'true),
9     ccp('0, 's['quot[('d['N@', 'NM@']), 'NM@']],
          'and[('_==_[('greater['NM@', 'N@']), 'T]),
               ('_==_[('greater['N@', 'NM@']), 'T])], 'true),
10    ccp('0, 'd['NM@', 'NM@'],
          '_==_[('greater['NM@', 'NM@']), 'T], 'true),
11    ccp('0, '-['d['NM@', 'NM@']],
          '_==_[('greater['NM@', 'NM@']), 'T], 'true),
12    ccp('NN@', 'gcd['NN@', ('d['NN@', 'NN@'])],
          '_==_[('greater['NN@', 'NN@']), 'T], 'true),
13    ccp('R@'@'NzInt,
          '/[('quot['R@'@'NzInt, ('gcd['R@'@'NzInt, ('s['0])])]),
             ('quot[('s['0]), ('gcd['R@'@'NzInt, ('s['0])])])],
          '_==_[('greater[('gcd['R@'@'NzInt, ('s['0])]), ('s['0])]), 'T],
          'true),
14    ccp('s['0], 's['quot[('d['NM@', 'NM@']), 'NM@']],
          '_==_[('greater['NM@', 'NM@']), 'T], 'true),
15    ccp('d['NN@', 'NM@'], '-['d['NN@', 'NM@']],
          'and[('_==_[('greater['NM@', 'NN@']), 'T]),
               ('_==_[('greater['NN@', 'NM@']), 'T])], 'true),
16    ccp('gcd['NM@', ('d['NN@', 'NM@'])], 'gcd['NN@', ('d['NN@', 'NM@'])],
          'and[('_==_[('greater['NN@', 'NM@']), 'T]),
               ('_==_[('greater['NM@', 'NN@']), 'T])], 'true),
```

```
17    ccp('/['R@'@'NzInt, ('*['S'@', 'R'@'@'NzInt])],
          '/[('/[('quot['R@'@'NzInt, ('gcd['R'@'@'NzInt, 'R@'@'NzInt])]),
                ('quot['R'@'@'NzInt, ('gcd['R'@'@'NzInt, 'R@'@'NzInt])])]),
            'S'@'],
          '_==_[('greater[('gcd['R@'@'NzInt, 'R'@'@'NzInt]), ('s['0])]), 'T],
          'true),
18    ccp('/[('+['S@'@'NzInt, ('*['R@', 'R'@'@'NzInt])]), 'R'@'@'NzInt],
          '+['R@',
            ('/[('quot['S@'@'NzInt, ('gcd['R'@'@'NzInt, 'S@'@'NzInt])]),
                ('quot['R'@'@'NzInt, ('gcd['R'@'@'NzInt, 'S@'@'NzInt])])])],
          '_==_[('greater[('gcd['S@'@'NzInt, 'R'@'@'NzInt]), ('s['0])]), 'T],
          'true),
19    ccp('/[('*['R@', 'S'@'@'NzInt]), 'R'@'@'NzInt],
          '/['R@',
            ('/[('quot['R'@'@'NzInt, ('gcd['R'@'@'NzInt, 'S'@'@'NzInt])]),
                ('quot['S'@'@'NzInt, ('gcd['R'@'@'NzInt, 'S'@'@'NzInt])])])],
          '_==_[('greater[('gcd['R'@'@'NzInt, 'S'@'@'NzInt]), ('s['0])]), 'T],
          'true),
20    ccp('/[('*['R@', 'S@'@'NzInt]), 'R'@'@'NzInt],
          '*['R@',
            ('/[('quot['S@'@'NzInt, ('gcd['R'@'@'NzInt, 'S@'@'NzInt])]),
                ('quot['R'@'@'NzInt, ('gcd['R'@'@'NzInt, 'S@'@'NzInt])])])],
          '_==_[('greater[('gcd['S@'@'NzInt, 'R'@'@'NzInt]), ('s['0])]), 'T],
          'true),
21    ccp('/[('quot['NJ@', ('gcd['NJ@', 'R'@'NzInt])]),
            ('quot[('-['R'@'NzInt]), ('gcd['NJ@', 'R'@'NzInt])])],
          '/[('-['NJ@']), 'R'@'NzInt],
          '_==_[('greater[('gcd['NJ@', ('-['R'@'NzInt])]), ('s['0])]), 'T],
          'true),
22    ccp('/[('-['R@'@'NzInt]), 'R'@'@'NzInt],
          '-['/[('quot['R@'@'NzInt, ('gcd['R'@'@'NzInt, 'R@'@'NzInt])]),
               ('quot['R'@'@'NzInt, ('gcd['R'@'@'NzInt, 'R@'@'NzInt])])]],
          '_==_[('greater[('gcd['R@'@'NzInt, 'R'@'@'NzInt]), ('s['0])]), 'T],
          'true)),
  membAxSet(
1     cmb('gcd[('d['NN, 'NM]), 'NM], 'NzNat,
          '_==_[('greater['NN, 'NM]), 'T], 'true),
2     cmb('/[('quot['NJ, ('gcd['NJ, 'NI])]),
            ('quot['NI, ('gcd['NJ, 'NI])])],
          'NzRat,
          '_==_[('greater[('gcd['NJ, 'NI]), ('s['0])]), 'T],
          'true),
3     cmb('/[('quot['NJ, ('gcd['NJ, 'NI@'NzNat])]),
            ('quot['NI@'NzNat, ('gcd['NJ, 'NI@'NzNat])])],
          'NzRat,
          '_==_[('greater[('gcd['NJ, 'NI@'NzNat]), ('s['0])]), 'T],
          'true),
4     cmb('/[('quot['NJ@'NzNat, ('gcd['NJ@'NzNat, 'NI])]),
            ('quot['NI, ('gcd['NJ@'NzNat, 'NI])])],
          'NzRat,
```

```
        '_==_[('greater[('gcd['NJ@'NzNat, 'NI]), ('s['0])]), 'T],
        'true),
    cmb('/[('quot['NJ@'NzNat, ('gcd['NJ@'NzNat, 'NI@'NzNat])]),
          ('quot['NI@'NzNat, ('gcd['NJ@'NzNat, 'NI@'NzNat])])],
        'NzRat,
        '_==_[('greater[('gcd['NJ@'NzNat, 'NI@'NzNat]), ('s['0])]), 'T],
        'true)))
```

Note that some of the critical pairs have been removed, but others may have been generated, or some of the critical pairs that already appeared may be simplified to a different form. Continuing with this process we can make the specification confluent, with changes completely intuitive and clear, decided by the user himself/herself. Although it can be a somewhat tedious we think that the advantages are clear.

Regarding sort-decreasingness, the proof obligations have also to be proved.

5.6.4 Array

We present here a specification for the abstract data type `ARRAY`, where both the data and the indices are natural numbers.

```
fmod ARRAY is
  sorts Nat Nat? Array .
  subsorts Nat < Nat? .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op equal : Nat Nat -> Bool [comm] .
  op nilArr : -> Array .
  op put : Nat Nat -> Array .
  op _[_] : Array Nat -> Nat? .
  op _in_ : Nat Array -> Bool .
  op undef : Nat -> Nat? . *** err-op
  var A : Array .
  vars I I' E : Nat .
  eq equal(0, 0) = true .
  eq equal(s(I), s(I')) = equal(I,I') .
  eq equal(s(I), 0) = false .
  eq put(I, E, A)[ I ] = E .
  ceq put(I, E, A)[ I' ] = A [ I' ] if equal(I, I') == false .
  eq I in nilArr = false .
  eq I in put(I, E, A) = true .
  ceq I in put(I', E, A) = I in A if equal(I, I') == false .
  ceq A [ I ] = undef(I) if not I in A . *** err-eqn
endfm
```

After some changes to deal with quotes in the variables and mix-fix syntax, and translating it to its meta-representation we get the following input to the tool.

```
red checking(
  fmod(
     'ARRAY,
     nilImportList,
```

```
    sort(qidSet('Nat, 'Nat?, 'Array, 'Boolean)),
    subsort('Nat, 'Nat?),
    opDeclList(
       opDecl('T, nilQidList, 'Boolean, emptyAttrSet),
       opDecl('F, nilQidList, 'Boolean, emptyAttrSet),
       opDecl('0, nilQidList, 'Nat, emptyAttrSet),
       opDecl('s, 'Nat, 'Nat, emptyAttrSet),
       opDecl('equal, qidList('Nat, 'Nat), 'Boolean, comm),
       opDecl('nilArr, nilQidList, 'Array, emptyAttrSet),
       opDecl('put, qidList('Nat, 'Nat, 'Array), 'Array,
              emptyAttrSet),
       opDecl('at, qidList('Array, 'Nat), 'Nat?, emptyAttrSet),
       opDecl('in, qidList('Nat, 'Array), 'Boolean,
              emptyAttrSet),
       opDecl('undef, 'Nat, 'Nat?, emptyAttrSet)),
    varDeclSet(
       varDecl('B, 'Boolean), varDecl('A, 'Array),
       varDecl('N, 'Nat), varDecl('M, 'Nat),
       varDecl('E, 'Nat)),
    emptyMembAxSet,
    equationSet(
       eq('equal['0, '0], 'T),
       eq('equal['s['N], 's['M]], 'equal['N, 'M]),
       eq('equal['s['N], '0], 'F),
       eq('at['put['N, 'E, 'A], 'N], 'E),
       ceq('at['put['N, 'E, 'A], 'M],
           'at['A, 'M], '_==_['equal['N, 'M], 'F], 'true),
       eq('in['N, 'nilArr], 'F),
       eq('in['N, 'put['N, 'E, 'A]], 'T),
       ceq('in['N, 'put['M, 'E, 'A]], 'in['N, 'A],
           '_==_['equal['N, 'M], 'F], 'true),
       ceq('at['A, 'N],
           'undef['N], '_==_['in['N, 'A], 'F], 'true)))) .
```

The output given by the tool is:

```
rewrites: 46517 in 429ms cpu (439ms real) (108179 rewrites/second)
result CheckingSolution:
checkingSolution(
   varDeclSet(
      varDecl('N, 'Nat), varDecl('E, 'Nat),
      varDecl('A, 'Array), varDecl('N@', 'Nat),
      varDecl('M@', 'Nat), varDecl('E@', 'Nat),
      varDecl('A@', 'Array)),
   critPairSet(
      ccp('T, 'in['N@', 'A@'],
          '_==_[('equal['N@', 'N@']), 'F], 'true),
      ccp('E@', 'at['A@', 'N@'],
          '_==_[('equal['N@', 'N@']), 'F], 'true),
      ccp('E@', 'undef['N@'],
```

```
        '_==_[('in['N@', ('put['N@', 'E@', 'A@'])]), 'F], 'true),
      ccp('at['A@', 'M@'], 'undef['M@'],
        'and[('_==_[('equal['N@', 'M@']), 'F]),
             ('_==_[('in['M@', ('put['N@', 'E@', 'A@'])]), 'F])],
        'true),
      ccp('undef['N@'], 'at['A, 'N@'],
        'and[('_==_[('in['N@', ('put['N, 'E, 'A])]), 'F]),
             ('_==_[('equal['N, 'N@']), 'F])],
        'true)),
  emptyMembAxSet)
```

Note that, since we cannot assume confluence (that is precisely what we are trying to prove) although it is clear that conditions like the one in the following critical pair are never going to be satisfied, they cannot be eliminated.

```
  ccp('T, 'in['N@', 'A@'], '_==_[('equal['N@', 'N@']), 'F], 'true)
```

Modulo checking unsatisfiability of the conditions, we can conclude that the specification is confluent. Since we do not have any membership proof obligation, we can conclude also that it is sort decreasing, and, therefore, Church-Rosser.

5.6.5 Lists and Stacks

The following two examples show the checking for two additional specifications, one for lists and another for stacks.

```
red checking(
     fmod('LIST,
       nilImportList,
       sort(qidSet('Elem, 'List, 'NeList)),
       subsortDeclSet(
          subsort('NeList, 'List),
          subsort('Elem, 'NeList)),
       opDeclList(
          opDecl('nil, nilQidList,
                 'List, emptyAttrSet),
          opDecl('cons, qidList('Elem, 'List),
                 'NeList, emptyAttrSet),
          opDecl('append, qidList('List, 'NeList),
                 'NeList, emptyAttrSet),
          opDecl('append, qidList('NeList, 'List),
                 'NeList, emptyAttrSet),
          opDecl('append, qidList('NeList, 'NeList),
                 'NeList, emptyAttrSet),
          opDecl('append, qidList('List, 'List),
                 'List, emptyAttrSet)),
       varDeclSet(
          varDecl('E, 'Elem), varDecl('NL1, 'NeList),
          varDecl('NL2, 'NeList), varDecl('NL3, 'NeList),
          varDecl('L1, 'List), varDecl('L2, 'List),
          varDecl('L3, 'List)),
```

```
       emptyMembAxSet,
       equationSet(
          eq('cons['E, 'nil], 'E),
          eq('append['L1, 'nil], 'L1),
          eq('append['nil, 'L1], 'L1),
          eq('append['cons['E, 'L1], 'L2],
             'cons['E, 'append['L1, 'L2]]),
          eq('append['E, 'L1],
             'cons['E, 'L1])))) .


rewrites: 17334 in 279ms cpu (289ms real) (61907 rewrites/second)
result CheckingSolution:
checkingSolution(
   emptyVarDeclSet,
   emptyCritPairSet,
   emptyMembAxSet)


red checking(
  fmod('STACK-OF-NAT,
       nilImportList,
       sort(qidSet('Stack, 'NeStack, 'Nat)),
       subsort('NeStack, 'Stack),
       opDeclList(
          opDecl('empty, nilQidList, 'Stack, emptyAttrSet),
          opDecl('push, qidList('Nat, 'Stack), 'NeStack, emptyAttrSet),
          opDecl('top, 'NeStack, 'Nat, emptyAttrSet),
          opDecl('pop, 'NeStack, 'Stack, emptyAttrSet)),
       varDeclSet(
          varDecl('X, 'Nat), varDecl('S, 'Stack)),
       emptyMembAxSet,
       equationSet(
          eq( 'top['push['X, 'S]], 'X),
          eq('pop['push['X, 'S]], 'S)))) .

rewrites: 1501 in 9ms cpu (19ms real) (150115 rewrites/second)
result CheckingSolution:
checkingSolution(
   emptyVarDeclSet,
   emptyCritPairSet,
   emptyMembAxSet)
```

References

[1] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. Manuscript. Submitted for publication, December 1997.

[2] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. In M. Bidoit and M. Dauchet, editors, *Proceedings*

TAPSOFT'97, Lecture Notes in Computer Science, volume 1214, pages 67–92, Springer-Verlag, 1997.

[3] 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.

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

[5] 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.

[6] I. Gnaedig, C. Kirchner, and H. Kirchner. Equational completion in order-sorted algebras. *Theoretical Computer Science*, 72:169–202, 1990.

[7] J. Goguen. OBJ as a theorem prover with application to hardware verification. In P. Subramanyam and G. Birtwistle, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 218–267. Springer-Verlag, 1989.

[8] J. Goguen. Theorem proving and algebra. Manuscript, August 1997.

[9] J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[10] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, Computer Science Laboratory, SRI International, March 1992.

[11] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[12] C. Kirchner. Order-sorted equational unification. In *5th International Conference on Logic Programming*, Seattle, U.S.A., August 1988. Also as Technical Report INRIA 954, December 1988.

[13] C. Kirchner, H. Kirchner, and J. Meseguer. Operational semantics of OBJ3. In T. Lepistö and A. Salomaa, editors, *Proceedings, 15th Intl. Coll. on Automata, Languages and Programming, Tampere, Finland, July 11-15, 1988*, pages 287–301. Springer LNCS 317, 1988.

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

[15] 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.

[16] 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.

[17] J. Meseguer. Membership algebra as a semantic framework for equational specification To appear in F. Parisi-Presicce, editor, Proc. WADT'97, Springer LNCS, 1998.