Building Equational Proving Tools by Reflection in Rewriting Logic*

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

Abstract

This paper explains the design and use of two equational proving tools, namely an inductive theorem prover—to prove theorems about equational specifications with an initial algebra semantics—and a Church-Rosser checker—to check whether such specifications satisfy the Church-Rosser property. These tools can be used to prove properties of order-sorted equational specifications in Cafe [11] and of membership equational logic specifications in Maude [7, 6]. The tools have been written entirely in Maude and are in fact *executable specifications* in rewriting logic of the formal inference systems that they implement.

Both of these tools treat equational specifications as *data* that is manipulated. For example, the inductive theorem prover may add an induction hypothesis as a new equational axiom; and the Church-Rosser checker computes critical pairs and membership axioms by inspecting the equations in the original specification. This makes a *reflective* design—in which theories become data at the metalevel—ideally suited for the task. The fact that rewriting logic is a reflective logic, and Maude efficiently supports reflective rewriting logic computations is systematically exploited in the tools. Rewriting logic reflection allows a modular and declarative treatment of *proof tactics*. Such tactics can be formally specificed by rewrite rules at the metalevel of the theory expressing the tool's inference system, and can be easily changed at will without any modification whatsoever to the inference system itself.

The very high level of abstraction at which the tools have been developed has made it relatively easy to build them, makes their implementation easy to understand, and will make their maintenance and future extensions much easier than for conventional implementations. Thanks to the high performance of the Maude engine these important benefits in ease of development, understandability, maintainability, and extensibility, and in flexibility for introducing formally-defined proof tactics, are achieved without sacrificing performance.

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

Contents

1 Introduction

This paper explains the design and use of two equational proving tools, namely an inductive theorem prover—to prove theorems about equational specifications with an initial algebra semantics—and a Church-Rosser checker—to check whether such specifications satisfy the Church-Rosser property. These tools have been developed as part of the Cafe project, and have been integrated, thanks to the efforts of our colleagues in Japan, within the overall Cafe environment. Through that integration, they can be used to prove formal properties of order-sorted equational specifications in Cafe [11]. They can also be used on their own to prove properties of equational specifications in Maude [7, 6]; that is, of its so-called functional modules, that are equational specifications in membership equational logic. An important feature of these tools is that they are written entirely in Maude and are in fact *executable specifications* in rewriting logic of the formal inference systems that they implement.

Both of these tools treat equational specifications as *data* that is manipulated. For example, the inductive theorem prover may add an induction hypothesis as a new equational axiom; and the Church-Rosser checker computes critical pairs and membership axioms by inspecting the equations in the original specification. This makes a *reflective* design—in which theories become data at the metalevel—ideally suited for the task. Indeed, the fact that rewriting logic is a reflective logic, and Maude efficiently supports reflective rewriting logic computations is systematically exploited in the tools.

Rewriting logic reflection has one important additional advantage, namely the modular and declarative treatment of *proof tactics* or *strategies*. Such tactics can be formally specificed by rewrite rules at the metalevel of the theory expressing the tool's inference system, and can be easily changed at will without any modification whatsoever to the inference system itself. By contrast, most conventional theorem proving tools either deal with tactics as extralogical features outside the formal framework, or build them in deep inside the tools's code, or both.

The very high level of abstraction at which the tools have been developed has made it relatively easy for us to build them, makes understanding their implementation much easier for other people, and will make their maintenance and future extensions much easier than if a conventional implementation, say in C, C++ or Java, had instead been chosen. Thanks to the high performance of the Maude engine—that can reach 590,000 rewrites per second for some applications running on a 200MHz Pentium Pro—these important benefits in ease of development, understandability, maintainability, and extensibility, and in flexibility for introducing formally-defined proof tactics, are achieved without sacrificing performance. In our experience, the tools—even though they have not been optimized for performance, and in spite of using several levels of reflection and sophisticated rewriting modulo associativity and associativity-commutativity—have quite competitive performance. For example, the inductive theorem prover is very responsive in interactive mode, with input-output time typically dominating over computation time. The Church-Rosser checker also offers reasonable performance. For example, generating all the critical pairs and membership assertions for the number hierarchy from the naturals to the rationals takes 2,156,488 rewrites in 31 seconds.

In the rest of this introduction we briefly review rewriting logic and Maude, including their reflective aspects, and explain in more detail the reflective design of the tools.

1.1 Rewriting Logic, Reflection, and Maude

Rewriting logic [21] is like a two-faced Janus, in that it expresses an essential equivalence between logic and computation in a particularly simple way. Namely, system *states* are in bijective correspondence with *formulas* (modulo whatever structural axioms are satisfied by such formulas: for example, modulo the associativity and commutativity of a certain connective) and concurrent *computations* in a system are in bijective correspondence with *proofs* (modulo appropriate notions of equivalence between computations and between proofs). Given this equivalence between computation and logic, a rewriting logic axiom of the form

$$t \longrightarrow t'$$

has two readings. Computationally, it means that a fragment of a system's state that is an instance of the pattern t can *change* to the corresponding instance of t' concurrently with any other state changes; that is, the computational reading is that of a *local concurrent transition*. Logically, it just means that we can derive the formula t' from the formula t; that is, the logical reading is that of an *inference rule*.

Rewriting logic is entirely neutral about the structure and properties of the formulas/states t. They are entirely *user-definable* as an algebraic data type satisfying certain equational axioms, so that rewriting deduction takes place *modulo* such axioms. Because of this ecumenical neutrality, rewriting logic has, from a logical viewpoint, good properties as a *logical framework* [20], in which many other logics can be naturally represented. And, computationally, it has also good properties as a *semantic framework* [23], in which many different system styles and models of concurrent computation can be naturally expressed without any distorting encodings.

The design of the present tools exploits and illustrates the logical framework capabilities of rewriting logic by formally specifying the *inference system* of each tool as rewrite theory. The fact that these tools reason about *equational theories* is not a restriction of the general framework capabilities: inference systems for reasoning about specifications in any other logic could be represented just as easily. However, the fact that rewriting logic contains equational logic as a sublanguage and that Maude has good support for its equational sublanguage is an added advantage.

We can represent a *rewrite theory* as a four-tuple $T = (\Omega, E, L, R)$, where (Ω, E) is a theory in membership equational logic[1] [25, 3], L is a set of labels, to label the rules, and R is the set of labeled rewrite rules axiomatizing the local state transitions of the system. Some of the rules in R may be conditional [21]. Of course, we can view an equational theory as the special case of a rewrite theory in which the labels L and rules R are both empty. In this way, equational logic appears naturally as a sublanguage of rewriting logic.

Maude [7, 6] is a language whose modules are theories in rewriting logic. The most general Maude modules are called *system modules*. They have the syntax `mod` T `endm` with T the rewrite theory in question, expressed with a syntax quite close to the corresponding mathematical notation.[2] The equations E in the equational theory (Ω, E) underlying the rewrite theory $T = (\Omega, E, L, R)$ are presented as a union $E = A \cup E'$, with A a set of *equational axioms* introduced as *attributes* of certain operators in the signature Ω—for example, an operator $+$ can be declared associative and commutative by keywords `assoc` and `comm`—and where E' is a set of equations that are assumed to be Church-Rosser and terminating *modulo* the axioms A. Maude supports rewriting modulo different combinations of such equational attributes: operators can be declared associative, commutative, with identity, and idempotent [6]. Maude contains a sublanguage of *functional modules* of the form `fmod` (Ω, E) `endfm`, where, as before, $E = A \cup E'$, with E' Church-Rosser and terminating modulo A. A system module `mod` T `endm` specifies the initial model of the rewrite theory T. Similarly, a functional module `fmod` (Ω, E) `endfm` specifies the initial algebra of the equational theory (Ω, E), which is at the same time the initial model of (Ω, E) when viewed as a (degenerate) rewrite theory.

Rewriting logic is reflective [8, 9] in the precise sense that there is a finitely presented rewrite theory U such that for any finitely presented rewrite theory T (including

[1] More generally, the choice of the equational logic underlying rewriting logic is a parameter that can be instantiated to different variants; however, since membership equational logic extends in a conservative way many other equational logics, including order-sorted equational logic [25], this choice is quite expressive and is the one that has been adopted for Maude.

[2] See [6] for a detailed description of Maude's syntax, which is quite similar to that of OBJ3 [16]. In this paper we adopt an optimistic viewpoint, assuming that the syntax of the modules that we present is for the most part self-explanatory.

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

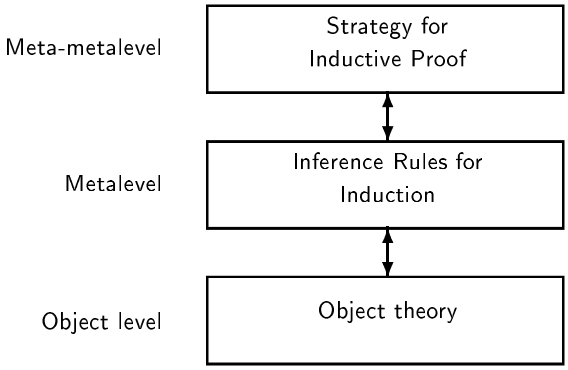
$$T \vdash t \longrightarrow t' \iff U \vdash \langle \overline{T}, \overline{t} \rangle \longrightarrow \langle \overline{T}, \overline{t'} \rangle \iff U \vdash \langle \overline{U}, \overline{\langle \overline{T}, \overline{t} \rangle} \rangle \longrightarrow \langle \overline{U}, \overline{\langle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

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 labeled 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.

1.2 Reflective Design of the Tools

Based on these concepts, each of the 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.

Regarding the rest of the paper, 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 more 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. We end the paper with some concluding remarks.

Acknowledgments

We have benefited very much from discussions and positive suggestions from Patrick Lincoln and Natarajan Shankar at SRI, and from those of our fellow members in the Cafe Project, particularly from Kokichi Futatsugi, Ataru Nakagawa, Toshimi Sawada, 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 [2, 3], 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 [13, 14]. We have benefited considerably from Joseph Goguen's 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 [21, 26, 22]. An introduction to Maude and its interpreter implementation can be found in [7, 5, 6]. For our present purposes the key point is that the Maude implementation supports an arbitrary number of levels of reflection and gives the user access to important reflective capabilities, including the possibility of defining and using internal strategy languages.

The reflective kernel of the Maude system is the built-in module `META-LEVEL`, which provides key functionality in the *universal theory* U for rewriting logic [8, 9, 4]:

- 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`.

Any rewrite theory $T = (\Sigma, E, R)$ and any term $t \in T_\Sigma(X)$ are reified, respectively, by data elements $\overline{T}$: `Module` and $\overline{t}$: `Term` in U.

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 nonzero integers, each with the usual operations, having the expected attributes. The sort `Qid` contains quoted identifiers beginning with the apostrophe symbol.

As already mentioned, `META-LEVEL` provides an abstract data type `Module` to represent modules in Maude. It includes constructors `mod` and `fmod` to represent system modules and functional modules, respectively. There are sorts and constructors for each of the declarations that can appear in modules.

```
op fmod : Qid ImportList SortDecl SubsortDeclSet OpDeclList
          VarDeclSet MembAxSet EquationSet -> Module .
op mod :  Qid ImportList SortDecl SubsortDeclSet OpDeclList
          VarDeclSet MembAxSet EquationSet RuleSet -> Module .
```

Besides sorts `Term` and `Module`, the module `META-LEVEL` includes other sorts whose terms represent items appearing in a module or auxiliary data. There are sorts `Sort` (for sort identifiers), `Attr` (for operator attributes) and `Assignment` (assignments of terms to variables in substitutions); sorts for the declaration of submodule inclusions (`Import`), sorts (`SortDecl`), subsort relations (`SubsortDecl`), operations(`OpDecl`), variables (`VarDecl`), equations (`Equation`), rules (`Rule`) and membership axioms (`MembAx`); sorts for sets[3] or lists of some of such declarations (`ImportList`, `SubsortDeclSet`, `OpDeclList`, `VarDeclSet`, `EquationSet`, `RuleSet`, `MembAxSet`); and some other sorts, such as `QidList` or `Substitution`.

Sort declarations are represented using a constructor

```
op sort : QidSet -> SortDecl .
```

where `QidSet` is the sort of sets of quoted identifiers. We have declarations:

```
subsort Qid < QidSet .
op emptyQidSet : -> QidSet .
op qidSet : QidSet QidSet -> QidSet [assoc comm id: emptyQidSet] .
```

[3]Although they are called "sets", actually they are "multiselts" because there is missing the idempotent attribute.

Subsort declarations are represented with the syntax:

```
subsort SubsortDecl < SubsortDeclSet .
op subsort : Qid Qid -> SubsortDecl .
op emptySubsortDeclSet : -> SubsortDeclSet .
op subsortDeclSet : SubsortDeclSet SubsortDeclSet -> SubsortDeclSet
      [assoc comm id: emptySubsortDeclSet] .
```

META-LEVEL also includes similar syntax declarations for representing the other elements that can appear in a module. Without going into all the details, let us for example focus in the representation of operation declarations. We have already mentioned the sorts OpDecl and OpDeclList, which provide declarations of operations and lists of declarations of operations, respectively. Each of the operation declarations is given by a term of sort OpDecl using a constructor

```
op opDecl : Qid QidList Qid AttrSet -> OpDecl .
```

AttrSet is a sort with a union operator that is declared as a supersort of Attr. The sort Attr is used for representing operator attributes allowed in Maude (see [6]) including attributes that are equational axioms used to rewrite modulo. For example, the commutativity attribute is represented by the constant

```
op comm : -> Attr .
```

and a commutative natural number addition operator is represented by the term

```
opDecl('_+_, qidList('Nat, 'Nat), 'Nat, comm)
```

Note that names of sorts and operators (and also those of variables) are represented in META-LEVEL as quoted identifiers. In particular, a "mixfix" operator like + is represented by the corresponding quoted identifier with underbar characters indicating the argument positions.

If there are no operation declarations in a particular module, then the corresponding argument will be nilOpDeclList. Otherwise, the list is given by the associative constructor opDeclList, which is declared also with identity element nilOpDeclList.

```
op nilOpDeclList : -> OpDeclList .
op opDeclList : OpDeclList OpDeclList -> OpDeclList
      [assoc id: nilOpDeclList] .
```

The intended meaning of the syntax used to represent modules can be easily inferred from examples. For instance, let us introduce the module NAT for natural numbers with zero and successor and commutative addition and multiplication operators[4].

```
fmod NAT is
  sorts Zero Nat .
  subsort Zero < Nat .
  op 0 : -> Zero .
  op s : Nat -> Nat .
```

[4]This module will be used later in some examples illustrating the tools. The extra variables X, Y and Z have been introduced because the formulas to be proved by the theorem prover in Section 4 have to use variables already declared in the module.

```
  op (_+_) : Nat Nat -> Nat [comm] .
  op (_*_) : Nat Nat -> Nat [comm] .
  vars N M X Y Z : Nat .
  eq 0 + N = N .
  eq s(N) + M = s(N + M) .
  eq 0 * N = 0 .
  eq s(N) * M = M + (N * M) .
endfm
```

The representation of NAT in META-LEVEL, which we denote $\overline{\texttt{NAT}}$, is the following term of sort 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('X, 'Nat), varDecl('Y, 'Nat), varDecl('Z, 'Nat)),
     emptyMembAxSet,
     equationSet(
       eq('_+_['0, 'N], 'N),
       eq('_+_['s['N], 'M], 's['_+_['N, 'M]]),
       eq('_+_['0, 'N], '0),
       eq('_+_['s['N], 'M], '_+_['N, '_*_['N, 'M]])))
```

Terms are reified as elements of the data type Term of terms, where constants and variables are, as already mentioned, represented as quoted identifiers, and more complex terms are represented using the constructors:

```
op _[_] : Qid TermList -> Term .
op _,_ : TermList TermList -> TermList [assoc] .
op error* : -> Term .
```

For example, the term

```
s(s(0)) + s(0)
```

is meta-represented as

```
'_+_['s['s['0]], 's['0]].
```

Note that meta-representation of terms can be iterated. For example, the meta-representation of the term `'_+_['s['s['0]], 's['0]]` is

```
'_[_][''_+_, ('_,_[('_[_][''s, ('_[_][''s, ''0])]), ('_[_][''s, ''0])])].
```

The operations meta-reduce and meta-apply are key for the reflective capabilities of the Maude system.

The operation meta-reduce takes as arguments the representations in META-LEVEL of a module T and a term t or a membership predicate $t : s$ in that module.

```
op meta-reduce : Module Term -> Term .
```

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,

```
Maude> red meta-reduce(NAT, s(0) + 0) .
result Term:  s(0)
```

Membership predicates can be represented in META-LEVEL using the constructor

```
op membPred : Term Qid -> Term .
```

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.

The operation meta-apply has syntax:

```
op meta-apply : Module Term Qid Substitution MachineInt -> ResultPair .
```

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) defining a partial substitution 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 _<-_. A substitution is a set of assignments of this form. The META-LEVEL syntax for representing assignments and substitutions is:

```
subsort Assignment < Substitution .
op _<-_ : Qid Term -> Assignment .
op emptySubstitution : -> Substitution .
op _;_ : Substitution Substitution -> Substitution
      [assoc comm id: emptySubstitution] .
```

meta-apply returns a pair of sort ResultPair consisting of a term and a substitution combined by means of the constructor

```
op {_,_} : Term Substitution -> ResultPair .
```

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.

The operations `meta-reduce`, `meta-apply` and the other functions in `META-LEVEL` are functions equationally definable in the module `META-LEVEL`. This is a consequence of the fact that, as already mentioned, rewriting logic is a reflective logic (see [9, 4]). However, for efficiency and convenience they are built-in functions in Maude. Assuming that all the built-in functions are in fact equationally defined, `META-LEVEL` then becomes a normal module in Maude that can itself be represented as a term of sort `Module` in `META-LEVEL`. The Maude implementation produces the exact same behavior as if `META-LEVEL` had thus been defined as a normal Maude module, but for efficiency reasons it performs 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. For example, we can meta-reduce the pair consisting of the meta-representation of `META-LEVEL` and the meta-representation of the term meta-reducing an expression in `NAT`, and will then get the meta-meta-representation of the result,

$$\texttt{Maude> red meta-reduce(}\overline{\texttt{META-LEVEL}}\texttt{, }\overline{\texttt{meta-reduce(}\overline{\texttt{NAT}}\texttt{,}\overline{\texttt{s(0) + 0}}\texttt{))}}\texttt{ .}$$
$$\texttt{result Term: }\overline{\overline{\texttt{s(0)}}}$$

As already mentioned, this capacity for supporting a reflective tower is used crucially in the inductive theorem prover: since the inductive inference rules are at the metalevel, the proof strategies must then be evaluated at the meta-metalevel.

`META-LEVEL` also includes some operations on sorts. We will only discuss here those we need: `sortLeq` and `leastSort`. At the meta-level, the sorts given by the user in his/her module are represented as quoted identifiers, that is, as 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[5] 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., [15]). 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 .
```

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`.

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

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 flexible ways of controlling the rewriting inference

[5]These error sorts are in fact the *kinds* of the membership equational logic specification implicitly defined by the user's specification [25]. Each kind contains all the well-sorted terms of the connected component, plus additional error terms that cannot be assigned a sort.

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 [8, 9].

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 emptyPairSet : -> PairSet .
  op pairSet : PairSet PairSet -> PairSet [assoc comm id: emptyPairSet] .
  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
```

The sort `PairSet` contains sets P of pairs of integers. For the sake of the example, let us suppose[6] that for any pair $\texttt{pair}(i,\ x)$ in an input set P, i is between 1 and the number of elements in P, and we cannot have in P two different pairs $\texttt{pair}(i,\ x)$ and $\texttt{pair}(j,\ y)$ 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.,

```
Maude> rew pairSet(pair(1, 3), pair(2, 2), pair(3, 1)) .
result PairSet: pairSet(pair(1, 1), pair(2, 2), pair(3, 3))
```

However, using the default-interpreter we have no control over the application of the rules—in this example of the single rule `sort`. Although not a problem in this case, because the module is in fact confluent and terminating, 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

[6]These requirements on the input could have been axiomatized by membership axioms, but this is not necessary for our present purposes.

example, we introduce the following module STRATEGY. This is a subset of the strategy language in [5, 4, 6].

```
protecting META-LEVEL[SORTING] .
sorts BindingList Strategy StrategyExpression .

op rewInWith : Module Term BindingList Strategy -> StrategyExpression .
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 orelse : Strategy Strategy -> Strategy .
op extTerm : ResultPair -> Term .
op extSubst : ResultPair -> Substitution .

var M : Module .
var L : Qid .
var T : Term .
var SB : Substitution .
var BL : BindingList .
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[7]. 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 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)
```

[7]We do not describe here either the use of this information—stored in what we call metavariables—or the strategies to manipulate them or access their contents. These and other strategies, together with fully detailed explanations about them, can be found in [5, 4, 6].

```
        == {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:

```
Maude> red rewInWith(SORTING,
             pairSet(pair(1, 3), pair(2, 2), pair(3, 1)),
             nilBindingList, apply('sort)).
result StrategyExpression:
rewInWith(SORTING,
             pairSet(pair(1, 1), pair(2, 2), pair(3, 3)),
             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 using the first successful match consistent with the given partial substitution[8].

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

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))
```

[8]Since we do not need to make use of metavariables for the tools, we give here a simplified version of this strategy. Note that if the representations of the terms assigned to variables contain metavariables, then they must be substituted using the representations of the terms they are bound to in the current list of bindings.

```
    = 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.

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[9]. As in the case of the Church-Rosser checker tool that will be discussed in Section 5, it is very natural to adopt a reflective design to specify in Maude an inductive theorem-prover. 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 $\overline{T}$, the metarepresentation of module T, 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 and illustrate their use. We also give an inductive proof method to ensure that a set of constructors generate the initial algebra of the specification—so that only terms involving such constructors need to be used in the *test sets* used in the induction scheme—and illustrate this proof method with examples. Finally, we discuss the generation of the test sets themselves.

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. Note the "backwards reasoning" form of the rules. That is, in each rule the inference above the bar is in fact the conclusion that can be established if we can prove the "subgoals" under the bar. This presentation of the rules agrees with the intended use of the inductive prover, in which a user will submit an atomic sentence to be proved as the initial goal, and then this goal will be successively transformed by rewriting—using the inference rules 1–6 as rewrite rules—into different sets of subgoals, until no subgoals are left if the proof succeeds.

[9]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 [25], Cafe modules are easily translated into corresponding Maude modules.

Note also that the atomic sentences that these rules can attempt to prove are precisely those of membership equational logic. Since a very general version of order-sorted algebra can be embedded in membership algebra in a conservative way [25], they remain sound for proving inductive theorems of order-sorted functional specifications in Cafe. In fact, the extra generality of membership equational logic allows us to prove useful properties of order-sorted specifications that cannot be stated as order-sorted sentences. Two good examples are sufficient generation proofs—that are expressed as inductive proofs of membership assertions for the defined symbols (Section 4.3)—and inductive proofs of membership assertions generated as proof obligations by the Church-Rosser Checker to ensure the ground-Church-Rosser property for a given order-sorted specification (Section 5.4).

1. **Test Set Induction**

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

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

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

2. **Proof in Variety**

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

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

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

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

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

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

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

6. **Equational and Membership Lemmas**

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

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

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 [13, 14] and LARCH [17]. We give now an overview of the specification in Maude of the inference system just introduced. This specification is given by a module called `ITP`, in which formulas are represented with the operations `trueFormula`, `equality`, `sortP`, `implication`, `conjunction`, and `VQuantification`.

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

In what follows, given a formula α, $\overline{\alpha}^f$ denotes its representation in the module `ITP`, that we describe informally below. We illustrate this representation using the module `NAT` in page 8. For example, the universally quantified atomic formula $(\forall\{\texttt{N},\texttt{M}\}).\texttt{N + M} = \texttt{M + N}$ is represented in `ITP` by the term

$$\texttt{VQuantification(qidSet(}\overline{\texttt{N}}\texttt{, }\overline{\texttt{M}}\texttt{), equality(}\overline{\texttt{N + M}}\texttt{, }\overline{\texttt{M + N}}\texttt{)).}$$

Similarly, $(\forall\{\texttt{N},\texttt{M}\}).\texttt{N + M} : \texttt{Nat}$ is represented by the term

$$\texttt{VQuantification(qidSet(}\overline{\texttt{N}}\texttt{, }\overline{\texttt{M}}\texttt{), sortP(}\overline{\texttt{N + M}}\texttt{, }\overline{\texttt{Nat}}\texttt{)).}$$

Note that the variables used in the formulas correspond to variables that have already been declared in the given module, in $\overline{\texttt{NAT}}$ in this case.

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 $\texttt{test(}\overline{\texttt{Nat}}\texttt{, termSet(}\overline{\texttt{0}}\texttt{, }\overline{\texttt{s(N)}}\texttt{))}$ is the representation of the test set[10] 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

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

is represented in `ITP` by the term

$$\texttt{proveinInitial(}I\texttt{, }\overline{\texttt{NAT}}\texttt{, }\overline{(\forall\{\texttt{N,M}\}).\texttt{N + M} = \texttt{M + N}}^{f}\texttt{, NATTS),}$$

where I should be a string of positive numbers, and `NATTS` is the representation of the set of test sets for the sorts of `NAT`, that is, the term

$$\begin{array}{l}\texttt{testSet(test(}\overline{\texttt{Nat}}\texttt{, termSet(}\overline{\texttt{0}}\texttt{, }\overline{\texttt{s(N)}}\texttt{)),}\\ \qquad\quad\texttt{test(}\overline{\texttt{Zero}}\texttt{, }\overline{\texttt{0}}\texttt{)).}\end{array}$$

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.

```
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] .
```

The rewrite rules in the module `ITP` mirror very closely the inference system described above. When explaining the rules in `ITP` we also explain briefly some of the auxiliary functions used in them; full details about these auxiliary functions can be found in [5].

We start with the declaration of some variables that will be used in the rules:

[10] 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 4.3 and 4.4, 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.

```
var IS : IntString .
vars X S S' : Qid .
vars T T1 T2 : Term .
var TS : TermSet .
var Th : Module .
var TSS : TestSet .
var G : GoalSet .
```

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

```
crl [testSetInduction]:
    goalSet(proveinInitial(IS, Th,
                VQuantification(qidSet(X, XS), equality(T1, T2)),
                testSet(test(S, TS), TSS)), G)
   => goalSet(makeNewGoals(intString(IS, 1), Th, XS,
                           X, T1, T2, TS,
                           testSet(test(S, TS), TSS)), G)
      if findSort(X, getVars(Th)) == S .

crl [testSetInduction]:
    goalSet(proveinInitial(IS, Th,
                 VQuantification(qidSet(X, XS), sortP(T, S')),
                 testSet(test(S, TS), TSS)), G)
   => goalSet(makeNewGoalsMb(intString(IS, 1), Th, XS,
                             X, T, S', TS,
                             testSet(test(S, TS), TSS)), G)
      if findSort(X, getVars(Th)) == S .
```

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

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

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

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

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`, `$\overline{T}$`, `$\overline{X}$`, `$\overline{x}$`, `$\overline{t}$`, `$\overline{t'}$`, `$\overline{\{u_1, \ldots, u_n\}}$`, `$\overline{TS}$`)` returns

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

with each new subgoal conveniently numbered after I.

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

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

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

with each new subgoal conveniently numbered after I.

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

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

```
rl [inVariety]:
   goalSet(proveinInitial(IS, Th, Alpha, TSS), G)
  => goalSet(proveinVariety(IS, Th, Alpha), G) .
```

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

```
rl [constantsLemma]:
   goalSet(proveinVariety(IS, Th,
                  VQuantification(XS, Alpha)), G)
  => goalSet(proveinVariety(IS,addNewConstants(XS, Th),
                    varsToNewConstants(XS, Alpha)), G) .
```

For T a functional module, and X a set of variables declared in T, the function addNewConstants($\overline{X}$, $\overline{T}$) returns $\overline{T'}$, 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.

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

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($\overline{\{x_1,\ldots,x_n\}}$, $\overline{\alpha}$) returns $\overline{\alpha[x_1*/x_1,\ldots,x_n*/x_n]}$.

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

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

```
rl [implicationElimination]:
   goalSet(proveinVariety(IS, Th, implication(Alpha, Beta)), G)
  => goalSet(proveinVariety(IS, addToModule(Alpha, Th), 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($\overline{\alpha}$, $\overline{T}$) 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 $t_i : s_i$ 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.

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

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 these inference rules.

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

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

eq proveinVariety(IS, Th, equal(T, T)) = emptyGoalSet .
```

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

```
rl [lemmaIntroduction]:
   goalSet(proveinInitial(IS, Th, Alpha, TSS), G)
  => goalSet(proveinInitial(intString(IS, 1),
                            addToModule(Beta, Th), Alpha, TSS),
            proveinInitial(intString(IS, 2),
                            Th, Beta, TSS), G) .
```

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.

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, ('IS <- I)),
       orelse(
         applyWithSubst('byRewriting, ('IS <- I)),
         and(applyWithSubst('constantsLemma, ('IS <- I)),
             orelse(
```

```
                applyWithSubst('byRewriting, ('IS <- I)),
                and(applyWithSubst('implicationElimination, ('IS <- I)),
                    and(orelse(
                          applyWithSubst('constantsLemma, ('IS <- I)),
                          idle),
                        applyWithSubst('byRewriting, ('IS <- I))))))) .
```

We can also represent a proof strategy that, for a given goal, applies the induction rule on a selected variable and then 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)))))))))) .
```

Suppose that we want to prove the associativity of the operation `+` (addition) as defined in the module `NAT` presented in page 8, that is, we want to prove the formula $(\forall\{\mathtt{X},\mathtt{Y},\mathtt{Z}\})$.`X + (Y + Z)` $=$ `(X + Y) + Z`. We can prove it applying the `induction` strategy on the variable `Z`.

```
Maude> rew rewInWith(ITP,
              'proveinInitial['1, NAT,
                 (∀{X, Y, Z}).X + (Y + Z) = (X + Y) + Z^f, NATTS],
              nilBindingList, induction(Z)) .

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

4.3 Sufficient Generation

The test set induction rule instantiates an induction variable by the terms in a test set. The most classical case is the instantiation of a variable of sort `Nat` by `0` and `s(N)`. These expressions suffice, because zero and successor are the *constructors* that build up all natural numbers. In general, however, how can we justify the correctness of a test set? We need to show that all the ground terms of a given sort are provably equal to constructor terms that are instances of the terms in the test set for that sort. This brings us to the topic of sufficient generation, that is, of showing that a given constructor subspecification—that may actually satisfy some equations of its own—generates all the sorts in the initial algebra of our specification, in the sense that all terms of that sort can be proved equal to constructor terms. We give here a general method of using the inductive theorem prover to prove sufficient generation, and illustrate the method with some examples; then we discuss a simple test set generation scheme in Section 4.4. Our method is based on a theorem about protecting extensions of membership equational logic specifications by Bouhoula, Jouannaud and Meseguer [2]. For the sake of a simpler exposition, we deal here with the case of order-sorted specifications. The more general case of membership equational logic specifications has a similar treatment and will be discussed elsewhere.

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 [2, 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[11] $f : s_1 \ldots s_n \to s$ 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 conservative embedding of order-sorted equational logic into membership equational logic [25], 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 $E_\Omega T$.

4.3.1 Proving Sufficient Generation of Natural Numbers

We will prove that the constructors for the module `NAT`[12] presented in page 8 are `0` and `s`. To do this we will introduce a new sort `ErrorNat` as supersort of `Nat` and will move up all the operations except those considered as candidates for constructors.

```
fmod 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) .
endfm
```

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 will be explained in Section 4.4. 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 nonconstructors using the `copDecl` operator for the constructors.

Then, calling the function `generateTestSets` with `CNAT` as argument gives us a pair consisting of a modified module, in which some variables may have been introduced,

[11] 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.

[12] We will not need the variables `X`, `Y` and `Z` anymore, so we are not considering them.

and the corresponding family of test sets. Note that, because new variables may have been introduced, 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[13]

```
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 $\overline{\texttt{ENAT}}$. Let us call ENATTS the test sets returned:

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

To prove the goal `(∀{N,M}).N * M: Nat` we introduce the formula `(∀{N, M}).N + M: Nat` as a lemma and prove it, thus settling the two proof obligations for sufficient generation within a single overall proof. We can do it by applying the rule

[13]In Maude, it is allowed the use of *flattened* form, with an arbitrary number of arguments, for operators declared to be associative. Thus, for example, given the declaration

```
op qidSet : QidSet QidSet -> QidSet [assoc comm id: emptyQidSet] .
```

the terms `qidSet(qidSet('Zero, 'Nat), 'ErrorNat)` and `qidSet('Zero, qidSet('Nat, 'ErrorNat))` are both equivalent to the flattened form `qidSet('Zero, 'Nat, 'ErrorNat)`.

`lemmaIntroduction` with the variable `Beta` instantiated to the meta-representation of the formula as described in page 17.

Doing this, we obtain the following two subgoals:

$$(1.1)\quad \texttt{ENAT} \cup \{\texttt{mb N + M: Nat .}\} \vdash_{\mathrm{ind}} (\forall\{\texttt{N, M}\}).\texttt{N * M: Nat}$$
$$(1.2)\quad \texttt{ENAT} \vdash_{\mathrm{ind}} (\forall\{\texttt{N, M}\}).\texttt{N + M: Nat}$$

Both of these subgoals can then be proved by induction. We can start, for example, applying the rule `testSetInduction` to the subgoal `1.1`. This generates two new subgoals:

$$(1.1.1)\quad \texttt{ENAT} \cup \{\texttt{mb N + M: Nat .}\} \vdash_{\mathrm{ind}} (\forall\{\texttt{M}\}).\texttt{0 * M: Nat}$$
$$(1.1.2)\quad \texttt{ENAT} \cup \{\texttt{mb N + M: Nat .}\} \vdash_{\mathrm{ind}} (\forall\{\texttt{N}\}).((\forall\{\texttt{M}\}).\texttt{N + M: Nat}) \Rightarrow ((\forall\{\texttt{M}\}).\texttt{s(N) + M: Nat})$$

These two new subgoals, `1.1.1` and `1.1.2`, one for each one of the terms in the test set of `Nat`, can be proved by rewriting applying the `simplify` strategy. Once these subgoals are simplified, the only subgoal remaining is `1.2`, which can then be proved applying the `induction` strategy on the variable `N`. The interaction with the tool can be done step by step or just giving:

```
red rewInWith(ITP,
        'proveinInitial[ '1, ENAT,  (∀{N, M}).N * M: Nat^f,  ENATTS],
        nilBindingList,
        and(applyWithSubst(
              'lemmaIntroduction,
              (('IS <- '1);    ('Beta <- (∀{N, M}).N + M: Nat^f))),
        and(applyWithSubst(
              'testSetInduction,
              (('X <- N); ('IS <- ('intString['1, '1])))),
        and(simplify('intString['1, '1, '1]),
        and(simplify('intString['1, '1, '2]),
            induction(N)))) .
```

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

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

4.3.2 Proving Sufficient Generation of Integers

Given the module `NAT` as in page 8, let us consider now the following specification for integers:

```
fmod 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 .
endfm
```

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 4.4, 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 $\overline{\texttt{EINT}}$ 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 set 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('p, 'Neg, 'Neg, emptyAttrSet),
      opDecl('s, 'ErrorInt, 'ErrorInt, emptyAttrSet),
      opDecl('p, 'ErrorInt, 'ErrorInt, emptyAttrSet),
```

```
    opDecl('_+_, qidList('ErrorInt, 'ErrorInt), 'ErrorInt, comm),
    opDecl('_*_, qidList('ErrorInt, 'ErrorInt), 'ErrorInt, comm),
    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:

$$\begin{array}{lll}
\texttt{EINT} & \vdash_{\mathrm{ind}} & (\forall\{\texttt{I}\}).\texttt{s(I): Int} \\
\texttt{EINT} & \vdash_{\mathrm{ind}} & (\forall\{\texttt{I}\}).\texttt{p(I): Int} \\
\texttt{EINT} & \vdash_{\mathrm{ind}} & (\forall\{\texttt{I}\}).\texttt{-I: Int} \\
\texttt{EINT} & \vdash_{\mathrm{ind}} & (\forall\{\texttt{I,J}\}).\texttt{I + J: Int} \\
\texttt{EINT} & \vdash_{\mathrm{ind}} & (\forall\{\texttt{I,J}\}).\texttt{I * J: Int} \\
\texttt{EINT} & \vdash_{\mathrm{ind}} & (\forall\{\texttt{I}\}).\texttt{square(I): Nat}
\end{array}$$

A detailed presentation of the proofs can be found in [5]. A slight variant of the last goal will be needed in Section 5.5.2 for proving inductively one of the proof obligations generated by the Church-Rosser checker tool when checking `INT`.

4.4 Test Set 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)` and `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 nonconstructor operations using `opDecl`. We have the following declarations:

```
sorts EModule EOpDecl EOpDeclList .

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

op nilEOpDeclList : -> EOpDeclList .
op eOpDeclList : EOpDeclList EOpDeclList -> EOpDeclList
      [assoc id: nilEOpDeclList] .
```

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 back into the standard module syntax before it is used in the ITP.

5 The Church-Rosser Checker

The goal of *executable* equational specification languages is to make the abstract data types specificed in them by initial algebra semantics *computable*. In practice this is accomplished by using specifications that are *ground*-Church-Rosser and terminating, so that the equations can be used from left to right as simplification rules; the result of evaluating an expression is then the canonical form that stands as a unique representative for the equivalence class of terms equal to the original term according to the equations. This approach is fully general; indeed, a well-known result of Bergstra and Tucker [1] shows that *any* computable algebraic data type can be specified by means of a finite set of ground-Church-Rosser and terminating equations, perhaps with the help of some auxiliary functions added to the original signature.

Therefore, for formal reasoning purposes it becomes very important to know whether a given specification is indeed ground-Church-Rosser and terminating. A nontrivial question is how to best support this with adequate tools. For proving termination one can use standard termination proof techniques that are well-supported in tools such as CiME [10]. A thornier issue is what to do for establishing the ground-Church-Rosser property for a terminating specification. The problem is that a specification with an initial algebra semantics can often be ground-Church-Rosser even though some of its critical pairs may not be joinable. That is, the specification can often be ground-Church-Rosser without being Church-Rosser for arbitrary terms with variables. In such a situation, blindly applying a completion procedure that is trying to establish the Church-Rosser property for arbitrary terms may be both quite hopeless—the procedure diverges or gets stuck because of unorientable rules, and even with success may return a specification that is quite different from the original one—and unnecessary: the specification was ground-Church-Rosser!

Our Church-Rosser checker tool is oriented to checking initial algebra specifications that have already been proved terminating using a termination tool and now need to be checked to be ground-Church-Roser. Since, for the reasons mentioned above, user interaction will typically be quite essential, completion is not attempted. Instead, if the specification cannot be shown to be ground-Church-Rosser by the tool, proof obligations are generated and are given back to the user as a useful guide in the attempt to establish the ground-Church-Rosser property. Since this property is in fact inductive, in some cases (see Section 5.5.2) the inductive theorem prover can be enlisted to prove some of these proof obligations. In other cases, the user may in fact have to modify the original specification by carefully considering the information conveyed by the proof obligations. We give in Section 5.5 some methodological guidelines for the use of the tool, and illustrate the use with some examples; we also explain there that the issue of finding general inductive proof techniques for proving the ground-Church-Rosser property is at the moment an interesting open problem.

The present tool only accepts order-sorted conditional specifications where each of the operation symbols has either no equational attributes, or only the commutativity attribute; furthermore, it is assumed that such specifications have already been proved terminating using another tool. The tool attempts to establish the ground-Church-Rosser property *modulo* the commutativity of some of the operators by checking a sufficient condition. For order-sorted specifications, being Church-Rosser and terminating means not only confluence—so that a unique normal form will be reached—but also a *descent* property, namely that the normal form will have the least possible sort among those of all other equivalent terms. Therefore, the tool's output consists of 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.

After introducing some basic formal concepts and results underlying the tool's design and discussing several auxiliary functions used by the tool, including order-sorted commutative unification, we explain key parts of the tool and then give recommendations for its use and some examples.

5.1 Church-Rosser Order-Sorted Specifications

In this section we introduce the notion of Church-Rosser order-sorted specification. We assume specifications of the form $\mathcal{S} = (\Sigma, R \cup A)$ where Σ is a preregular order-sorted signature [15], and $R \cup A$ is a set of equations such that A is a set of *sort-preserving* equational axioms, that is, whenever $t =_A t'$ we have $LS(t) = LS(t')$. The equations R will be used as rewrite rules modulo the axioms A. Furthermore, in what follows, and for the purposes of the present tool, the axioms A will only consists of *commutativity* axioms for certain binary operators in Σ. The problem them is to check whether our specification $\mathcal{S}$ has the Church-Rosser property.

After giving some auxiliary definitions, we introduce the notion of Church-Rosser order-sorted specifications and describe the sufficient condition used by our tool to attempt checking the Church-Rosser property.

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 substitution is a replacement operation uniquely defined by a mapping from variables to terms, and is written out as $\{x_1 \leftarrow s_1, \ldots, x_n \leftarrow s_n\}$. Given a set of axioms A, a substitution σ is an A-*unifier* of t and t' if $\sigma t =_A \sigma t'$, and it is an A-*match* from t to t' if $t' =_A t\sigma$.

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

We say that a term t *A-overlaps* another term with distinct variables t' if there is a nonvariable subterm $t'|_p$ of t' for some position p such that the terms t and $t'|_p$ can be A-unified.

Definition 1 *Given an order-sorted equational specification $\mathcal{S} = (\Sigma, R \cup A)$ with the above assumptions, and given conditional rewrite rules $l \to r$ `if` $\bar{u} \downarrow \bar{v}$ and $l' \to r'$ `if` $\bar{u'} \downarrow \bar{v'}$ in R such that $vars(l) \cap vars(l') = \emptyset$ and $l|_p\sigma =_A l'\sigma$, for some nonvariable position $p \in \mathcal{P}_M(l)$ and A-unifier σ, then $ccp(l\sigma[r'\sigma]_p,\ r\sigma,\ \bar{u}\sigma \downarrow \bar{v}\sigma \wedge \bar{u'}\sigma \downarrow \bar{v'}\sigma)$ is a critical pair.*

In the uses we will make of the above definition we will always assume that the unification and the comparison for equality have been performed *modulo* A, where A consists only of commutativity axioms for certain operators in an order-sorted signature; that is, we use order-sorted unification with commutative axioms and equality modulo such axioms. We consider all the unifiers, not only the most general ones. Note also that the critical pairs accumulate the substitution instances of the conditions in the two rules, as in [2].

Given a specification $\mathcal{S} = (\Sigma, R \cup A)$, a critical pair $ccp(t, t', c)$ is more general than another critical pair $ccp(u, u', c')$ if there exists an A-match substitution σ such that $t\sigma =_A u$, $t'\sigma =_A u'$ and $c\sigma =_A c'$, where, as before, we assume A to consist of commutativity axioms for some operations or to be otherwise empty.

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 modulo A 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 in order to get disjoint sets of variables appearing in each of them. Under the assumption that the order-sorted equational specification $\mathcal{S}$—with A involving only commutativity axioms—is terminating, then if $CCP(\mathcal{S}) = \emptyset$ we are guaranteed that the specification $\mathcal{S}$ is *confluent*—in the standard sense that if t can be rewritten modulo A to u and v using the rules in $\mathcal{S}$, then u and v can be rewritten modulo A to some w—therefore, each term t has a unique canonical form $t\downarrow_{\mathcal{S}}$. Note that, due to the presence of conditional equations, we can have $CCP(\mathcal{S}) \neq \emptyset$ with $\mathcal{S}$ still confluent, because all the conditions in the critical pairs may be unsatisfiable, but establishing such unsatisfiability may require additional reasoning. More importantly for our purposes, even in the unconditional case we can have $CCP(\mathcal{S}) \neq \emptyset$ with $\mathcal{S}$ *ground*-confluent, that is, confluent for all ground terms. Therefore, assuming termination, $CCP(\mathcal{S}) = \emptyset$ will ensure the confluence and, a fortiori, the ground-confluence of $\mathcal{S}$, but it is only a sufficient condition.

For an order-sorted specification it is not enough to be confluent. The canonical forms should also provide the most complete information possible about the sort of the term. This intuition is captured by our notion of Church-Rosser specifications.

Definition 2 *We call a confluent and terminating order-sorted specification $\mathcal{S}$* Church-Rosser *iff it additionallysatisfies the following* descent *property: for each term t we have $LS(t) \geq LS(t\downarrow_{\mathcal{S}})$. Similarly, we call a ground-confluent and terminating specification $\mathcal{S}$* ground-Church-Rosser *iff for each ground term t we have $LS(t) \geq LS(t\downarrow_{\mathcal{S}})$.*

Note that these notions are more general and flexible than the requirement of confluence and *sort-decreasingness* [19, 12]. In an earlier presentation of our work on the tools to the Cafe Group in March 1997 these notions were called *justice* and *ground-justice*. The issue is how to find checkable conditions for descent that, in addition to the computation of critical pairs, will ensure the Church-Rosser property. This leads us into the topic of specializations.

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 $\rho(X) = (\bar{X}, \mu)$. Note that 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.

The notion of specialization can 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 $(\forall X, l \to r \texttt{ if } c)$ is a rule $(\forall \rho(X), \rho(l) \to \rho(r) \texttt{ if } \rho(c))$ where ρ is a specialization of X.

The checkable conditions that we have to add to the critical pairs to test for the descent property are called membership constraints.

Definition 3 *Let $\mathcal{S}$ be an order-sorted specification whose signature satisfies the assumptions already mentioned. Then, the set of* membership constraints *for a conditional equation $t = t' \texttt{ if } c$ is defined as*

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

A membership constraint $t : s \texttt{ if } c$ is more general than another membership constraint $t' : s' \texttt{ if } c'$ if there exists a substitution σ such that $t\sigma =_A t'$, $s \leq s'$ and $c\sigma =_A c'$; again, we assume A to consist only of commutativity axioms for some operators or to be otherwise empty.

Let $CMC(\mathcal{S})$ denote 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 returns

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

A fundamental result[14] underlying our tool is that the 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 fact, for terminating unconditional specifications this check is a necessary and sufficient condition; however, for conditional such specifications, the check is only a sufficient condition because if the specification has conditional equations we can have unsatisfiable conditions in the critical pairs or in the membership constraints; that is, we can have $\langle\ CCP(\mathcal{S}), CMC(\mathcal{S})\ \rangle \neq \langle\ \emptyset, \emptyset\ \rangle$ with $\mathcal{S}$ still Church-Rosser. Furthermore, even if we assume that the specification is unconditional, since for specifications with an initial algebra semantics we only need to check that $\mathcal{S}$ is ground-Church-Rosser, we may often have specifications that satisfy this property, but for which the tool returns nonempty sets of critical pairs or of membership constraints as proof obligations.

[14] A detailed proof of this result is beyond the scope of this paper and will be presented elsewhere; for related results in membership equational logic see [2].

Of course, in other cases, it may in fact be a matter of some error in the user's 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`.

The input to the tool is given by the specification module we want to check that, as already explained, must be an order-sorted conditional specification with operators involving only the commutativity attribute and, furthermore, it 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.

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 descent 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 descent is carried out by two different functions, `confluenceCheck` and `descCheck`, which will be explained, respectively, in Sections 5.3 and 5.4. The modules passed to these functions are previously *prepared* in order to get the variables needed in the process added. The declarations are then given as follows:

```
sorts CritPair CritPairSet CheckingSolution .
subsort CritPair < 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 -> CritPairSet .
op descCheck : Module -> MembAxSet .

var Th : Module .
eq checking(Th)
  = checkingSolution(
        eliminateDuplicateVarDecl(
```

```
            varDeclSet(
               varDeclSetInCritPairs(prepareModule(Th),
                  confluenceCheck(prepareModule(Th))),
               varDeclSetInMembAxs(prepareModule(Th),
                  descCheck(prepareModule(Th))))),
        confluenceCheck(prepareModule(Th)),
        descCheck(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, it 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 some of the functions 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 .
```

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 descent checking—are created.

```
op prepareModule : Module -> Module .
```

Finally, we need to 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 varDeclSetInCritPairs : Module CritPairSet -> VarDeclSet .
op varDeclSetInMembAxs : Module MembAxSet -> VarDeclSet .
op eliminateDuplicateVarDecl : VarDeclSet -> VarDeclSet .
```

5.2.1 General Operations on Terms

We summarize 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 nonvariable positions in a term and to apply a substitution on a term.

```
sorts Position PositionSet .
subsort Position < PositionSet .

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 .
```

The function `replace` takes terms t and t' and a position p as input and returns the term $t[t']_p$ if $p \in \mathcal{P}(t)$, or `error*` otherwise.

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

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 .
```

The function `substitute` takes a term t and a substitution σ and returns $t\sigma$.

```
op substitute : TermList Substitution -> TermList .
```

Finally, we will define the function `rename`, which takes a list of terms 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 .
```

5.2.2 Order-Sorted Commutative Unification and Matching

In this section we sketch the order-sorted unification algorithm implemented. It yields a complete set of unifiers for a unification problem in which it is assumed that the order-sorted signature of the specification in question can have some operators declared to be *commutative*—in which case, all the remaining subsort-overloaded versions of any such operator should also have been declared commutative—but no other equational axioms have been declared as attributes for the operators. Therefore, the order-sorted unification is performed *modulo* the commutativity of certain operators.

The basic idea is to turn each of the textbook-style inference rules for such a unification algorithm into corresponding (equational) rewrite rules in Maude. The inference rules and their Maude counterparts can be found in [5]; here we only summarize the basic concepts and the top-level functionality. Following [18], we unify using the sort information as soon as possible in order to quickly discard failures. 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.

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

A set of substitutions Φ is a complete set of $\mathcal{S}$-solutions of the equation $t =_A^? t'$ 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 $\alpha \lesssim_A \sigma[vars(t) \cup vars(t')]$.

Note that as mentioned in Section 5.1 we do not compute a minimal set of unifiers. With subsort overloading we assume that an operator symbol has always the same equational attributes—to rewrite modulo such attributes—in all its subsort-overloaded versions; and such equational attributes are restricted to be either the empty set, or the single equation of commutativity.

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 .

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

op unify : Module Term Term -> SubstitutionSet .
```

The specification for order-sorted matching is quite similar to the specification for unification. The main differences come from the fact that now we have to consider noncommutative 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 check for the correct sorting of an equation in solved form $x =_c^? t$ is reduced to comparing 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 order-sorted matching with the commutative axiom, assuming no ad-hoc overloading of function symbols, and that any subsort-overloaded symbols have the same attributes.

```
op match? : Module EquationSet -> Bool .
```

5.3 Confluence

We are now ready to present 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—those of the form $cp(t, t, c)$—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` that reduces 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.

```
  op critPairs : Module -> CritPairSet .
  op delete : CritPairSet -> CritPairSet .
  op simplify : CritPairSet Module -> CritPairSet .
  op maximalCPSet : CritPairSet Module -> CritPairSet .

  vars Th : Module .
  eq confluenceCheck(Th)
    = maximalCPSet(delete(simplify(delete(critPairs(Th)), Th)), Th) .
```

In order to get a 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. It proceeds comparing each critical pair *cp* with all the others; if some critical pair is more general than *cp* then *cp* is removed. As said in Section 5.1, a critical pair $ccp(t, t', c)$ is more general than another one $ccp(u, u', c')$ if there exists a substitution σ such that $t\sigma =_A u$, $t'\sigma =_A u'$ and $c\sigma =_A c'$, with A the commutativity attributes declared for some operators in the specification.

One critical pair is generated for each unifier for each of the possible nonvariable overlappings of the lefthand sides of any two equations in the module. These 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 nonvariable 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`).

To simplify the exposition we present only the top-level functions, and only for the unconditional case. They are easily extended to the case with conditions, just adding equations to handle the different cases. As said above, the critical pair is formed by `critPairs4` for a pair of equations with an overlapping at some position with some substitution. In the cases when some or both of the equations involved is conditional, then the conjunction of the conditions with the substitution applied to them is placed as the condition of the critical pair.

```
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, 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, 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, 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, E, E', P, emptySubstitutionSet)
  = emptyCritPairSet .
```

5.3.1 Nonconfluence of the Natural Numbers

Our first example illustrating the Church-Rosser checker is the very simple natural number specification NAT already presented in Section 4.3.1. This specification is perfectly reasonable. Its initial model is $\mathbb{N}$, with the sum and product functions. However, although it is ground-confluent 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.

The output of the system is given as follows.

```
result CheckingSolution:
   checkingSolution(
      varDeclSet(varDecl('N, 'Nat), varDecl('N@', 'Nat)),
      cp('s['_+_['N, ('_+_['N@', ('_*_['N, 'N@'])])]],
         's['_+_['N@', ('_+_['N, ('_*_['N, 'N@'])])]]),
      emptyMembAxSet)
```

Note that all the variables in the critical pairs, and in the membership constraints (none of the latter in this case) are collected and included as the first component of the output triple.

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 of 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 trivial, and so is eliminated. Reducing each of the terms of the other one in the specification we obtain the critical pair

$$\texttt{cp(}\overline{\texttt{s(N + (N@' + (N * N@')))}}\texttt{, }\overline{\texttt{s(N@' + (N + (N * N@')))}}\texttt{).}$$

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.

In this case the set of membership axioms is empty. This means that the equations are descending. We will study descent checking in Section 5.4, and will return to this example, to make it confluent, in Section 5.5.1.

5.4 Descent

Given a specification $\mathcal{S}$ the function `descCheck` 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 descent 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 descent condition, that is, $LS(r\downarrow) \not\leq LS(l)$, a membership axiom of the form $\texttt{mb(}r\texttt{,}LS(l)\texttt{)}$ is generated. Similarly, for a conditional equation $(\forall X, l = r \texttt{ if } c)$ failing to satisfy the descent condition a *conditional* membership axiom of the form $\texttt{cmb(}r\texttt{,}LS(l)\texttt{,c)}$ is generated.

Comparing the least sort for the sides of each equation is not enough. To check that an equation satisfies the descent condition we have to generate all the possible instances of it obtained for each of the possible specializations of its variables and verify that in each of these instances the least sort of the term in the lefthand side is greater or equal to the least sort of the term in the righthand side.

We do not want all these proof obligations, we only need the most general ones. As said in Section 5.1, a membership constraint $t : s \texttt{ if } c$ is more general than another membership constraint $t' : s' \texttt{ if } c'$ if there exists a substitution σ such that $t\sigma = t'$, $s \leq s'$ and $c\sigma = c'$. The function `maximalMASet` returns the set of maximals in the set of membership axioms that takes as input.

As in the discussion about the specification for the generation of critical pairs, we only consider the unconditional case. However, in this case the way in which the conditional equations are treated is exactly the same one as for unconditional equations.

```
op descCheck1 : Module EquationSet -> MembAxSet .
op eqInstanceSet : Module EquationSet -> EquationSet .
op maximalMASet : MembAxSet Module -> MembAxSet .

var Th : Module .
var ES : EquationSet .
vars T T' : Term .

eq descCheck(Th)
  = maximalMASet(
       descCheck1(Th, eqInstanceSet(Th, equations(Th))), Th) .

eq descCheck1(Th, equationSet(eq(T, T'), ES))
  = if sortLeq(Th, leastSort(Th, meta-reduce(Th, T')),
               leastSort(Th, T))
    then descCheck1(Th, ES)
    else membAxSet(mb(T', leastSort(Th, T)), descCheck1(Th, ES))
    fi .
eq descCheck1(Th, emptyEquationSet)
  = emptyMembAxSet .
```

The checking of the descent condition with the above equations is illustrated in the following section with a module INT for integers.

5.4.1 The Integers Fail Confluence and Descent

Let us continue with the specification of the integers introduced in Section 4.3.2 that illustrates the way in which the descent check is accomplished. As in the NAT example, this specification of INT is perfectly reasonable: it is ground-confluent, and its initial 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.

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

- Descent also fails. The equation

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

gives rise to a membership constraint because 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. To check descent, 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 descent 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.

```
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 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 second critical pair comes from the overlapping at the top of equations

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

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.

We will revisit this example, to find a confluent variant and to establish the ground-Church-Rosser property for it, in Section 5.5.2.

5.5 How to Use the Church-Rosser Checker

This section illustrates with examples the use of the Church-Rosser checker tool, and suggests some methods that—using the feedback provided by the tool—can help the user establish that his/her specification is ground-Church-Rosser. For additional examples see [5].

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 unconditional and 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 nontrivial sort orderings such as, for example, the specifications of the number hierarchy discussed in this section and in [5]—the specification will typically be *ground* Church-Rosser, but not Church-Rosser, so that a collection of critical pairs and of membership constraints will be returned by the tool as proof obligations.

An 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 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 studied 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 a 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

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

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

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

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 Confluent

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 the checker's output, and how we can make the specification confluent. Since no membership assertions are generated by the tool, this shows that the specification is Church-Rosser and a fortiori ground-Church-Rosser.

As indicated in Section 5.3.1 the tool gives as output:

```
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 nonterminating one. The way to handle this with our approach consists in studying 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.

As was seen in Section 5.3.1, only one critical pair is generated by the tool for the specification of the natural numbers presented in Section 2. This 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 ← 's['N@']), ('M@' ← 's['N'])}`. This means that a term of the form `s(N) * s(M)` can be rewritten to terms `s(N + (M + (N * M)))` and `s(M + (N + (N * M)))`. Note that these terms cannot be further reduced, but their ground instances can be reduced; that is, the tool's output 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 in some other way, as, for example,

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

Replacing the original equation by this one in the input to the Church-Rosser checker tool does not return any critical pair. 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, for example,

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

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

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

5.5.2 Proving the Integers Ground-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, so as to make the specification confluent. We will also see how the inductive theorem prover can be used to take care of the membership proof obligation generated. After taking care in this way of all proof obligations we establish the ground-Church-Rosser property.

In Section 5.4.1 we saw 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. Therefore, 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 here. 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)) .
```

Calling the checker with these equations in the input module instead of the previous ones no critical pair is given in the output.

Regarding descent, 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 ground-decreasing. In this case, we have to prove `INT` $\vdash_{\mathrm{ind}}$ `(∀I) I * I: Nat`. This can be done using the theorem prover presented in Section 4, as was shown in [5]. Therefore, we have successfully transformed our original `INT` specification into one that is confluent and ground-descending, and therefore ground-Church-Rosser.

6 Concluding Remarks

Our experience in building and using the two tools that we have described has been very encouraging. There are several natural extensions of these tools that we plan to pursue in the near future. The inductive theorem prover at present can only handle *atomic* sentences. Its inference rules should be extended to handle Horn clauses, conjunctions of such clauses, and also general clauses. Furthermore, the underlying logic should be extended from equational to rewriting logic, so as to be able to prove theorems of rewriting logic specifications in Cafe and Maude. Yet another promising direction for extension is supporting a mixture of the current explicit induction methods and the automated induction techniques for membership equational logic proposed in [2, 3].

Similarly, the Church-Rosser checker should be extended in several directions. First, the set of equational attributes used to rewrite modulo should be extended to contain other axioms besides commutativity. Furthermore, using the results in [2, 3], the tool should be generalized so as to deal with membership equational logic specifications and not just order-sorted specifications. Completion should also be added as an additional facility, perhaps integrating CiME as an auxiliary termination tool. Finally, this tool should also be extended from equational logic to rewriting logic, to check *coherence* [27] of rewrite theories, or to complete such theories so that they become coherent.

More broadly, the present experience suggests that the reflective approach that we have taken in building the present tools is a promising general methodology to build many other theorem proving and formal analysis tools based on formal systems for different logics. For example, a mechanization of linear logic in Maude has already been demonstrated [4]; and one could also build in this way a tool supporting temporal logic reasoning about nonexecutable properties of Maude modules or of systems defined in other languages. An important added benefit of building a formal environment of tools this way is that, since each tool is a theory in rewriting logic, they are much easier to interoperate by just combining their corresponding rewrite theories. We have for example seen the usefulness of using the inductive theorem prover to prove proof obligations generated by the Church-Rosser checker. In general, using the reflective techniques and the flexible logical framework capabilities of rewriting logic, we hope to make good advances towards the goal of *formal interoperability* [24], that is, the capacity to move in a mathematically rigorous way across different formalizations, and to use in a rigorously integrated way the different tools supporting such formalizations.

References

[1] J. Bergstra and J. Tucker. Characterization of computable data types by means of a finite equational specification method. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, Seventh Colloquium*, pages 76–90. Springer-Verlag, 1980. LNCS, Volume 81.

[2] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. Manuscript, SRI International, November 1997, submitted for publication.

[3] 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, volume 1214 of *Lecture Notes in Computer Science*, pages 67–92. Springer-Verlag, 1997.

[4] M. Clavel. Reflection in general logics and in rewriting logic with applications to the Maude language. Ph.D. Thesis, University of Navarre, 1998.

[5] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Design and implementation of the Cafe prover and Church-Rosser checker tools. Technical report, SRI International, December 1997.

[6] M. Clavel, F. Durán, S. Eker, J. Meseguer, and P. Lincoln. An introduction to Maude (beta version). Manuscript, SRI International, March 1998.

[7] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications, RWLW'96, Asilomar, California*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 1996.

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

[9] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications, RWLW'96, Asilomar, California*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 1996.

[10] E. Contejean and C. Marché. The CiME system: Tutorial and user's manual. Manuscript, Université Paris-Sud, Centre d'Orsay.

[11] K. Futatsugi and T. Sawada. Cafe as an extensible specification environment. In *Proc. of the Kunming International CASE Symposium, Kunming, China, November*, 1994.

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

[13] 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.

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

[15] 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.

[16] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, SRI International, Computer Science Laboratory, 1992. To appear in J.A. Goguen and G.R. Malcolm, editors, *Applications of Algebraic Specification Using OBJ*, Academic Press, 1998.

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

[18] 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.

[19] 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.

[20] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications, RWLW'96, Asilomar, California*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 1996.

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

[22] 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.

[23] J. Meseguer. Rewriting logic as a semantic framework for concurrency: A progress report. In U. Montanari and V. Sassone, editors, *Proc. 7th Int. Conf. on Concurrency Theory, CONCUR'96, Pisa, Italy, August 1996*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372. Springer-Verlag, 1996.

[24] J. Meseguer. Formal interoperability. In *Proceedings of the 1998 Conference on Mathematics in Artificial Intelligence, Fort Laurerdale, Florida, January 1998*, 1998. `http://rutcor.rutgers.edu/~amai/Proceedings.html`.

[25] J. Meseguer. Membership algebra as a semantic framework for equational specification. in F. Parisi-Presicce, ed., Proc. WADT'97, Springer LNCS 1376, 1998.

[26] 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.

[27] P. Viry. Rewriting: An effective model of concurrency. In C. Halatsis et al., editors, *PARLE'94, Proc. Sixth Int. Conf. on Parallel Architectures and Languages Europe, Athens, Greece, July 1994*, volume 817 of *LNCS*, pages 648–660. Springer-Verlag, 1994.