Termination Checker and Knuth-Bendix Completion Tools for Maude Equational Specifications

Francisco Durán
E.T.S.I. Informática
Universidad de Málaga
Málaga, Spain
duran@lcc.uma.es

July 28, 2000

Abstract

This document explains the design and use of a termination checker tool and of a Knuth-Bendix completion tool. The termination checker tool checks whether an equational specification terminates, and the Knuth-Bendix completion tool tries to complete an equational specification. These tools can be used to prove the termination or to complete order-sorted equational specifications in Maude [7, 6, 4]. The tools have been written entirely in Maude and are in fact executable specifications in rewriting logic [17] of the formal inference system that they implement. The fact that rewriting logic is reflective [8, 3], and that Maude efficiently supports reflective rewriting logic computations [5, 4] is systematically exploited in the design of the tools.

Contents

1 Introduction

This document is concerned with the question of how to construct a termination checker for Maude equational specifications and on how to construct a decision procedure for the word problem for a given finite set of identities E. Because of the word problem is in general undecidable, any such method is necessarily incomplete, that is, it cannot succeed for all finite sets E.

A rewrite system R is terminating for a set of terms $\mathcal{T}$ if there are no infinite derivations of terms in $\mathcal{T}$. Obviously, if $\longrightarrow_R$ is contained in some well-founded partial ordering on $\mathcal{T}$, then R is terminating. In general, it is undecidable whether a system is terminating, although termination is decidable for finite ground systems [15].

A term rewriting system R terminates if and only if there exists a reduction order $\succ$ that satisfies $l \succ r$ for all $l \to r$ in R. In the case of conditional rewrite systems, a conditional rule $l \to r$ *if* c is terminating iff $l \succ r$ and $l \succ c$.

As suggested by Knuth and Bendix [16], we can use the critical pairs generated automatically by a superposition test, as for example in [11], in order to extend the system with new rules. These new rules can give place to new critical pairs, which will then give place to new rules, and so on. Hopefully, this process eventually leads to a convergent system, which does not generate any further nonconvergent critical pair and thus the process terminates. This process is called *Knuth-Bendix completion*, or simply *completion*.

The completion process is as follows. Starting with a finite set of equations and a reduction order on terms, the completion procedure attempts to find a finite canonical system for the theory presented by the equations by generating critical pairs and orienting them as necessary. If reducing the two sides of a critical pair $s = t$ yields and equation $u = v$, where u and v are not identical, then adding a new rule $u \to v$ or $v \to u$ supplies a rewrite proof for $s = t$. To decide between the two orientations, a given reduction ordering $\succ$ is employed: if $u \succ v$ then $u \to v$ is added, while if $v \succ u$ then $v \to u$ is chosen. If neither $u \succ v$ nor $v \succ u$ then the system cannot be completed with such an order and the process terminates. The procedure can then terminate with either of the following three outcomes: success in finding a canonical system, giving such a canonical system as result; failure in finding the canonical system, because there is a critical pair which cannot be oriented; or nonterminating execution, generating an infinite number of rules.

The present tools only accept 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 do not contain any built-in sorts or functions. The tools attempt to check termination and to complete *modulo* the commutativity of some of the operators.

We explain in Section 2 the design and specification of the termination checker tool. In Section 3.1 we give an abstract description of the completion procedure, following [2], by giving a set of inference rules. Then, in Section 3.2 we describe Huet's completion procedure, which is the one implemented and discussed in Section 3.3. After introducing some basic formal concepts and results underlying the tools' design, and after discussing several auxiliary functions used by the tools, we explain key parts of the tools and then give some examples. We present the different modules with their main declarations, so that a general idea of the whole system can be gained, omitting declarations of operators not relevant for the general discussion, as well as equations describing the semantics of some functions. We add dots "`...`" to indicate the missing equations and declarations.

We will see in these sections how the termination checker and the completion procedure are parameterized by a reduction order. By using the reflective capabilities of the Maude system, and thanks to the reflective design of the tool, the user is able to choose the reduction order to be used among a set of available orders, but, more importantly, the tools also allow us to specify our own reduction orders at will, and to use them in a very simple way. Section 5 discusses two particular reduction orders, namely, the lexicographic path order and the Knuth-Bendix order. However, we shall see in some of the examples illustrating the use of the tools in Section 6 how new reduction orders can easily be specified by the user.

Acknowledgements

The author is grateful to José Meseguer for his advice and collaboration in the long-term joint project of which this work is a part. We are very thankful for the many fruitful discussions with the other members of the Maude team, Manuel Clavel, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, and José Quesada, with whom we have designed and built the Maude language and system.

2 A Termination Checker Tool

Termination is an important property of term rewriting systems. And, as we have seen above, reduction orders are an important tool for proving termination of rewrite systems. In fact, the main problem for a given rewrite system is to find an appropriate reduction order that shows its termination. This is the reason why it is so important to be able to have a wide range of different possible reduction orders available, and even to be able to define new ones.

The termination checker is parameterized by a reduction order. Since the current version of Maude does not support parameterized modules, and since Full Maude cannot be used to specify the tool because we wish to build an execution environment for it, we are forced to simulate this parameterization. What we do in the following is to define a module `ORDER` in which we define the `_>>>_` operator, but without giving any equations for it. This operator is then used in the `TERMINATION` module as defining the reduction order. Then, we shall see in Section 4 how we can combine the module defining a particular reduction order with this specification, as if we were instantiating the `TERMINATION` module with a view from `ORDER` to such a module.

```
fmod ORDER is
  pr META-LEVEL .
  op _>>>_ : Term Term -> Bool .
endfm
```

The function `check-termination` in the following module `TERMINATION` is then as follows. It checks the termination of a set of equations by checking the termination condition on each of the equations. If all the equations are terminating then it returns `ok`; otherwise it returns `fail` with the equation that failed the check.

```
fmod TERMINATION is
  pr ORDER .

  sort TerminationCheckSol .
  op fail : Equation -> TerminationCheckSol .
  op ok : -> TerminationCheckSol .

  op check-termination : EquationSet -> TerminationCheckSol .

  vars T T' T'' T''' : Term .
  var EqS : EquationSet .

  eq check-termination((eq T = T' .) EqS)
    = if T >>> T'
      then check-termination(EqS)
      else fail((eq T = T' .))
      fi .
  eq check-termination((ceq T = T' if T'' = T''' .) EqS)
    = if T >>> T' and T >>> T'' and T >>> T'''
      then check-termination(EqS)
      else fail((ceq T = T' if T'' = T''' .))
      fi .
  eq check-termination(none) = ok .
endfm
```

3 A Knuth-Bendix Completion Tool

3.1 Abstract Completion

Following [2], we describe completion by a set of inference rules. A specific completion procedure can then be obtained from this set of rules by fixing a strategy for rule application.

The inference system given in Figure 3.1 work on pairs $(E; R)$, where E is a finite set of identities (unoriented equations) and R is a finite set of (oriented) rewrite rules. Intuitively, E contains input identities or critical pairs that have not yet been transformed into rules, whereas R is a terminating set of rewriting rules. Let $\succ$ be a reduction ordering on terms, which is given as an input to the completion procedure and ensures termination

$$
\begin{array}{rll}
\textbf{Delete} & \dfrac{(E \cup \{t = t\}; R)}{(E; R)} & \\[2ex]
\textbf{Compose} & \dfrac{(E; R \cup \{s \to t\})}{(E; R \cup \{s \to u\})} & \text{if } t \to_R u \\[2ex]
\textbf{Simplify} & \dfrac{(E \cup \{s = t\}; R)}{(E \cup \{u = t\}; R)} & \text{if } s \to_R u \\[2ex]
\textbf{Orient} & \dfrac{(E \cup \{s = t\}; R)}{(E; R \cup \{s \to t\})} & \text{if } s \succ t \\[2ex]
\textbf{Collapse} & \dfrac{(E; R \cup \{s \to t\})}{(E \cup \{u = t\}; R)} & \text{if } s \stackrel{\sqsupset}{\to}_R u \\[2ex]
\textbf{Deduce} & \dfrac{(E; R)}{(E; R \cup \{s \to t\})} & \text{if } s = t \in \mathrm{cp}(R)
\end{array}
$$

Figure 1: Inference rules for completion.

of R. The goal is to transform an initial pair $(E_0; \emptyset)$ into a pair $(\emptyset, R)$ such that R is convergent and equivalent to E_0.

The rule **Delete** removes a trivial equation $t = t$. **Simplify** uses R to reduce identities. Taken together, **Simplify** and **Delete** can be used to remove joinable critical pairs. **Compose** reduces the right-hand side of a rule. **Orient** takes an identity $s = t$ that is orientable with the help of $\succ$ and adds the corresponding rule to R. **Collapse** rewrites the left-hand side of a rule $s \to t$ and turns the result into an equation $u = t$, but only when the rule being applied to s is smaller than the rule being removed. The notation $s \stackrel{\sqsupset}{\to}_R u$ is used to express that s is reduced by a rule $l \to r \in R$ such that l cannot be reduced by $s \to t$. The symbol $\sqsupset$ stands for the strict part of the *encompassment* quasi-order. The rule **Deduce** derives an identity that follows from a critical pair of rules in R, and adds this identity to E.

We saw in [11] how to compute the critical pairs in the conditional case. For completion, conditions are treated in a straightforward way, just keeping them all along the process. The inference rules for completion in the conditional case are given in Figure 3.1. Note that the **Orient** rule orients a critical pair taking into account not only the left and righthand sides of such a critical pair, but also its condition.

Delete	$\dfrac{(E \cup \{t = t \ \mathit{if} \ c\}; R)}{(E; R)}$	
Compose	$\dfrac{(E; R \cup \{s \to t \ \mathit{if} \ c\})}{(E; R \cup \{s \to u \ \mathit{if} \ c\})}$	if $t \to_R u$
Simplify	$\dfrac{(E \cup \{s = t \ \mathit{if} \ c\}; R)}{(E \cup \{u = t \ \mathit{if} \ c\}; R)}$	if $s \to_R u$
Orient	$\dfrac{(E \cup \{s = t \ \mathit{if} \ c\}; R)}{(E; R \cup \{s \to t \ \mathit{if} \ c\})}$	if $s \succ t$ and $s \succ c$
Collapse	$\dfrac{(E; R \cup \{s \to t \ \mathit{if} \ c\})}{(E \cup \{u = t \ \mathit{if} \ c\}; R)}$	if $s \mathrel{\sqsupset\!\!\!\to}_R u$
Deduce	$\dfrac{(E; R)}{(E; R \cup \{s \to t \ \mathit{if} \ c\})}$	if $s = t \ \mathit{if} \ c \in \mathrm{cp}(R)$

Figure 2: Inference rules for completion in the conditional case.

3.2 Huet's Completion Procedure

In the present tool we have implemented Huet's completion procedure, which can be seen in Figure 3.2 as given in [1]. This procedure takes as input a finite set E of Σ-identities and a reduction order $\succ$ on $T(\Sigma, V)$, and gives as output, either a finite convergent term rewrite system R_i that is equivalent to E, if the procedure terminates successfully, or `fail`, if the procedure terminates unsuccessfully. If the procedure does not terminate, it generates an infinite limit system R_w that is convergent and equivalent to E. Huet was the first to give a complete proof of correctness of a completion procedure that allows for simplification of rules [14].

Like the inference rules for completion, Huet's procedure works on a set E of identities and a terminating set R of rewrite rules. In order to keep track of whether a rule has already been used for computing critical pairs, rules are *marked* or not.

3.3 The Maude Specification of Huet's Completion Procedure

In this section, we present the Maude specification of the Huet's completion procedure, which follows quite closely the algorithm given in Figure 3.2. In this specification, instead of considering marked rules, we maintain two separate sets of rules, to keep the marked and unmarked rules separated.

```
R_0 := ∅; E_0 := E; i := 0;
while E_i ≠ ∅ or there is an unmarked rule in R_i do
    while E_i ≠ ∅ do
        Choose an identity s = t ∈ E_i;
        Reduce s, t to some R_i-normal forms ŝ, t̂;
        If ŝ = t̂, then
            R_{i+1} := R_i;
            E_{i+1} := E_i − {s = t};
            i := i + 1;
        Otherwise, if ŝ ⊁ t̂ and t̂ ⊁ ŝ, then terminate with output Fail;
        Otherwise, let l, r be such that {l, r} = {ŝ, t̂} and l ≻ r;
            R_{i+1} := {g → d̂ | g → d ∈ R_i, g cannot be reduced
                              with l → r, and d̂ is a normal
                              form of d w.r.t. R_i ∪ {l → r}}
                       ∪ {l → r};
                       (* g → d̂ inherits the marker of g → d *)
                       (* l → r is not marked *)
            E_{i+1} := (E_i − {s = t}) ∪
                       {g' = d | g → d ∈ R_i and g can be
                              reduced to g' with l → r};
            i := i + 1;
    end while
    If there is an unmarked rule in R_i, then let l → r be such a rule.
        R_{i+1} := R_i;
        E_{i+1} := {s = t | ⟨s, t⟩ is a critical pair of l → r with
                       itself or a marked rule in R_i};
        i := i + 1;
        Mark l → r;
end while
return R_i;
```

Figure 3: Huet's completion procedure.

The tool has a reflective design. The completion process itself is at the metalevel of the specification being completed, as is the order given to orient the critical pairs. These two modules are then combined at their metalevel, that is, the meta-metalevel of the specification to be completed, where these modules can be handled as data and therefore can be put together. The environment of the tool is also given at this higher level of reflection, providing a user-friendly interface to the tool, perfectly integrated with Full Maude in the same way in which the Church-Rosser checker is [11].

3.3.1 Generation of Critical Pairs

The generation of critical pairs is specified in the following `CRITICAL-PAIRS` module, which we do not comment on in much detail since it follows quite closely the specification given in [11]. Note that, as explained in [11], the generation of critical pairs takes place in an order-sorted and conditional context. Therefore, unification is order-sorted (modulo commutativity) and the critical pairs may be conditional. There are however some significant differences with [11], mainly due to the way in which the critical pairs generated are used. In the Church-Rosser checker tool we generate all the critical pairs for a given set of oriented equations, while in the Huet procedure we wish to obtain the critical pairs of a set of oriented equations with those in another set. This was accomplished by the `critPairs2` function, and although its specification has slightly changed, we have kept that name for it. Another significant difference is that, since here the critical pairs are oriented and are considered then as oriented equations, we do not use the sort `CritPair`; we use instead the sort `Equation` and its constructors to represent the critical pairs. The renaming of the equations is also different. Since the process is repeated several times, we do not know beforehand the variables that we are going to need, and therefore, we must rename the variables in the equations involved each time the function `critPairs2` is called, adding the new variables to the set of variables in the module.

```
fmod CRITICAL-PAIRS is
  pr UNIFICATION .
  pr MATCHING .
  pr EXT-TERM .
  pr TUPLE`(2`)`[VarDeclSet`,EquationSet`] .
  pr TUPLE`(2`)`[VarDeclSet`,TermList`] .

  var M : Module .
  vars Eq Eq' : Equation .
```

```
vars EqS EqS' : EquationSet .

op critPairs2 : Module EquationSet EquationSet
     -> Tuple`[VarDeclSet`,EquationSet`] .
op critPairs3 : Module Equation Equation PositionSet
     -> EquationSet .
op critPairs4 : Module Equation Equation Position SubstitutionSet
     -> EquationSet .
```

Variables are renamed using the function `rename` following a very simple scheme: A variable `X` is renamed as `X@@...@` where `@@...@` is the minimum number of `@`'s such that `X@@...@` is not used in the second equation. The function `rename` returns a tuple with the new set of variables and the equation.

Thus, `rename` takes as arguments the set of variables available in the module, the set of variables that cannot be used, that is, those in the other equations, and the equation to be renamed, and gives as result a tuple with a set of variables in which we have all the variables in the original module plus those newly created, and the resulting equation.

```
op rename : VarDeclSet VarDeclSet Equation
      -> Tuple`[VarDeclSet`,Equation`] .
```

The function `varDeclSetInEquation` returns the set of declarations of the variables in an equation.

```
op varDeclSetInEquation : VarDeclSet Equation -> VarDeclSet .
```

`lhs` returns the lefthand side of the equation given as argument.

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

Then, the specification of the `critPairs2` is as follows.

```
ceq critPairs2(M, (Eq EqS), EqS')
  = ((p1 critPairs2(M, Eq, EqS')) (p1 critPairs2(M, EqS, EqS')),
     (p2 critPairs2(M, Eq, EqS')) (p2 critPairs2(M, EqS, EqS')))
    if EqS =/= none .
eq critPairs2(M, Eq, (Eq' EqS))
  = (((p1 rename(varDeclSet(M),
            varDeclSetInEquation(varDeclSet(M), Eq'), Eq))
      (p1 critPairs2(M, Eq, EqS))),
     (p2 critPairs2(M, Eq, EqS))
     critPairs3(
```

```
        setVarDeclSet(M,
           p1 rename(varDeclSet(M),
                 varDeclSetInEquation(varDeclSet(M), Eq'), Eq)),
        p2 rename(varDeclSet(M),
              varDeclSetInEquation(varDeclSet(M), Eq'), Eq),
        Eq',
        allNonVarPos(M, lhs(Eq)))) .
  eq critPairs2(M, EqS, none) = (varDeclSet(M), none) .
  eq critPairs2(M, none, EqS) = (varDeclSet(M), none) .
```

The functions `critPairs3` and `critPairs4` are as in [11], although, as mentioned above, the critical pairs generated are given as elements of sort `Equation`.

```
  ...
endfm
```

3.3.2 The Completion Procedure

The following specification follows quite closely Huet's completion procedure algorithm given in Figure 3.2. As mentioned above, instead of considering marked rules, we maintain two separate sets of rules, in which we keep the marked and the unmarked ones separated. As in the module `TERMINATION` of Section 2, the module `ORDER`, with an operator `_>>>_` declared but with no equations for it, is included. The module `COMPLETION` and the one provided by the user will be combined as explained in Section 4.

```
fmod COMPLETION is
  pr ORDER .
  pr UNIT .
  pr CRITICAL-PAIRS .
  pr TUPLE`(2`)`[EquationSet`,EquationSet`] .
  pr TUPLE`(3`)`[EquationSet`,EquationSet`,EquationSet`] .

  var M : Module .
  vars EqS RlS RlS' : EquationSet .
  vars T T' T'' T''' : Term .
  var TL : TermList .
  vars C S F X Y : Qid .
  vars Rl Rl' Eq : Equation .
  var N : MachineInt .
  var VDS : VarDeclSet .
  var SS : QidSet .
```

If the completion process terminates, the solution can be either `fail` or a module with a confluent set of equations.

```
sort CompletionSolution .
subsort Module < CompletionSolution .
op fail : Equation -> CompletionSolution .
```

The main function in this module is `complete`, which takes a functional module as input and returns the result of completing it.

```
op complete : Module -> CompletionSolution .
```

At the beginning, the equations are removed from the module to be completed. We are considering just equations, which initially are supposed to be unoriented. Once they are oriented they can be unmarked or marked. We distinguish the ones from the others by keeping two separate sets. Thus, the function `complete3` has four arguments, namely, the module, the set of unoriented equations or critical pairs, unmarked oriented equations, and marked oriented equations.

```
op complete3 : Module EquationSet EquationSet EquationSet
      -> CompletionSolution .

eq complete(M)
  = complete3(setEquationSet(M, none), equationSet(M), none, none) .
```

The `complete1` function takes as argument the result of a call to `orient`. If `orient` fails in orienting some of the critical pairs then `complete1` also fails. If the set of unmarked rules is empty then the process terminates with the set of marked rules as the rule set of the resulting module. Otherwise, one of the unmarked rules is chosen with the `choose` function before calling `complete2`. `complete2` calculates all the critical pairs of the chosen rule with itself and with the set of marked rules, and then calls `complete3`.

The orient function takes an equation from the set of equations and includes it in the set of unmarked rules after orienting it in accordance with the given order. We do not give here the equations for the conditional case. `addRule` adds a new rule to the set of unmarked rules and uses it to simplify all other rules.

```
sort OrientSolution .
subsort Tuple`[EquationSet`,EquationSet`] < OrientSolution .
op fail : Equation -> OrientSolution .
```

```
op orient : Module Tuple`[EquationSet`,EquationSet`,EquationSet`]
      -> OrientSolution .
op simplify : Module Equation EquationSet EquationSet EquationSet
      -> Tuple`[EquationSet`,EquationSet`] .
op addRule : Module Equation EquationSet EquationSet EquationSet
      -> Tuple`[EquationSet`,EquationSet`,EquationSet`] .
op complete1 : Module OrientSolution -> CompletionSolution .
op complete2 : Module Tuple`[Equation`,EquationSet`] EquationSet
      -> CompletionSolution .

eq complete3(M, EqS, RlS, RlS')
  = complete1(M, orient(M, (EqS, RlS, RlS'))) .

eq orient(M, (none, RlS, RlS')) = (RlS, RlS') .
eq orient(M, ((eq T = T' .) EqS, RlS, RlS'))
  = if meta-reduce(setEquationSet(M, RlS RlS'), T)
       == meta-reduce(setEquationSet(M, RlS RlS'), T')
    then orient(M, (EqS, RlS, RlS'))
    else if meta-reduce(setEquationSet(M, RlS RlS'), T)
            >>> meta-reduce(setEquationSet(M, RlS RlS'), T')
         then orient(M,
                addRule(M,
                  (eq meta-reduce(setEquationSet(M, RlS RlS'), T)
                     = meta-reduce(
                         setEquationSet(M, RlS RlS'), T') .),
                  EqS, RlS, RlS'))
         else if meta-reduce(setEquationSet(M, RlS RlS'), T')
                 >>> meta-reduce(setEquationSet(M, RlS RlS'), T)
              then orient(M,
                     (addRule(M,
                       (eq meta-reduce(
                             setEquationSet(M, RlS RlS'), T')
                          = meta-reduce(
                              setEquationSet(M, RlS RlS'), T) .),
                       EqS, RlS, RlS')))
              else fail(eq T = T' .)
              fi
         fi
    fi .

eq simplify(M, Rl, none, EqS, RlS) = (EqS, RlS) .
eq simplify(M, Rl, (eq T = T' .) RlS, EqS, RlS')
  = if meta-reduce(setEquationSet(M, Rl), T) == T
    then simplify(M, Rl, RlS, EqS,
```

```
          (eq T = meta-reduce(
                   setEquationSet(M, Rl RlS RlS'), T') .) RlS')
    else simplify(M, Rl, RlS,
          (eq meta-reduce(setEquationSet(M, Rl), T) = T' .) EqS,
          RlS')
    fi .

eq addRule(M, Rl, EqS, RlS, RlS')
  = (p1 simplify(M, Rl, RlS', p1 simplify(M, Rl, RlS, EqS, none),
          none),
     Rl p2 simplify(M, Rl, RlS, EqS, none),
     p2 simplify(M, Rl, RlS', p1 simplify(M, Rl, RlS, EqS, none),
          none)) .

eq complete1(M, fail(Eq)) = fail(Eq) .
eq complete1(M, (none, RlS'))
  = setEquationSet(
      setVarDeclSet(M, varDeclSetInEquationSet(M, RlS')),
      RlS') .
eq complete1(M, (Rl RlS, RlS'))
  = complete2(M, choose(Rl RlS), RlS') .

eq complete2(M, (Rl, RlS), RlS')
  = complete3(
      setVarDeclSet(M,
         (p1 critPairs2(prepareModule(M), Rl, RlS'))
         (p1 critPairs2(prepareModule(M), RlS', Rl))
         (p1 critPairs2(prepareModule(M), Rl, Rl))),
      (p2 critPairs2(prepareModule(M), Rl, RlS')
       p2 critPairs2(prepareModule(M), RlS', Rl)
       p2 critPairs2(prepareModule(M), Rl, Rl)),
      RlS, Rl RlS') .
```

As in [1], the choice of the next unmarked oriented equation in `complete` is guided by the heuristic that smaller rules are more helpful than larger ones because they have more simplifying power. The size of a term is computed by the function `size`, and `minRule` splits a set of rules into one of least size and the remaining rules. Note that `choose` assumes a nonempty set of equations as input.

```
op choose : EquationSet -> Tuple`[Equation`,EquationSet`] .
op minRule : Equation MachineInt EquationSet EquationSet
     -> Tuple`[Equation`,EquationSet`] .
op size : Equation -> MachineInt .
```

```
  op size : Term -> MachineInt .
  op size : TermList -> MachineInt .

  eq choose(Rl RlS) = minRule(Rl, size(Rl), RlS, none) .

  eq minRule(Rl, N, none, RlS) = (Rl, RlS) .
  eq minRule(Rl, N, Rl' RlS, RlS')
    = if size(Rl') < N
      then minRule(Rl', size(Rl'), RlS, Rl RlS')
      else minRule(Rl, N, RlS, Rl' RlS')
      fi .

  eq size(eq T = T' .) = size(T) + size(T') .
  eq size(ceq T = T' if T'' = T''' .) = size(T) + size(T') .
  eq size(X) = 1 .
  eq size({C}S) = 1 .
  eq size(F[TL]) = 1 + size(TL) .
  eq size((T, TL)) = size(T) + size(TL) .

  ...
endfm
```

4 The Environment of the Tools

To simplify the implementation of an environment for the termination checker and the Knuth-Bendix completion tools, as we already did for the Church-Rosser checker, we have built it on top of Full Maude. As explained in [10], and illustrated in [10, 11], extensions to Full Maude often require extensions to the signature in which the input will be parsed. In the case of the termination checker and the Knuth-Bendix completion tools, we have added declarations for two new commands, `check termination_with_.` and `complete_with_.`, to check the termination of a given module given a module specifying a reduction order, and to complete a module with a given module specifying a reduction order for it, respectively.

```
fmod KBC-SIGN is
  including FULL-MAUDE-SIGN .

  op complete_with_. : ModExp ModExp -> PreCommand .
  op check termination_with_. : ModExp ModExp -> PreCommand .

endfm
```

We extend the persistent database of Full Maude by defining a subclass of `DatabaseClass` inheriting its behavior, and adding new functionalities to it. In this case, we only want to add the new commands `check termination_with_.` and `complete_with_..` It is enough to add the declarations making the class `KBC` a subclass of `DatabaseClass`, and the corresponding rule describing the behavior associated with the new command.

```
mod KBC-DATABASE-HANDLING is
  pr DATABASE-HANDLING .
  pr UNIT .

  vars ME ME' : ModExp .
  var DB : Database .
  var M : Module .
  var VDS : VarDeclSet .
  var MAS : MembAxSet .
  var MN : ModName .
  var QIL : QidList .
  vars T T' : Term .
  var TL : TermList .

  sort KBC .
  subsort KBC < DatabaseClass .

  op KBC : -> KBC .

  var Atts : AttributeSet .
  var X@KBC : KBC .
  var O : Oid .
```

The processing of the `check termination_with_.` command consists in executing the `check-termination` function in the module `'TERMINATION` (see Section 2) in a module including both the module giving the reduction order and the `'TERMINATION` module.

```
  op processTermination : ModExp ModExp Database -> QidList .
  op processTerminationAux : Term -> QidList .

  eq processTermination(ME, ME', DB)
    = ('\n 'Termination 'check 'of my-meta-pretty-print(ME)
       processTerminationAux(
          meta-reduce(
             addImportList((including 'TERMINATION .),
                getFlatUnit(ME', evalModExp(ME', DB))),
```

```
          'check-termination[
             up(equationSet(getFlatUnit(ME, evalModExp(ME, DB))))]))) .

  eq processTerminationAux(error*)
    = ('\n 'An 'error 'has 'occurred 'during 'the 'termination 'check.) .
  eq processTerminationAux('fail[T])
    = ('\n 'The 'specification 'is 'nonterminating.
       '\n 'The 'following 'equation 'fails 'the 'check:
       '\n '\t
       my-meta-pretty-print(
          addImportList((including 'META-LEVEL .), emptyFModule('FOO)), T)) .
  eq processTerminationAux({'ok}'TerminationCheckSol)
    = ('\n 'The 'specification 'is 'terminating.) .
```

The processing of the `complete_with_.` command consists in executing the `complete` function in the module `COMPLETION` (see Section 3.3.2) in a module including both the module giving the reduction order and the `COMPLETION` module. Since we do not take into account the sortdecreasingness of the equations when orienting them, after completing the specifications, we check the descent property (see citeDuranMeseguer00a,Duran99). If they are not sortdecreasing, then the corresponding set of proof obligations is given as output of the tool. We use the `descentCheck` function of [11, Section 3.3] to check the descent property on the specification given by the completion process.

```
  op processKBC : ModExp ModExp Database -> QidList .
  op processKBCAux : Term -> QidList .

  eq processKBC(ME, ME', DB)
    = ('\n 'Knuth-Bendix 'completion 'of my-meta-pretty-print(ME)
       processKBCAux(
          meta-reduce(
             addImportList((including 'COMPLETION .),
                getFlatUnit(ME', evalModExp(ME', DB))),
             'complete[up(getFlatUnit(ME, evalModExp(ME, DB)))]))) .
  eq processKBCAux('fmod_is_______endfm[TL])
    = if meta-reduce(
            addImportList((including 'CRC .), emptyFModule('FOO)),
            'descentCheck['prepareModule['fmod_is_______endfm[TL]]])
         == {'none}'MembAxSet
      then ('\n 'Completion 'solution: '\n
            my-meta-pretty-print(
               downModule('fmod_is_______endfm[TL]),
               downModule('fmod_is_______endfm[TL]))
```

```
         'The 'equations 'in 'the 'resulting 'module 'are
         'sort 'decreasing)
   else ('\n 'Completion 'solution: '\n
         my-meta-pretty-print(
            downModule('fmod_is_______endfm[TL]),
            downModule('fmod_is_______endfm[TL]))
         'Proof 'obligations:
         my-meta-pretty-print(
            addVarDeclSet(
               downVarDeclSet(
                  meta-reduce(
                     addImportList((including 'CRC .),
                        emptyFModule('FOO)),
                     'varDeclSetInMembAxs[
                        'prepareModule['fmod_is_______endfm[TL]],
                        'descentCheck[
                           'prepareModule['fmod_is_______endfm[TL]]]])),
               downModule('fmod_is_______endfm[TL])),
            downMembAxSet(
               meta-reduce(
                 addImportList((including 'CRC .),
                    emptyFModule('FOO)),
                 'descentCheck['prepareModule['fmod_is_______endfm[TL]]]))))
   fi .
```

The rules for handling the new commands are as follows.

```
  rl [KBC] :
     < O : X@KBC | db : DB, input : ('complete_with_.[T, T']),
        output : QIL, default : MN, Atts >
     => < O : X@KBC | db : DB, input : nilTermList,
           output : processKBC(parseModExp(T), parseModExp(T'), DB),
           default : MN, Atts > .
  rl [KBC] :
     < O : X@KBC | db : DB, input : ('check`termination_with_.[T, T']),
        output : QIL, default : MN, Atts >
     => < O : X@KBC | db : DB, input : nilTermList,
           output : processTermination(parseModExp(T), parseModExp(T'), DB),
           default : MN, Atts > .
endm
```

As in [11], we define a new constant KBC-GRAMMAR, of sort FModule. This constant is identified with the metarepresentation of the module to be used in the parsing process, which extends the module identified with EXT-GRAMMAR.

```
fmod META-KBC-SIGN is
  inc META-LEVEL .
  pr META-FULL-MAUDE-SIGN .

  op KBC-GRAMMAR : -> FModule .

  eq KBC-GRAMMAR
    = addImportList((including 'KBC-SIGN .), EXT-GRAMMAR) .
endfm
```

We finish our specification redefining the `FULL-MAUDE` module.

```
mod KBC-FULL-MAUDE is
  pr META-KBC-SIGN .
  pr KBC-DATABASE-HANDLING .
  pr PREDEF-UNITS .
  inc LOOP-MODE .

  subsort Object < State .

  op kbc : -> Oid .
  op init : -> System .

  var Atts : AttributeSet .
  var X@KBC : KBC .
  var X@Database : DatabaseClass .
  var O : Oid .
  var DB : Database .
  var MN : ModName .
  vars QIL QIL' QIL'' : QidList .
  var TL : TermList .

  rl [init] :
     init
     => [nil,
         < kbc : KBC |
              db : evalUnit(CONFIGURATION,
                     evalUnit(TRIV,
                       evalUnit(UP, emptyDatabase))),
              input : nilTermList, output : nil,
              default : nullModName >,
         nil] .

  crl [in] :
     [QIL,
```

```
     < O : X@KBC | db : DB, input : nilTermList,
          output : nil, default : MN, Atts >,
     QIL']
    => if meta-parse(KBC-GRAMMAR, QIL) == error*
       then [nil,
             < O : X@KBC | db : DB, input : nilTermList,
                  output : ('ERROR: 'incorrect 'input '.),
                  default : MN, Atts >,
             QIL']
       else [nil,
             < O : X@KBC | db : DB,
                  input : meta-parse(KBC-GRAMMAR, QIL),
                  output : nil, default : MN, Atts >,
             QIL']
       fi
       if QIL =/= nil .

  crl [out] :
    [QIL,
     < O : X@Database | db : DB, input : TL,
          output : QIL', default : MN, Atts >,
     QIL'']
    => [QIL,
        < O : X@Database | db : DB, input : TL,
             output : nil, default : MN, Atts >,
        (QIL' QIL'')]
       if QIL' =/= nil .
endm
```

5 Reduction Orders

A reduction ordering on a set of terms $T(\Sigma, V)$ is any well-founded rewrite ordering of $T(\Sigma, V)$ (see, e.g., [9, 1]). Reduction orders are an important tool for proving termination of rewrite systems and for orienting critical pairs during the completion process. A key problem for a given rewrite system is to find an appropriate reduction order. Thus, it is desirable to have a wide range of different possible reduction orders available.

As we have seen in previous sections, the termination checker and the completion procedure are parameterized by a reduction order. By using the reflective capabilities of the Maude system, the user is able to choose the order to be used among a set of available orders, but, more importantly, the tools also allows us to specify our own reduction orders at will, and use them in a very simple way.

In the present section we give the specification of two reduction orders, namely, the lexicographic path order and the Knuth-Bendix order. We will illustrate the use of the tools with these orderings in Section 6.

5.1 The Lexicographic Path Order

Path orderings are constructed on well-founded orderings of vocabularies. The idea is that a term s should be bigger than any term that is built from terms smaller than s which are connected together by a function symbol which is smaller than the root of s in the given order of functions. To compare a term s with another term t we proceed recursively. If the root of s is smaller than the root of t, then the only way s can be bigger than t is if one of the subterms of s is bigger than t. If the root of t is smaller, then s must be bigger than each of the subterms of t. When the roots are of the same precedence, their immediate subterms are compared recursively. Thus, in order to compare two terms we first compare their root symbols, and then recursively we compare the collections of their immediate subterms. Depending on how they are considered these collections of subterms we can have different types of orderings: the multiset path order, if the collections are seen as unordered multisets; lexicographic path order, if seen as ordered tuples; recursive path order with status, if a combination of the previous two ones is considered; etc. In the following we consider the lexicographic path order in more detail.

Definition 1 *(From [1]) Let Σ be a finite signature and let $>$ be a strict order on Σ. The* lexicographic path order $>_{lpo}$ *on $T(\Sigma, V)$ induced by $>$ is defined as follows: $s >_{lpo} t$ iff*

- *$t \in Var(s)$ and $s \neq t$, or*
- *$s = f(s_1, \ldots, s_m)$, $t = g(t_1, \ldots, t_n)$, and*
  - *there exists i, $1 \leq i \leq m$, with $s_i \geq_{lpo} t$, or*
  - *$f > g$ and $s >_{lpo} t_j$ for all j, $1 \leq j \leq n$, or*
  - *$f = g$, $s >_{lpo} t_j$ for all j, $1 \leq j \leq n$, and there exists i, $1 \leq i \leq m$, such that $s_1 = t_1, \ldots, s_{i-1} = t_{i-1}$ and $s_i >_{lpo} t_i$.*

Note that, in the last case, when $f = g$, the collections of immediate subterms are compared with respect to $>_{lpo}^n$, the n-fold lexicographic product of $>_{lpo}$ with itself, which explains the name *lexicographic* path order.

The definition of an order relation on terms belongs to the metalevel of the module of the terms to be ordered. Thus, given the metarepresentation of modules and terms defined by the Maude predefined module `META-LEVEL`, any strict order on Σ can be given as a strict order on quoted identifiers, that is, as an order satisfying the axioms in the following functional theory `STRICT-ORDER`.

```
(fth STRICT-ORDER is
   pr QID .
   op _>>_ : Qid Qid -> Bool .
   vars A B C : Qid .
   ceq A >> B = false if B >> A .
   ceq A >> C = true if A >> B and B >> C .
 endfth)
```

Then, we specify the lexicographic path order as a module parameterized by the choice of a strict order on Σ, that is, by an interpretation (view) of the parameter theory `STRICT-ORDER`. In this module we specify an operator `_>lpo_` whose definition follows quite closely the definition above. We also define several auxiliary functions: `any_>=lpo_`, that checks whether any of the terms in the list of terms given as first argument is greater or equal than the term given as second argument w.r.t. `_>=lpo_`; `all_>lpo_`, that checks whether the term given as first argument is greater than all the terms in the list given as second argument w.r.t. `_>=lpo_`; and `_occurs‘in_`, that checks whether the variable given as first argument occurs in the term given as second argument.

```
(fmod LPO[X :: STRICT-ORDER] is
   pr META-LEVEL .
   pr EXT-BOOL .

   op _>lpo_ : Term Term -> Bool .
   op _>lpo_ : TermList TermList -> Bool .
   op _>=lpo_ : Term Term -> Bool .
   op any_>=lpo_ : TermList Term -> Bool .
   op all_>lpo_ : Term TermList -> Bool .
   op _occurs‘in_ : Qid Term -> Bool .
   op _occurs‘in_ : Qid TermList -> Bool .

   vars T T’ : Term .
   vars TL TL’ : TermList .
   vars F F’ S S’ C C’ X Y : Qid .
```

```
  eq X occurs in Y = X == Y .
  eq X occurs in {C}S = false .
  eq X occurs in F[TL] = X occurs in TL .
  eq X occurs in (T, TL) = (X occurs in T) or (X occurs in TL) .

  eq T >=lpo T' = (T == T') orElse (T >lpo T') .

  eq (T, TL) >lpo (T', TL') = (T >lpo T') orElse (TL >lpo TL') .

  eq T >lpo X = (X =/= T) andThen (X occurs in T) .
  eq X >lpo {C}S = false .
  eq X >lpo F[TL] = false .
  eq {C}S >lpo {C'}S' = C >> C' == true .
  eq {C}S >lpo F[TL] = (C >> F == true) andThen (all {C}S >lpo TL) .
  eq F[TL] >lpo {C}S = (F >> C == true) orElse (any TL >=lpo {C}S) .
  eq F[TL] >lpo F'[TL']
    = (any TL >=lpo F'[TL'])
      orElse ((F >> F' == true) andThen (all F[TL] >lpo TL'))
      orElse ((F == F')
              andThen (all F[TL] >lpo TL')
              andThen (TL >lpo TL')) .

  eq any (T, TL) >=lpo T' = (T >=lpo T') orElse (any TL >=lpo T') .
  eq any T >=lpo T' = T >=lpo T' .

  eq all T >lpo (T', TL) = (T >lpo T') andThen (all T >lpo TL) .
  eq all T >lpo T' = T >lpo T' .
endfm)
```

5.2 The Knuth-Bendix Order

The Knuth-Bendix order was proposed in [16]. Here we present a slight variation of it, as presented in [1]. The Knuth-Bendix order is similar to the lexicographic path orders in that it compares the root symbols with respect to a given precedence order $>$, and then recursively the collections of immediate subterms.

Let Σ be a *finite* signature. A Knuth-Bendix order on $T(\Sigma, V)$ is determined by a strict order $>$ on Σ and a *weight function* $w : \Sigma \cup V \to Z^+$, where Z^+ denotes the set of nonnegative integers. Such a weight is said to be *admisible* for $>$ if and only if it satisfies the following properties:

- There exists $w_0 \in Z^+$ such that $w(x) = w_0$ for all variables $x \in V$ and $w(c) \geq w_0$ for all constants $c \in \Sigma$.

- If $f \in \Sigma$ is a unary function symbol of weight $w(f) = 0$, then f is the greatest element in Σ, i.e., $f \geq g$ for all $g \in \Sigma$.

Let as consider the functional theory `STRICT-ORDER` as given in Section 5.1. And let us consider a second parameter, specifying the weight function, given by the following functional theory `WEIGHT-FUNCTION`.

```
(fth WEIGHT-FUNCTION is
   pr QID .
   pr MACHINE-INT .
   op weight0 : -> MachineInt .
   op weight : Qid -> MachineInt .
 endfth)
```

This theory requires a function `weight` and a constant `weight0`. Note that, given the definition of weight that we are considering, we do not allow assigning different weights to different functions with the same name.

The weight function w can be extended to a function $w : T(\Sigma \cup V) \rightarrow Z^+$ as follows:

$$w(t) := \Sigma_{x \in vars(t)} w(x) \cdot |t|_x + \Sigma_{f \in \Sigma} w(f) \cdot |t|_f,$$

where $|t|_x$ (respectively, $|t|_f$) denotes the number of occurrences of the variable x (respectively, function symbol f) in t. Thus, $w(t)$ simply adds up the weights of all occurrences of symbols in t.

```
(fmod WEIGHT[X :: WEIGHT-FUNCTION] is
   pr META-LEVEL .

   op extweight : Term -> MachineInt .
   op extweight : TermList -> MachineInt .
   vars S F C V : Qid .
   var T : Term .
   var TL : TermList .
   eq extweight(F[TL]) = weight(F) + extweight(TL) .
   eq extweight({C}S) = weight(C) .
   eq extweight(V) = weight0 .
   eq extweight((T, TL)) = extweight(T) + extweight(TL) .
 endfm)
```

Definition 2 *Let Σ be a* finite *signature, let $>$ be a strict order on Σ, and let $w : \Sigma \cup V \rightarrow Z^+$ be a weight function. The* Knuth-Bendix order *$>_{kbo}$ on $T(\Sigma, V)$ induced by $>$ and w is defined as follows: for $s, t \in T(\Sigma, V)$, we have $s >_{kbo} t$ iff*

1. $|s|_x \geq |t|_x$ *for all* $x \in V$ *and* $w(s) > w(t)$, *or*

2. $|s|_x \geq |t|_x$ *for all* $x \in V$ *and* $w(s) = w(t)$, *and one of the following properties holds:*

   (a) *There are a unary function* f, *a variable* x, *and a positive integer* n *such that* $s = f^n(x)$ *and* $t = x$.

   (b) *There exist function symbols* f, g *such that* $f > g$ *and* $s = f(s_1, \ldots, s_m)$, $t = f(t_1, \ldots, t_n)$.

   (c) *There exist a function symbol* f *and an index* i, $1 \leq i \leq m$, *such that* $s = f(s_1, \ldots, s_m)$, $t = f(t_1, \ldots, t_m)$, *and* $s_1 = t_1, \ldots s_{i-1} = t_{i-1}$ *and* $s_i >_{kbo} t_i$.

We specify in the following module the function `_>kbo_`, whose definition mirrors very closely the previous definition. This module is parameterized by theories `STRICT-ORDER` and `WEIGHT-FUNCTION`. In it there are some auxiliary functions: `more-occurrences`, that checks whether the number of occurrences of each of the variables is greater in the term given as first argument than in the term given as second argument, and `kbo2a`, `kbo2b`, and `kbo2c`, which specify, respectively, conditions *2.(a)*, *2.(b)*, and *2.(c)*.

```
(fmod KBO[X :: STRICT-ORDER, Y :: WEIGHT-FUNCTION] is
   pr EXT-BOOL .
   pr WEIGHT[Y] .

   op _>kbo_ : Term Term -> Bool .

   vars T T' : Term .
   vars F F' C C' S S' V V' : Qid .
   vars TL TL' : TermList .
   var VS : QidSet .

   eq T >kbo T'
     = more-occurrences(T, T', (vars(T) ; vars(T')))
       andThen
       ( (extweight(T) > extweight(T'))
         orElse
         ( (extweight(T) == extweight(T'))
           andThen
           ( kbo2a(T, T')
             orElse
             ( kbo2b(T, T') orElse kbo2c(T, T') ) ) ) ) .
```

```
op kbo2a : Term Term -> Bool .
op kbo2aAux : TermList Term -> Bool .

eq kbo2a(F[T], T') = kbo2aAux(T, T') .
eq kbo2a(F[T, TL], T') = kbo2aAux(T, T') orElse kbo2aAux(TL, T') .
eq kbo2a(V, T) = false .
eq kbo2a({C}S, T) = false .

eq kbo2aAux((T, TL), T') = kbo2aAux(T, T') orElse kbo2aAux(T, TL) .
eq kbo2aAux(F[T], T') = (F[T] == T') orElse kbo2aAux(T, T') .
eq kbo2aAux(F[T, TL], T')
  = (F[T, TL] == T') orElse kbo2aAux((T, TL), T') .
eq kbo2aAux(V, T) = V == T .
eq kbo2aAux({C}S, T) = {C}S == T .

op kbo2b : Term Term -> Bool .

eq kbo2b(F[TL], F'[TL']) = (F >> F') == true .
eq kbo2b(F[TL], {C}S) = (F >> C) == true .
eq kbo2b({C}S, F[TL]) = (C >> F) == true .
eq kbo2b({C}S, {C'}S') = (C >> C') == true .
eq kbo2b(T, V) = false .
eq kbo2b(V, T) = false .

op kbo2c : Term Term -> Bool .

eq kbo2c(F[TL], F'[TL']) = (F == F') andThen (TL >kbo TL') .
eq kbo2c(F[TL], {C}S) = (F == C) .
eq kbo2c(T, V) = false .
eq kbo2c(V, T) = false .
eq kbo2c({C}S, F[TL]) = false .
eq kbo2c({C}S, {C'}S') = (C == C') .

op _>kbo_ : TermList TermList -> Bool .

eq (T, TL) >kbo (T', TL')
  = (T >kbo T') orElse ((not T' >kbo T) andThen (TL >kbo TL')) .
eq T >kbo (T', TL') = (T >kbo T') .
eq (T, TL) >kbo T' = (T >kbo T') .

op vars : Term -> QidSet .
op vars : TermList -> QidSet .
eq vars(F[TL]) = vars(TL) .
eq vars((T, TL)) = vars(T) ; vars(TL) .
eq vars(V) = V .
```

```
  eq vars({C}S) = none .
  eq V ; V = V .

  op more-occurrences : Term Term QidSet -> Bool .
  op occurrences : Term Qid -> MachineInt .
  op occurrences : TermList Qid -> MachineInt .

  eq more-occurrences(T, T', (V ; VS))
    = (occurrences(T, V) >= occurrences(T', V))
      andThen
      more-occurrences(T, T', VS) .
  eq more-occurrences(T, T', none) = true .

  eq occurrences(F[TL], V) = occurrences(TL, V) .
  eq occurrences((T, TL), V) = occurrences(T, V) + occurrences(TL, V) .
  eq occurrences({C}S, V) = 0 .
  eq occurrences(V, V') = if V == V' then 1 else 0 fi .
endfm)
```

Thus, the Knuth-Bendix order makes a lexicographic comparison, where first the weights are considered, then their root symbols, and finally recursively the collections of the immediate subterms.

6 Some Examples

In this section we illustrate the use of the completion tool with several examples. All of these examples have already appeared in the literature; they have been taken from [16, 9, 12].

6.1 Group Theory I

The first example we present is the traditional definition of an abstract group, as presented in [1]. Here we have three operators: a constant `1`, a unary operator `i`, and a infix binary operator `_*_`, which satisfy the axioms in the following module.

```
(fmod GROUP-I is
  sort S .
  op _*_ : S S -> S .
  op 1 : -> S .
  op i : S -> S .
  var X : S .
  var Y : S .
```

```
  var Z : S .
  eq (X * Y) * Z = X * (Y * Z) .
  eq 1 * X = X .
  eq i(X) * X = 1 .
endfm)
```

Let us use the lexicographic path order as defined in Section 5.1, which we instantiate with the following module defining precedences for the operators.

```
(fmod GROUP-I-ORDER is
  pr QID .
  op _>>_ : Qid Qid -> Bool .
  eq '_*_ >> '1 = true .
  eq 'i >> '_*_ = true .
  eq 'i >> '1 = true .
endfm)
```

Note that the order relation `_>>_` defined in `GROUP-I-ORDER` is not completely defined. Although it is declared as taking any two quoted identifiers, it is in fact only declared for those identifiers in the signature of the module on which the order will be used.

In order to instantiate the `LPO` module with the previous specification we need to define the appropriate view from the `STRICT-ORDER` theory to the `GROUP-I-ORDER` module. Note that since in this case we have used the same names for the sorts and the operators, the view is defined trivially as follows.

```
(view GROUP-I-ORDER from STRICT-ORDER to GROUP-I-ORDER is
  op _>>_ to _>>_ .
endv)
```

We are now ready to generate the order specification by instantiating the `LPO` module with the `GROUP-I-ORDER` view. This can be done by giving the appropriate module expression, either in another module or in a command. Once all these modules are entered in the tool, we are ready to check termination and to complete the specification.

```
Maude> (check termination GROUP-I
            with LPO[GROUP-I-ORDER] * (op _>lpo_ to _>>>_) .)
rewrites: 15632 in 150ms cpu (160ms real) (104213 rewrites/second)

Termination check of GROUP-I
The specification is terminating.
```

```
Maude> (complete GROUP-I
            with LPO[GROUP-I-ORDER] * (op _>lpo_ to _>>>_) .)
rewrites: 132249 in 1460ms cpu (1470ms real) (90581 rewrites/second)

Knuth-Bendix completion of GROUP-I
Completion solution:
fmod GROUP-I is
        including BOOL .
        sorts S .
        op _*_ : S S -> S .
        op 1 : -> S .
        op i : S -> S .
        var X : S .
        var Y : S .
        var Z : S .
        var Y@ : S .
        var Z@ : S .
        var Y@@ : S .
        var Z@@ : S .
        var Y@@@ : S .
        eq Y@ * ( i ( Y@ ) * Z@@ ) = Z@@ .
        eq Y@@ * i ( Y@@ ) = 1 .
        eq Z@@ * 1 = Z@@ .
        eq ( X * Y ) * Z = X * ( Y * Z ) .
        eq i ( X ) * X = 1 .
        eq i ( Y@ ) * ( Y@ * Z@ ) = Z@ .
        eq 1 * X = X .
        eq i ( Y@@ * Y@@@ ) = i ( Y@@@ ) * i ( Y@@ ) .
        eq i ( i ( Y@@ ) ) = Y@@ .
        eq i ( 1 ) = 1 .
endfm
The equations in the resulting module are sort decreasing
```

This specification was first used by Knuth and Bendix in [16] to show the use of their completion procedure. We can use the Knuth-Bendix order, as they did, in order to complete such a specification by using the reduction order specified by the module resulting from instantiating the `KBO` module of Section 5.2 with the `GROUP-I-ORDER` module above and the weight function given by the following specification `GROUP-I-WEIGHT`.

```
(fmod GROUP-I-WEIGHT is
   pr QID .
   pr MACHINE-INT .
```

```
   op weight : Qid -> MachineInt .
   op weight0 : -> MachineInt .
   eq weight('_*_) = 0 .
   eq weight('i) = 0 .
   eq weight('1) = 1 .
   eq weight0 = 1 .
 endfm)
```

As done previously for the `_>>_` function, in the `GROUP-I-WEIGHT` module, we only give the definition of the `weight` function on the quoted identifiers corresponding to the operators used in the module in question.

In order to be able to use these modules to instantiate the `KBO` module we need to define the appropriate views. We can use the `GROUP-I-ORDER` view above for the `GROUP-I-ORDER` module and the following `GROUP-I-WEIGHT` view from `WEIGHT-FUNCTION` to `GROUP-I-WEIGHT`.

```
(view GROUP-I-WEIGHT from WEIGHT-FUNCTION to GROUP-I-WEIGHT is
   op weight to weight .
   op weight0 to weight0 .
 endv)
```

This reduction order also shows the termination of the specification.

```
Maude> (check termination GROUP-I
            with KBO[GROUP-I-ORDER, GROUP-I-WEIGHT]
                   * (op _>kbo_ to _>>>_) .)
rewrites: 45098 in 170ms cpu (180ms real) (265282 rewrites/second)

Termination check of GROUP-I
The specification is terminating.
```

Now, we are ready to execute the `complete_with_` command with the induced order, obtaining the same result.

```
Maude> (complete GROUP-I
            with KBO[GROUP-I-ORDER, GROUP-I-WEIGHT]
                   * (op _>kbo_ to _>>>_) .)
rewrites: 160910 in 1600ms cpu (1610ms real) (100568 rewrites/second)

Knuth-Bendix completion of GROUP-I
Completion solution:
fmod GROUP-I is
        including BOOL .
        sorts S .
```

```
        op _*_ : S S -> S .
        op 1 : -> S .
        op i : S -> S .
        var X : S .
        var Y : S .
        var Z : S .
        var Y@ : S .
        var Z@ : S .
        var Y@@ : S .
        var Z@@ : S .
        var Y@@@ : S .
        eq Y@ * ( i ( Y@ ) * Z@@ ) = Z@@ .
        eq Y@@ * i ( Y@@ ) = 1 .
        eq Z@@ * 1 = Z@@ .
        eq ( X * Y ) * Z = X * ( Y * Z ) .
        eq i ( X ) * X = 1 .
        eq i ( Y@ ) * ( Y@ * Z@ ) = Z@ .
        eq 1 * X = X .
        eq i ( Y@@ * Y@@@ ) = i ( Y@@@ ) * i ( Y@@ ) .
        eq i ( i ( Y@@ ) ) = Y@@ .
        eq i ( 1 ) = 1 .
endfm
The equations in the resulting module are sort decreasing
```

It may be hard to find the appropriate order for a given specification, and, even having the right order, we must be cautious about the specification of the parameters. For example, in `GROUP-I-WEIGHT`, we assigned weight 0 to the operators `_*_` and `i`, and 1 to the operator `1`. If instead we assign weight 0 to `i`, the completion process does not terminate.

```
(fmod GROUP-I-WEIGHT-2 is
   pr QID .
   pr MACHINE-INT .
   op weight : Qid -> MachineInt .
   op weight0 : -> MachineInt .
   eq weight('_*_) = 0 .
   eq weight('i) = 1 .
   eq weight('1) = 1 .
   eq weight0 = 1 .
 endfm)

(view GROUP-I-WEIGHT-2 from WEIGHT-FUNCTION to GROUP-I-WEIGHT-2 is
   op weight to weight .
   op weight0 to weight0 .
 endv)
```

```
(view GROUP-I-ORDER from STRICT-ORDER to GROUP-I-ORDER is
   op _>>_ to _>>_ .
 endv)

Maude> (check termination GROUP-I
            with KBO[GROUP-I-ORDER, GROUP-I-WEIGHT-2]
                   * (op _>kbo_ to _>>>_) .)
rewrites: 45450 in 200ms cpu (210ms real) (227250 rewrites/second)

Termination check of GROUP-I
The specification is terminating.

Maude> (complete GROUP-I
            with KBO[GROUP-I-ORDER, GROUP-I-WEIGHT-2]
                   * (op _>kbo_ to _>>>_) .)
```

6.2 Group Theory II

Let us consider now the same specification of group theory of Section 6.1, but now with right identity and right inverse instead of left identity and with left inverse. This example has also been taken from [16].

```
(fmod GROUP-II is
    sort S .
    op 1 : -> S .
    op i : S -> S .
    op _*_ : S S -> S .
    vars X Y Z : S .
    eq X * 1 = X .
    eq X * i(X) = 1 .
    eq (X * Y) * Z = X * (Y * Z) .
 endfm)
```

We can use the modules defining the order and weight functions of Section 6.1. The result is as follows.

```
Maude> (check termination  GROUP-II
            with KBO[GROUP-I-ORDER, GROUP-I-WEIGHT]
                   * (op _>kbo_ to _>>>_) .)
rewrites: 44981 in 240ms cpu (280ms real) (187420 rewrites/second)

Termination check of GROUP-II
The specification is terminating.

Maude> (complete GROUP-II
```

```
            with KBO[GROUP-I-ORDER, GROUP-I-WEIGHT]
                   * (op _>kbo_ to _>>>_) .)
rewrites: 205320 in 3180ms cpu (3190ms real) (64566 rewrites/second)

Knuth-Bendix completion of GROUP-II
Completion solution:
fmod GROUP-II is
        including BOOL .
        sorts S .
        op _*_ : S S -> S .
        op 1 : -> S .
        op i : S -> S .
        var X : S .
        var Y : S .
        var Z : S .
        var X@ : S .
        var Z@ : S .
        var Z@@ : S .
        var X@@ : S .
        var X@@@ : S .
        var X@@@@ : S .
        eq X * i ( X ) = 1 .
        eq X * 1 = X .
        eq X@ * ( i ( X@ ) * Z@ ) = Z@ .
        eq ( X * Y ) * Z = X * ( Y * Z ) .
        eq i ( X@@ ) * X@@ = 1 .
        eq i ( X@@@ ) * ( X@@@ * Z@@ ) = Z@@ .
        eq 1 * X@@ = X@@ .
        eq i ( X@@@ * X@@@@ ) = i ( X@@@@ ) * i ( X@@@ ) .
        eq i ( i ( X@@@ ) ) = X@@@ .
        eq i ( 1 ) = 1 .
endfm
The equations in the resulting module are sort decreasing
```

6.3 The Inverse Property

We can use again the same order and weight functions of Section 6.1 in order to complete the following specification, also taken from [16], in which we have only two operators, `_*_` and `i`. In this case we assume only one axiom, specifying the inverse property.

```
(fmod INVERSE-PROPERTY is
    sort S .
    op 1 : -> S .
```

```
    op i : S -> S .
    op _*_ : S S -> S .
    vars X Y : S .
    eq i(X) * (X * Y) = Y .
 endfm)
```

The result of the tools is the following.

```
Maude> (check termination INVERSE-PROPERTY
            with KBO[GROUP-I-ORDER, GROUP-I-WEIGHT]
                    * (op _>kbo_ to _>>>_) .)
rewrites: 44628 in 160ms cpu (170ms real) (278925 rewrites/second)

Termination check of INVERSE-PROPERTY
The specification is terminating.

Maude> (complete INVERSE-PROPERTY
            with KBO[GROUP-I-ORDER, GROUP-I-WEIGHT]
                    * (op _>kbo_ to _>>>_) .)
rewrites: 54285 in 530ms cpu (540ms real) (102424 rewrites/second)

Knuth-Bendix completion of INVERSE-PROPERTY
Completion solution:
fmod INVERSE-PROPERTY is
        including BOOL .
        sorts S .
        op _*_ : S S -> S .
        op 1 : -> S .
        op i : S -> S .
        var X : S .
        var Y : S .
        var Y@ : S .
        eq X * ( i ( X ) * Y@ ) = Y@ .
        eq i ( X ) * ( X * Y ) = Y .
        eq i ( i ( X ) ) * Y = X * Y .
endfm
The equations in the resulting module are sort decreasing
```

6.4 Uniqueness of the Identity and Inverse Functions

Another interesting example, also from [16], is the following specification, with which we can prove the uniqueness of the left identity and left inverse functions. Let us consider the following specification.

```
(fmod GROUP-III is
```

```
    sort S .
    op 1 : -> S .
    op e : -> S .
    op i : S -> S .
    op f : S -> S .
    op _*_ : S S -> S .
    vars X Y Z : S .
    eq 1 * X = X .
    eq e * X = X .
    eq i(X) * X = 1 .
    eq f(X) * X = e .
    eq (X * Y) * Z = X * (Y * Z) .
 endfm)
```

In this case we cannot use the order and weight functions defined in Section 6.1 for the `GROUP-I` module, since we have additional operators `e` and `f`. We give below such modules and the appropriate views for the instantiation.

```
(fmod GROUP-III-ORDER is
   pr QID .
   op _>>_ : Qid Qid -> Bool .

   eq '_*_ >> '1 = true .
   eq '_*_ >> 'e = true .
   eq 'i >> '_*_ = true .
   eq 'f >> '_*_ = true .
   eq 'i >> '1 = true .
   eq 'f >> '1 = true .
   eq 'f >> 'i = true .
   eq 'f >> 'e = true .
   eq 'e >> '1 = true .
   eq 'i >> 'e = true .
 endfm)

(fmod GROUP-III-WEIGHT is
   pr QID .
   pr MACHINE-INT .
   op weight : Qid -> MachineInt .
   op weight0 : -> MachineInt .
   eq weight('_*_) = 0 .
   eq weight('i) = 0 .
   eq weight('f) = 0 .
   eq weight('1) = 1 .
   eq weight('e) = 1 .
```

```
   eq weight0 = 1 .
 endfm)

(view GROUP-III-WEIGHT from WEIGHT-FUNCTION to GROUP-III-WEIGHT is
   op weight to weight .
   op weight0 to weight0 .
 endv)

(view GROUP-III-ORDER from STRICT-ORDER to GROUP-III-ORDER is
   op _>>_ to _>>_ .
 endv)
```

The results given by the tools are as follows.

```
Maude> (check termination GROUP-III
            with KBO[GROUP-III-ORDER, GROUP-III-WEIGHT]
                   * (op _>kbo_ to _>>>_) .)
rewrites: 45240 in 170ms cpu (180ms real) (266117 rewrites/second)

Termination check of GROUP-III
The specification is terminating.

Maude> (complete GROUP-III
            with KBO[GROUP-III-ORDER, GROUP-III-WEIGHT]
                   * (op _>kbo_ to _>>>_) .)
rewrites: 201324 in 2240ms cpu (2250ms real) (89876 rewrites/second)

Knuth-Bendix completion of GROUP-III
Completion solution:
fmod GROUP-III is
        including BOOL .
        sorts S .
        op e : -> S .
        op _*_ : S S -> S .
        op 1 : -> S .
        op i : S -> S .
        op f : S -> S .
        var X : S .
        var Y : S .
        var Z : S .
        var Y@ : S .
        var Z@ : S .
        var Y@@ : S .
        var Z@@ : S .
        var Z@@@ : S .
```

```
        var Y@@@ : S .
        eq Z@@ * ( i ( Z@@ ) * Z@@@ ) = Z@@@ .
        eq Z@@ * i ( Z@@ ) = 1 .
        eq Z@@ * 1 = Z@@ .
        eq ( X * Y ) * Z = X * ( Y * Z ) .
        eq i ( X ) * X = 1 .
        eq i ( Y@ ) * ( Y@ * Z@ ) = Z@ .
        eq 1 * X = X .
        eq i ( Y@@ * Y@@@ ) = i ( Y@@@ ) * i ( Y@@ ) .
        eq i ( i ( Z@@ ) ) = Z@@ .
        eq i ( 1 ) = 1 .
        eq f ( Z@@ ) = i ( Z@@ ) .
        eq e = 1 .
endfm
The equations in the resulting module are sort decreasing
```

In addition to other ten reductions, the tool returns equations `f(Z@@) = i(Z@@)`, with `Z@@` a variable of sort `S`, and `e = 1`, which shows that the identity and inverse functions are unique.

6.5 Central Groupoids

In the reduction order proposed originally by Knuth and Bendix in [16] they do not consider an order relation on the operators of a specification, they just consider the weight function. This order can be enough in many cases, as [16] illustrates with some examples and as we show here with the following specification, taken from [16].

```
(fmod CENTRAL-GROUPOIDS is
    sort S .
    op _*_ : S S -> S .
    vars X Y Z : S .
    eq (X * Y) * (Y * Z) = Y .
 endfm)
```

In this specification of central groupoids we have a unique operator `_*_`, on which there is one single axiom. In order to use the `KBO` module of Section 5.2 we need to give modules defining the order on the operators and the weight function. Note however that the following module `CENTRAL-GROUPOIDS-ORDER` does not include any equation defining the `_>>_` operator.

```
(fmod CENTRAL-GROUPOIDS-ORDER is
```

```
   pr QID .
   op _>>_ : Qid Qid -> Bool .
 endfm)

(fmod CENTRAL-GROUPOIDS-WEIGHT is
   pr QID .
   pr MACHINE-INT .
   op weight : Qid -> MachineInt .
   op weight0 : -> MachineInt .
   eq weight('_*_) = 0 .
   eq weight0 = 1 .
 endfm)

(view CENTRAL-GROUPOIDS-ORDER
      from STRICT-ORDER to CENTRAL-GROUPOIDS-ORDER is
   op _>>_ to _>>_ .
 endv)

(view CENTRAL-GROUPOIDS-WEIGHT
      from WEIGHT-FUNCTION to CENTRAL-GROUPOIDS-WEIGHT is
   op weight to weight .
   op weight0 to weight0 .
 endv)
```

The results given by the tools are as follows.

```
Maude> (check termination CENTRAL-GROUPOIDS
        with KBO[CENTRAL-GROUPOIDS-ORDER, CENTRAL-GROUPOIDS-WEIGHT]
               * (op _>kbo_ to _>>>_) .)
rewrites: 44738 in 170ms cpu (180ms real) (263164 rewrites/second)

Termination check of CENTRAL-GROUPOIDS
The specification is terminating.

Maude> (complete CENTRAL-GROUPOIDS
        with KBO[CENTRAL-GROUPOIDS-ORDER, CENTRAL-GROUPOIDS-WEIGHT]
               * (op _>kbo_ to _>>>_) .)
rewrites: 62259 in 580ms cpu (590ms real) (107343 rewrites/second)

Knuth-Bendix completion of CENTRAL-GROUPOIDS
Completion solution:
fmod CENTRAL-GROUPOIDS is
        including BOOL .
        sorts S .
        op _*_ : S S -> S .
```

```
        var X : S .
        var Y : S .
        var Z : S .
        var X@ : S .
        var Z@ : S .
        eq Y * ( ( Y * Z ) * Z@ ) = Y * Z .
        eq ( X * Y ) * ( Y * Z ) = Y .
        eq ( X@ * ( X * Y ) ) * Y = X * Y .
endfm
The equations in the resulting module are sort decreasing
```

6.6 Central Groupoids II

Another interesting example from [16] is the following one. In this example we consider the model of central groupoids with n^2 elements given by the ordered set S of pairs $\{(x_1, x_2) \mid x_1, x_2 \in S_0\}$, where S_0 is a set of n elements. If we define the product $(x_1, x_2) * (y_1, y_2) = (x_2, y_1)$, we find that the basic identity $(a * b) * (b * c) = b$ is satisfied. We can then give the following specification.

```
(fmod CENTRAL-GROUPOIDS-II is
    sort S .
    op _*_ : S S -> S .
    vars X Y Z : S .
    ops p1_ p2_ : S -> S .
    eq (X * X) * X = p1 X .
    eq X * (X * X) = p2 X .
    eq (X * Y) * (Y * Z) = Y .
    eq (p2 X) * Y = X * Y .
 endfm)
```

Let us use the module `CENTRAL-GROUPOIDS-ORDER` in the previous section and the following `CENTRAL-GROUPOIDS-II-WEIGHT` module.

```
(fmod CENTRAL-GROUPOIDS-II-WEIGHT is
   pr QID .
   pr MACHINE-INT .
   op weight : Qid -> MachineInt .
   op weight0 : -> MachineInt .
   eq weight('_*_) = 1 .
   eq weight('p1_) = 1 .
   eq weight('p2_) = 1 .
   eq weight0 = 1 .
 endfm)
```

```
(view CENTRAL-GROUPOIDS-II-WEIGHT
      from WEIGHT-FUNCTION to CENTRAL-GROUPOIDS-II-WEIGHT is
   op weight to weight .
   op weight0 to weight0 .
 endv)
```

The results given by the tools are then as follows.

```
Maude> (check termination CENTRAL-GROUPOIDS-II
        with KBO[CENTRAL-GROUPOIDS-ORDER, CENTRAL-GROUPOIDS-II-WEIGHT]
               * (op _>kbo_ to _>>>_) .)
rewrites: 45230 in 160ms cpu (170ms real) (282687 rewrites/second)

Termination check of CENTRAL-GROUPOIDS-II
The specification is terminating.

Maude> (complete CENTRAL-GROUPOIDS-II
        with KBO[CENTRAL-GROUPOIDS-ORDER, CENTRAL-GROUPOIDS-II-WEIGHT]
               * (op _>kbo_ to _>>>_) .)
rewrites: 657290 in 8110ms cpu (8380ms real) (81046 rewrites/second)

Knuth-Bendix completion of CENTRAL-GROUPOIDS-II
Completion solution:
fmod CENTRAL-GROUPOIDS-II is
        including BOOL .
        sorts S .
        op _*_ : S S -> S .
        op p1_ : S -> S .
        op p2_ : S -> S .
        var X : S .
        var Y : S .
        var Z : S .
        var X@ : S .
        var Y@ : S .
        var Z@ : S .
        var Y@@ : S .
        var Z@@ : S .
        var X@@ : S .
        var X@@@ : S .
        eq Y * p1 Z = Y * Z .
        eq Y@ * p2 Y@ = p2 Y@ .
        eq X@@ * ( X@ * Z@@ ) = X@@ * p2 X@ .
        eq ( X@ * Z@ ) * Y@ = p1 Z@ * Y@ .
        eq p1 X * X = p1 X .
        eq p1 Y@@ * p2 Y@@ = Y@@ .
```

```
        eq p2 X * Y = X * Y .
        eq p1 ( Y@ * Z@ ) = p2 Y@ .
        eq p1 p1 Z = p1 Z .
        eq p1 p2 X = p2 X .
        eq p2 ( X * X@@@ ) = p1 X@@@ .
        eq p2 p1 Z = p1 Z .
        eq p2 p2 X@ = p2 X@ .
endfm
The equations in the resulting module are sort decreasing
```

If instead of the module `CENTRAL-GROUPOIDS-II` we consider the following module with only three axioms we get a complete module with twenty five equations.

```
(fmod CENTRAL-GROUPOIDS-III is
    sort S .
    op _*_ : S S -> S .
    vars X Y Z : S .
    ops p1_ p2_ : S -> S .
    eq (X * X) * X = p1 X .
    eq X * (X * X) = p2 X .
    eq (X * Y) * (Y * Z) = Y .
 endfm)

Maude> (check termination CENTRAL-GROUPOIDS-III
        with KBO[CENTRAL-GROUPOIDS-ORDER, CENTRAL-GROUPOIDS-II-WEIGHT]
               * (op _>kbo_ to _>>>_) .)
rewrites: 45375 in 200ms cpu (210ms real) (226875 rewrites/second)

Termination check of CENTRAL-GROUPOIDS-III
The specification is terminating.

Maude> (complete CENTRAL-GROUPOIDS-III
        with KBO[CENTRAL-GROUPOIDS-ORDER, CENTRAL-GROUPOIDS-II-WEIGHT]
               * (op _>kbo_ to _>>>_) .)
rewrites: 694179 in 4770ms cpu (4790ms real) (145530 rewrites/second)

Knuth-Bendix completion of CENTRAL-GROUPOIDS-III
Completion solution:
fmod CENTRAL-GROUPOIDS-III is
        including BOOL .
        sorts S .
        op _*_ : S S -> S .
        op p1_ : S -> S .
        op p2_ : S -> S .
```

```
        var X : S .
        var Y : S .
        var Z : S .
        var X@ : S .
        var Y@ : S .
        var Z@ : S .
        var Y@@ : S .
        var Z@@ : S .
        var X@@ : S .
        eq X * ( X * X ) = p2 X .
        eq Y * ( ( Y * Z ) * Z@ ) = Y * Z .
        eq Y * p2 ( Y * Z ) = Y * Z .
        eq Y@ * ( p2 Y@ * Z@ ) = p2 Y@ .
        eq Y@ * p2 p2 Y@ = p2 Y@ .
        eq Z@ * p1 Z@ = Z@ * Z@ .
        eq ( X * X ) * X = p1 X .
        eq ( X * Y ) * ( Y * Z ) = Y .
        eq ( Z * Z ) * ( p1 Z * Z@@ ) = p1 Z .
        eq ( Z * Z ) * p2 p1 Z = p1 Z .
        eq ( X@ * Y@ ) * p2 Y@ = Y@ .
        eq ( X@ * ( X * Y ) ) * Y = X * Y .
        eq ( X@ * ( Z@ * Z@ ) ) * p1 Z@ = Z@ * Z@ .
        eq ( X@ * p1 Y@@ ) * Y@@ = p1 Y@@ .
        eq ( X@@ * p2 Z ) * ( Z * Z ) = p2 Z .
        eq p1 Y@ * ( Y@ * Z@ ) = Y@ .
        eq p1 Y@@ * p2 Y@@ = Y@@ .
        eq p1 ( X * Y ) * Y = X * Y .
        eq p1 p1 Y@ * Y@ = p1 Y@ .
        eq p1 p2 Z * ( Z * Z ) = p2 Z .
        eq p2 X@ * ( ( X@ * X@ ) * Z@ ) = X@ * X@ .
        eq p2 Y@ * Y@ = Y@ * Y@ .
        eq p2 X@@ * p1 X@@ = X@@ * X@@ .
        eq p1 ( Z * Z ) = p2 Z .
        eq p2 ( Z * Z ) = p1 Z .
endfm
The equations in the resulting module are sort decreasing
```

6.7 Defining New Orders

We have seen in the previous sections how to use the tool with the reduction orders defined in Sections 5.1 and 5.2. In this section we illustrate the possibility of defining new reduction orders and using them to use the tool in order to complete specifications.

Dershowitz and Jouannaud use in [9] the order induced by the size of a term to complete the following specification of group theory.

```
(fmod GROUP-IV is
    sort S .
    op 1 : -> S .
    op i : S -> S .
    op __ : S S -> S .
    vars X Y : S .
    eq X  1 = X .
    eq 1  X = X .
    eq i(X)  (X  Y) = Y .
 endfm)

(fmod SIZE-ORDER is
   pr META-LEVEL .
   op _>size_ : Term Term -> Bool .
   op size : Term -> MachineInt .
   op size : TermList -> MachineInt .
   vars S F C V : Qid .
   vars T T' : Term .
   var TL : TermList .
   eq T >size T' = size(T) > size(T') .
   eq size(F[TL]) = 1 + size(TL) .
   eq size({C}S) = 1 .
   eq size(V) = 1 .
   eq size((T, TL)) = size(T) + size(TL) .
 endfm)
```

The tools can be used with this order in exactly the same way as the other cases we have presented in the previous sections.

```
Maude> (check termination GROUP-IV with SIZE-ORDER *(op _>size_ to _>>>_) .)
rewrites: 2540 in 0ms cpu (10ms real) (~ rewrites/second)

Termination check of GROUP-IV
The specification is terminating.

Maude> (complete GROUP-IV with SIZE-ORDER *(op _>size_ to _>>>_) .)
rewrites: 31561 in 370ms cpu (380ms real) (85300 rewrites/second)

Knuth-Bendix completion of GROUP-IV
Completion solution:
fmod GROUP-IV is
        including BOOL .
        sorts S .
```

```
        op 1 : -> S .
        op i : S -> S .
        op __ : S S -> S .
        var X : S .
        var Y : S .
        var X@ : S .
        var Y@ : S .
        var Y@@ : S .
        eq i ( i ( Y@ ) ) = Y@ .
        eq i ( 1 ) = 1 .
        eq X  1 = X .
        eq Y@  i ( Y@ ) = 1 .
        eq Y@  ( i ( Y@ )  Y@@ ) = Y@@ .
        eq i ( X )  ( X  Y ) = Y .
        eq i ( X@ )  X@ = 1 .
        eq 1  X = X .
endfm
The equations in the resulting module are sort decreasing
```

6.8 The Use of the Commutative Attribute

As we saw in [11], the unification and matching algorithms implemented support the commutative attribute. Thus, as for the Church-Rosser checker, the Knuth-Bendix completion tool at hand can complete specifications with operators declared commutative with the `comm` attribute. However, we must be careful when completing this kind of specifications using the tool, since the orders presented in Sections 5.1 and 5.2 will not be useful in many cases.

Let us consider for example, the following specification of the group theory, which is as the one given in Section 6.1, but with the `_*_` operator declared commutative.

```
(fmod GROUP-V is
    sort S .
    op 1 : -> S .
    op i : S -> S .
    op _*_ : S S -> S [comm] .
    vars X Y Z : S .
    eq 1 * X = X .
    eq i(X) * X = 1 .
    eq (X * Y) * Z = X * (Y * Z) .
 endfm)
```

Using for example the Knuth-Bendix order specification of Section 5.2 instantiated with the views `GROUP-I-ORDER` and `GROUP-I-WEIGHT` given in

Section 6.1, we get the following result.

```
Maude> (complete GROUP-V
            with KBO[GROUP-I-ORDER, GROUP-I-WEIGHT]
                   * (op _>kbo_ to _>>>_) .)
rewrites: 45637 in 330ms cpu (340ms real) (138293 rewrites/second)

Knuth-Bendix completion of GROUP-V
The specification cannot be completed.
The critical pair
        eq '_*_ [ '_*_ [ 'X , 'Y ] , 'Z ] = '_*_ [ 'X , '_*_ [ 'Y , 'Z ] ] .
cannot be oriented with the order given.
```

This result is due to the fact that there can be critical pairs which cannot be oriented. Although in the particular order in which the arguments of these terms are given they can be ordered with the given order, they cannot if their arguments are permuted. And this is in fact what has happened. The commutative attribute allows the system to change the order in which the arguments are considered, which has in fact been done in order to construct its internal normal-form representation.

Of course, it is not that the specification cannot be completed using the tool. It is just that a different order will have to be provided. For example, we can use a multiset path ordering, or more simply, the order induced by the size function as in Section 6.7. The general idea is to provide an order on terms which is compatible with the commutative attribute, in the sense that the order between two terms does not depend on the representatives modulo commutativity.

6.9 Completion of a Specification of the Integer Numbers

Let us consider the following order-sorted specification of the integer numbers, which is adapted from the one given by Ganzinger in [12], and in which, in addition to the constant `0` and the unary operators `s_` and `-_`, we have infix binary operators `_+_` and `_*_`, and a unary operator `square` for the square root of integer numbers.

```
(fmod INT is
    sorts NzNat Nat NzInt Int .
    subsorts NzNat < Nat < Int .
    subsorts NzNat < NzInt < Int .
    op 0 : -> Nat .
    op s_ : Nat -> NzNat .
    op _+_ : Nat Nat -> Nat .
```

```
    op _+_ : Nat NzNat -> Nat .
    op _+_ : NzNat Nat -> Nat .
    op _+_ : NzNat NzNat -> NzNat .
    op _+_ : Int Int -> Int .
    op -_ : Nat -> Int .
    op -_ : NzNat -> NzInt .
    op -_ : Int -> Int .
    op -_ : NzInt -> NzInt .
    op _*_ : Nat Nat -> Nat .
    op _*_ : Int Int -> Int .
    op square : Int -> Nat .
    vars I J L : Int .
    vars M N : Nat .
    var K : NzNat .
    eq - 0 = 0 .
    eq - - I = I .
    eq I + 0 = I .
    eq 0 + I = I .
    eq K + s M = s(K + M) .
    eq (- s K) + s M = (- K) + M .
    eq I + (- J) = - ((- I) + J) .
    eq I * 0 = 0 .
    eq 0 * I = 0 .
    eq I * s N = (I * N) + I .
    eq I * - J = - (I * J) .
    eq (- I) * J = - (I * J) .
    eq square(I) = I * I .
 endfm)
```

In order to complete the specification above, let us consider the lexicographic path order defined by the inequalities in the following module.

```
(fmod INT-ORDER is
   pr QID .
   op _>>_ : Qid Qid -> Bool .
   eq 'square >> '_*_ = true .
   eq '_*_ >> '_+_ = true .
   eq '_*_ >> '-_ = true .
   eq '_+_ >> 's_ = true .
   eq '_+_ >> '-_ = true .
 endfm)

(view INT-ORDER from STRICT-ORDER to INT-ORDER is
   op _>>_ to _>>_ .
 endv)
```

The results given by the tools are as follows.

```
Maude> (check termination INT with LPO[INT-ORDER] * (op _>lpo_ to _>>>_) .)
rewrites: 17531 in 80ms cpu (90ms real) (219137 rewrites/second)

Termination check of INT
The specification is terminating.

Maude> (complete INT with LPO[INT-ORDER] * (op _>lpo_ to _>>>_) .)
rewrites: 380499 in 3480ms cpu (3560ms real) (109338 rewrites/second)

Knuth-Bendix completion of INT
Completion solution:
fmod INT is
        including BOOL .
        sorts NzNat Nat NzInt Int .
        subsort NzNat < Nat .
        subsort NzNat < NzInt .
        subsort Nat < Int .
        subsort NzInt < Int .
        op 0 : -> Nat .
        op -_ : NzNat -> NzInt .
        op -_ : Nat -> Int .
        op -_ : NzInt -> NzInt .
        op -_ : Int -> Int .
        op _+_ : NzNat NzNat -> NzNat .
        op _+_ : NzNat Nat -> Nat .
        op _+_ : Nat NzNat -> Nat .
        op _+_ : Nat Nat -> Nat .
        op _+_ : Int Int -> Int .
        op _*_ : Nat Nat -> Nat .
        op _*_ : Int Int -> Int .
        op s_ : Nat -> NzNat .
        op square : Int -> Nat .
        var M : Nat .
        var I : Int .
        var N : Nat .
        var J : Int .
        var K : NzNat .
        eq - - I = I .
        eq - 0 = 0 .
        eq I + - J = - ( - I + J ) .
        eq I + 0 = I .
        eq K + s M = s ( K + M ) .
        eq - s K + s M = - K + M .
```

```
        eq 0 + I = I .
        eq I * - J = - ( I * J ) .
        eq I * s N = ( I * N ) + I .
        eq I * 0 = 0 .
        eq - I * J = - ( I * J ) .
        eq 0 * I = 0 .
        eq square ( I ) = I * I .
endfm
Proof obligations:
        mb I * I : Nat .
```

In this example, completion has just verified the completeness of the initial system. No new equations have been added. Note however that the last of the equations, `square(I) = I * I`, is nondecreasing. As in the Church-Rosser checker tool [11], nondecreasing equations are detected and the corresponding proof obligations are generated, but that information is not used for orienting the equations. Decreasingness is not taken into account during the completion procedure. Note that a specification can have nondreacreasing equations and still be ground-Church-Rosser if the corresponding proof obligations are verified. In [13], the sorts of the terms are used when orienting the critical pairs, in such a way that no nondecreasing equations are generated. Ganzinger follows a complete different approach in [12], where he proposes to transform order-sorted specifications into equivalent many-sorted ones by using injections to implement subsort relations.

As we have seen in Section 6.8, using the commutativity attribute is tricky. As defined, both the lexicographic path order and the Knuth-Bendix order follow a lexicographic order, and thus changing the order in which the arguments of a commutative function are given may give place very frequently to critical pairs that cannot be oriented. For example,

```
(fmod INT-2 is
    sorts NzNat Nat NzInt Int .
    subsorts NzNat < Nat < Int .
    subsorts NzNat < NzInt < Int .
    op 0 : -> Nat .
    op s_ : Nat -> NzNat .
    op _+_ : Nat Nat -> Nat [comm] .
    op _+_ : Nat NzNat -> Nat [comm] .
    op _+_ : NzNat Nat -> Nat [comm] .
    op _+_ : NzNat NzNat -> NzNat [comm] .
    op _+_ : Int Int -> Int [comm] .
    op -_ : Nat -> Int .
    op -_ : NzNat -> NzInt .
```

```
    op -_ : Int -> Int .
    op -_ : NzInt -> NzInt .
    op _*_ : Nat Nat -> Nat [comm] .
    op _*_ : Int Int -> Int [comm] .
    op square : Int -> Nat .
    vars I J L : Int .
    vars M N : Nat .
    var K : NzNat .
    eq - 0 = 0 .
    eq - - I = I .
    eq I + 0 = I .
    eq s N + s M = s(s(N + M)) .
    eq (- s K) + s M = (- K) + M .
    eq I * 0 = 0 .
    eq I * s N = (I * N) + I .
    eq I * - J = - (I * J) .
    eq square(I) = I * I .
 endfm)

Maude> (check termination INT-2 with LPO[INT-ORDER] * (op _>lpo_ to _>>>_) .)
rewrites: 17441 in 90ms cpu (100ms real) (193788 rewrites/second)

Termination check of INT-2
The specification is terminating.

Maude> (complete INT-2 with LPO[INT-ORDER] * (op _>lpo_ to _>>>_) .)
rewrites: 16811 in 190ms cpu (200ms real) (88478 rewrites/second)

Knuth-Bendix completion of INT-2
The specification cannot be completed.
The critical pair
        eq '_+_['-_['s_['K]], 's_['M]] = '_+_['-_['K], 'M] .
cannot be oriented with the order given.
```

Since the top operators coincide, the arguments are compared in lexicographic order, but not necessarily ordering the arguments as given. In fact, the tool evaluates

```
('-_['s_['K]], 's_['M]) >>> ('M, '-_['K])
```

therefore giving false as result.

6.10 Conditional Specifications: A Second Specification of the Integer Numbers

Let us consider now another specification of the integer numbers, which is also a slight variation of one given by Ganzinger in [12], and in which, in addition to the constant `0` and the unary operators `s_` and `-_`, there is a boolean operator `_<=_` and a `max` operator.

```
(fmod BOOLEAN is
    sort Boolean .
    ops true false : -> Boolean .
 endfm)

(fmod INT-II is
    pr BOOLEAN .
    sorts NzNat Nat NzInt Int .
    subsorts NzNat < Nat < Int .
    subsorts NzNat < NzInt < Int .
    op 0 : -> Nat .
    op s_ : Nat -> NzNat .
    op -_ : Nat -> Int .
    op -_ : NzNat -> NzInt .
    op -_ : Int -> Int .
    op -_ : NzInt -> NzInt .
    op _<=_ : Int Int -> Boolean .
    op max : Int Int -> Int .
    vars I J L : Int .
    vars M K : Nat .
    var N : NzNat .
    eq - 0 = 0 .
    eq - - I = I .
    eq 0 <= N = true .
    eq - M <= N = true .
    eq M <= - N = false .
    eq s M <= s K = M <= K .
    eq M <= K = - K <= - M .
    ceq max(I, J) = I if I <= J == true .
    ceq max(I, J) = J if I <= J == false .
    eq max(I, I) = I .
    eq I <= max(I, J) = true .
    eq I <= max(J, I) = true .
 endfm)
```

In order to complete the specification above, let us consider the lexicographic path order defined by the inequalities in the following module.

```
(fmod INT-II-ORDER is
   pr QID .
   op _>>_ : Qid Qid -> Bool .
   eq '-_ >> '0 = true .
   eq '-_ >> 's_ = true .
   eq '<= >> '0 = true .
   eq '<= >> 's_ = true .
   eq '<= >> 'true = true .
   eq '<= >> 'false = true .
   eq 'max >> '0 = true .
   eq 'max >> 's_ = true .
   eq 'max >> 'true = true .
   eq 'max >> 'false = true .
 endfm)

(view INT-II-ORDER from STRICT-ORDER to INT-II-ORDER is
   op _>>_ to _>>_ .
 endv)
```

As shown by the output of the tool, the specification cannot be completed given such an order.

```
Maude> (complete INT-II
            with LPO[INT-II-ORDER] * (op _>lpo_ to _>>>_) .)
rewrites: 17788 in 200ms cpu (250ms real) (88940 rewrites/second)

Knuth-Bendix completion of INT-II
The specification cannot be completed .
The equation
      eq '_<=_['M, 'K] = '_<=_['-_['K], '-_['M]] .
cannot be oriented with the order given
```

Note that the equation `eq M <= K = - K <= - M` cannot be oriented with the lexicographic path order given, basically because the variables are permuted. One could consider using some alternative to the lexicographic path order, like the multiset path order or some other variant of the recursive path orders (see, for example, [1]).

The Knuth-Bendix order can also help us to complete the `INT-II` module, although we found a different kind of problem due to the conditions of the critical pairs.

```
(fmod INT-II-WEIGHT is
   pr QID .
   pr MACHINE-INT .
```

```
   op weight : Qid -> MachineInt .
   op weight0 : -> MachineInt .
   eq weight('0) = 1 .
   eq weight('s_) = 1 .
   eq weight('-_) = 1 .
   eq weight('_<=_) = 1 .
   eq weight('max) = 1 .
   eq weight('true) = 1 .
   eq weight('false) = 1 .
   eq weight0 = 1 .
 endfm)

(view INT-II-WEIGHT from WEIGHT-FUNCTION to INT-II-WEIGHT is
   op weight to weight .
   op weight0 to weight0 .
 endv)

(view INT-II-ORDER from STRICT-ORDER to INT-II-ORDER is
   op _>>_ to _>>_ .
 endv)

Maude> (complete INT-II
            with KBO[INT-II-ORDER, INT-II-WEIGHT]
                   * (op _>kbo_ to _>>>_) .)
rewrites: 115997 in 1110ms cpu (1120ms real) (104501 rewrites/second)

Knuth-Bendix completion of INT-II
The specification cannot be completed .
The equation
        ceq 'I@ = 'J@
            if '_and_['_==_['_<=_['I@, 'J@], {'true}'Boolean],
                      '_==_['_<=_['I@, 'J@], {'false}'Boolean]]
               = {'true}'Bool .
cannot be oriented with the order given
```

This critical pair cannot be oriented. It is clear that neither `I@ >>> J@` nor `J@ >>> I@`. However, the condition is clearly unsatisfiable. As we pointed out in [11], dealing with conditional critical pairs with conditions that cannot be satisfied or even to reduce the conditions if possible is a tricky matter for which we do not have good methods at the moment. The difficult question is how to prove that two terms are not equal if we do not know whether our specification is confluent. Some research needs to be done in this direction before deciding how to handle these cases.

References

[1] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[2] Leo Bachmair, Nachum Dershowitz, and Jieh Hsiang. Orderings for equational proofs. In *Proceedings of 1st IEEE Symposium on Logic in Computer Science*, pages 346–357. IEEE Computer Society Press, 1986.

[3] Manuel Clavel. *Reflection in General Logics and in Rewriting Logic with Applications to the Maude Language*. PhD thesis, Universidad de Navarra, 1998.

[4] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. Maude: Specification and programming in rewriting logic. Manuscript, SRI International, 1999. Available at `http://maude.csl.sri.com`.

[5] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, and José Meseguer. Metalevel computation in Maude. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of 2nd International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. Available at `http://www.elsevier.nl/locate/entcs/volume15.html`.

[6] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, and José Meseguer. An introduction to Maude (beta version). Manuscript, SRI International, March 1998.

[7] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In José Meseguer, editor, *Proceedings of 1st International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. Available at `http://www.elsevier.nl/locate/entcs/volume4.html`.

[8] Manuel Clavel and José Meseguer. Axiomatizing reflective logics and languages. In Gregor Kiczales, editor, *Proceedings of Reflection'96*, pages 263–288, 1996.

[9] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 244–320. Elsevier, 1990.

[10] Francisco Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Málaga, June 1999.

[11] Francisco Durán and José Meseguer. A Church-Rosser checker tool for Maude equational specifications. Manuscript, 2000.

[12] Harald Ganzinger. Order-sorted completion: the many-sorted way. *Theoretical Computer Science*, 89:3–32, 1991.

[13] Isabelle Gnaedig, Claude Kirchner, and Hélène Kirchner. Equational completion in order-sorted algebras. *Theoretical Computer Science*, 72:169–202, 1990.

[14] Gérard Huet. A complete proof of correctness of the knuth-bendix completion algorithm. *Journal of Computer and System Sciences*, 1(23):11–21, 1981.

[15] Gérard Huet and Dallas Lankford. On the uniform halting problem for term rewriting systems. Technical Report 283, INRIA, 1978.

[16] Donald Knuth and Peter Bendix. Simple word problems in universal algebras. In John Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[17] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.