A quick ITP tutorial*

Manuel Clavel
Sistemas Informáticos y Programación
Universidad Complutense de Madrid
clavel@sip.ucm.es

Miguel Palomino
Sistemas Informáticos y Programación
Universidad Complutense de Madrid
miguelpt@sip.ucm.es

Abstract

The ITP tool is an experimental inductive theorem prover for proving properties of Maude equational specifications, i.e., specifications in membership equational logic with an initial algebra semantics. The ITP tool has been written entirely in Maude and is in fact an executable specification of the formal inference system that it implements.

1 Introduction

Rewriting logic [10, 1] is a flexible formalism for the specification of computational systems in which the dynamic aspects are captured by means of rewrite rules while the static data structures are formalized in its underlying membership equational logic [11]. Maude [3] is a specification language that implements rewriting logic; in particular, it implements membership equational logic through functional modules which can be executed very efficiently.

Mechanical reasoning about such functional modules is supported by the ITP tool, an experimental, interactive, rewriting-based theorem prover written entirely in Maude. The users must guide the ITP by assuming the main decisions in the proof development: namely, when and which a lemma is needed or inductive reasoning is helpful. The ITP, in its turn, assists the users by automatizing the underlying equational and arithmetic reasoning, and by checking the correctness of their proof decisions.

Rewriting based theorem provers like the ITP use term rewriting as their basic proof engine, and they are useful for proving properties of equational specifications. In this regard, a key feature of the ITP is its reflective design [2], that exploits the reflective capabilities of Maude in order to efficiently automatize equational simplification proofs and to cleanly integrate them with proofs in the theory of linear arithmetic with uninterpreted function symbols. The former is accomplished by calling the underlying Maude's default rewriting engine, and the latter by equationally defining a *rewriting strategy* different from Maude's default one. Recently, a graphical interface has been built on top of the ITP to spare the user from the command-line shell required by Maude.

The ITP is being used in the development of two related projects. The first one is the ASIP project, which is based on J. Goguen's seminal work on algebraic semantics of imperative programs. It aims to provide a version of the ITP that may be used to formally specify and verify software. The other project is the development of SCC, an experimental tool for checking the *sufficient completeness* (the property that operations are defined on all valid inputs) of partial specifications written in Maude.

The aim of this paper is to give a quick overview of all these developments, as well as to illustrate how to use the ITP, for which we use several examples. Knowledge of Maude is assumed, but the syntax should be self-explanatory in most examples.

*Research supported by the projects MELODIAS TIC 2002-01167 and MIDAS TIC 2003–0100.

2 Getting started

To run the current version of the ITP you need to have installed the latest version of the Maude system (version 2.1.1), which can be downloaded from the Maude's home page: http://maude.cs.uiuc.edu.

The current version of the ITP is distributed as the gzip-tar-file itp-tool-v*xxx*.tar.gz (where *xxx* is the actual version identifier), which is available at http://maude.sip.ucm.es/itp. To untar this file, type tar xvfz itp-tool-v*xxx*.tar.gz. This command will create a directory itp-tool-v*xxx* with a subdirectory itp-src containing the ITP tool's source files:

```
bash>tar xvfz itp-tool-vxxx.tar.gz
itp-tool-vxxx/
itp-tool-vxxx/itp-src/
itp-tool-vxxx/itp-src/unification.maude
itp-tool-vxxx/itp-src/basic.maude
itp-tool-vxxx/itp-src/check.maude
itp-tool-vxxx/itp-src/ext-mod.maude
itp-tool-vxxx/itp-src/ext-term.maude
itp-tool-vxxx/itp-src/itp-tool.maude
bash>
```

2.1 A first proof

To give a taste of the tool, consider the following specification, that imports the predefined INT module to build of lists of integers:

```
fmod LIST is
  protecting INT .

  sorts NeList List .
  subsort NeList < List .

  op [] : -> List [ctor] .
  op _:_ : Int List -> NeList [ctor] .
  op _++_ : List List -> List .

  var I : Int .
  vars L L' : List .

  eq [] ++ L = L .
  eq (I : L) ++ L' = I : (L ++ L') .
endfm
```

Here, NeList is a subsort of List used to represent the nonempty lists. An obvious property that the specification should satisfy is that concatenation of lists (_++_) is associative. To prove it, once the LIST module has been added to Maude's database we load the ITP with the Maude command in itp-tool and initialize its own database with loop init-itp. Then, the property can be presented to the ITP using the ITP command goal; all ITP commands have to be written enclosed in brackets:[1]

```
(goal list-assoc : LIST |-
    A{L1:List ; L2:List ; L3:List}
     (((L1:List ++ L2:List) ++ L3:List) =
       (L1:List ++ (L2:List ++ L3:List))) .)
```

Here, list-assoc is the name of the goal, LIST is the module in which it is to be proved, and the symbol A (representing ∀) precedes a list of universally quantified variables which have to be annotated with their sorts. Due to parsing restrictions it is convenient to be generous in the use of parentheses; in particular, note that they are used to enclose the terms to be proved equal.

The ITP then outputs

```
=================================
label-sel: list-assoc@0
=================================
A{L1:List ; L2:List ; L3:List}
    ((L1:List ++ L2:List)++ L3:List =
      L1:List ++(L2:List ++ L3:List))
+++++++++++++++++++++++++++++++++
```

indicating that the goal has been correctly processed, has been internally labelled as list-assoc$0 and is ready to be worked upon.

Now we can try to prove the property by structural induction on the first variable, using the ind command.

```
(ind on L1:List .)
```

As will be further explained in Section 4, the ITP then generates a corresponding subgoal for each operator with codomain List that has been declared with the attribute ctor (taking subsorts into account), and selects one

[1] This is required by the LOOP-MODE module, through which all user interfaces are implemented in Maude [3, Chapter 11].

of them as the working subgoal by tagging it with `label-sel`; in this case the first one, corresponding to the empty list `[]`:

```
=================================
label-sel: list-assoc@1.0
=================================
A{L2:List ; L3:List}
    (([]++ L2:List)++ L3:List =
      []++(L2:List ++ L3:List))

=================================
label: list-assoc@2.0
=================================
A{V0#0:Int ; V0#1:List}
((A{L2:List ; L3:List}
   ((V0#1:List ++ L2:List)++ L3:List =
     V0#1:List ++(L2:List ++ L3:List)))==>
 (A{L2:List ; L3:List}
   (((V0#0:Int : V0#1:List)++ L2:List)++
                              L3:List =
     (V0#0:Int : V0#1:List)++
                 (L2:List ++ L3:List))))
+++++++++++++++++++++++++++++++++
```

The working subgoal can be explicitly selected using the ITP command `sel`; e.g. `(sel list-assoc@2.0 .)`

Though the output for the second subgoal is all but clear, notice that the expression before the arrow `==>` corresponds to the induction hypothesis and how the term `V0#0:Int : V0#1:List` is substituted for `V0#1:List` in the subsequent expression.

At this point, we can try to automatically prove the selected subgoal with the command `auto`, that first transforms all variables into fresh constants and then rewrites the terms in both sides of the equality as much as possible by using the equations in the module as rewrite rules.

```
(auto .)
```

The command succeeds, the subgoal is discharged, and the ITP presents us with the remaining subgoal generated by the induction; note how `label` has been replaced by `label-sel`.

```
=================================
label-sel: list-assoc@2.0
=================================
A{V0#0:Int ; V0#1:List}
 ((A{L2:List ; L3:List}
   ((V0#1:List ++ L2:List)++ L3:List =
     V0#1:List ++(L2:List ++ L3:List)))==>
 (A{L2:List ; L3:List}
   (((V0#0:Int : V0#1:List)++ L2:List)++
                              L3:List =
     (V0#0:Int : V0#1:List)++
                 (L2:List ++ L3:List))))
+++++++++++++++++++++++++++++++++
```

We can also try to prove this subgoal automatically with the `auto` command. Again, the ITP succeeds and this completes the proof.

```
(auto .)

q.e.d
+++++++++++++++++++++++++++++++++
```

3 A script safari

The essentials of the ITP have already been covered in the previous section: terms are proved equal by reducing them to syntactically equal terms and structural induction is based on the constructor memberships. Here we continue exploring other ITP commands and illustrate how they are used in several scripts.

3.1 Another induction scheme: `c-ind`

In addition to the `ind` command to reason by induction on the structure of terms, the ITP also provides the `c-ind` command to reason by induction over the natural numbers. This command takes a term of sort `Nat` as argument and generates two subgoals from the original goal: one in which the term is assumed to be equal to `0` and another one in which it states that the goal holds for `N` assuming that it holds for values less than `N`. To illustrate its use, let us extend `LIST` with an operation to determine the length of a list.

```
fmod LIST-LENGTH is
  protecting LIST .

  op length : List -> Nat .

  var I : Int .
  var L : List .
  eq length([]) = 0 .
  eq length(I : L) = 1 + length(L) .
endfm
```

Now, the associativity of the operator `_++_` can alternatively be proved by using `c-ind`:

```
(goal list-assoc : LIST-LENGTH |-
  A{L1:List ; L2:List ; L3:List}
    (((L1:List ++ L2:List) ++ L3:List) =
      (L1:List ++ (L2:List ++ L3:List))) .)

(c-ind on (length(L1:List)) .)

=================================
label-sel: list-assoc@1.0
=================================
A{L1:List ; L2:List ; L3:List}
  ((length(L1:List)= 0)==>
     ((L1:List ++ L2:List)++ L3:List =
       L1:List ++(L2:List ++ L3:List)))

=================================
label: list-assoc@2.0
=================================
A{V0#0:Nat}
 ((A{L1:List ; L2:List ; L3:List}
   ((length(L1:List)< V0#0:Nat = true)==>
    ((L1:List ++ L2:List)++ L3:List =
       L1:List ++(L2:List ++ L3:List))))==>
  (A{L1:List ; L2:List ; L3:List}
   ((length(L1:List)= V0#0:Nat)==>
   ((L1:List ++ L2:List)++ L3:List =
     L1:List ++(L2:List ++ L3:List)))))
+++++++++++++++++++++++++++++++++
```

The first (and selected) subgoal simply assumes that `length(L1:List)` is `0`. Using `auto` seems to have no effect except for the transformation of the variables into constants (the second subgoal is not modified and we do not show it).

```
(auto .)

=================================
label-sel: list-assoc@1.0
=================================
(L1*List ++ L2*List)++ L3*List =
    L1*List ++(L2*List ++ L3*List)

=================================
label: list-assoc@2.0
...
```

To try to understand what is going on we can use the `show-all` command, which outputs the functional module the ITP is currently reasoning about.

```
(show-all .)
```

From its output—that we do not show—it is apparent that the resulting module is identical to the original `LIST-LENGTH` except for the additional declaration of the new constants `L1*List`, `L2*List`, and `L3*List`, and the equation `length(L1*) = 0`. Obviously, this additional information is not enough to further reduce any of the terms in the goal.

To proceed with the proof we must supply the ITP with additional guidelines to follow and what turns out to be useful in this case is to make a case analysis on the structure of `L1*List`. For this we can use the command `ctor-split`, which replaces the subgoal with the following ones, corresponding to the constructors for `List`:

```
(ctor-split on L1*List .)

=================================
label-sel: list-assoc@1.1.1.1.1.1.0
=================================
(L1*List =[])==>
   ((L1*List ++ L2*List)++ L3*List =
     L1*List ++(L2*List ++ L3*List))

=================================
label: list-assoc@1.1.1.1.1.2.0
=================================
A{V1#0:Int ; V1#1:List}
  ((L1*List = V1#0:Int : V1#1:List)==>
   ((L1*List ++ L2*List)++ L3*List =
     L1*List ++(L2*List ++ L3*List)))
```

Now we can try to prove the selected subgoal, which assumes that `L1*List` is the empty list, and `auto` easily succeeds.

```
(auto .)
```

The other subgoal generated by `ctor-split`, `list-assoc@1.1.1.1.1.2.0`, deserves closer attention. The list `L1*List` is now built using `_:_` and is not clear at all why the terms at both sides of the equality symbol should reduce to a common one.

But recall that this subgoal has arisen while we are trying to prove the first subgoal generated by `c-ind`, that is, the one in which `length(L1*List)` is `0`, and this is inconsistent with `L1*List` being constructed with `_:_`. On the one hand, by the induction hypothesis the current module contains the equation

```
length(L1*List) = 0
```

On the other hand, if the `auto` command is used to reduce the subgoal the following chain of reductions is triggered:

```
length(L1*List)
    = length(V1#0:Int : V1#1:List)
    = 1 + length(V1#1:List)
```

At this point, the decision procedures for linear arithmetic that are interwoven in the ITP rewrite strategy discover the inconsistency

```
0 = 1 + length(V1#1:List)
```

Thus, `auto` notices the inconsistency and automatically discharges the subgoal.

```
(auto .)
```

At this point we are left with the proof of the second subgoal generated by `c-ind`, `list-assoc@2.0`, which proceeds exactly like that for the first one. After applying `auto`, it is necessary to make a case analysis whose subcases can be discharged with `auto`. (This time, it is the proof of the first case which succeeds by inconsistency.)

```
(auto .)
(ctor-split on L1*List .)
(auto .)
(auto .)
```

In this example, the script produced by `c-ind` is certainly more cumbersome than the one for `ind`. However, there are proofs (e.g., in the mergesort algorithm) where `c-ind` gives rise to proofs that cannot easily be done by structural induction.

3.2 Sorts

The key feature of membership equational logic is its capability to define sorts by means of membership axioms. This allows the specification of very precise types, like sorted lists, and of operations that reflect that typing.

The ITP can also reason about sorts, for which it provides the command `ctor-term-split` which will be illustrated below. Reasoning about sorts is subtle; ideally, operations should be defined at the kind level and the operations' sorts defined by means of memberships. As of yet, however, this is not fully supported by the ITP so that in the example below we will make use of supersorts instead of kinds.[2]

Also, this example is more involved than the previous ones and, in addition to illustrate the treatment of sorts, it will serve to introduce two additional ITP commands: `lem`, to introduce auxiliary lemmas, and `split` for case analysis.

Consider the following specification of sorted lists.

```
fmod ORDERED-LIST is
  protecting INT .
  sort Int? OrdList OrdList? .
  subsort Int < Int? .
  subsort OrdList < OrdList? .

  op [] : -> OrdList? .
  op _:_ : Int? OrdList? -> OrdList? .
  op insert : Int? OrdList? -> OrdList? .
  op insertion-sort : OrdList? -> OrdList? .

  vars E1 E2 I J : Int? .
  vars OL OL1 OL2 : OrdList? .

  mb [] : OrdList .
  cmb (I : []) : OrdList if I : Int .
  cmb (I : J : OL) : OrdList
    if I : Int /\ J : Int
       /\ I <= J = true
       /\ (J : OL): OrdList .

  eq insertion-sort([]) = [] .
  eq insertion-sort(I : OL) =
     insert(I, insertion-sort(OL)) .
```

[2] Intuitively, kinds represent collections of sorts; see [11] for a detailed discussion, but also Section 4.

```
  eq insert(I, []) = (I : []) .
  ceq insert(I, (J : OL)) = I : J : OL
    if I <= J .
  ceq insert(I,(J : OL)) = J : insert(I,OL)
    if I > J .
endfm
```

OrdList is intended to capture ordered lists whereas its supersort OrdList? comprises arbitrary ones. Note that OrdList is directly defined by membership assertions and not constructors; this will be explained in Section 4.

We want the list returned by a call to insertion-sort to be actually ordered so that insertion-sort(L) : OrdList for all lists L, and we set ourselves to prove it. (For the operation insertion-sort to be well-defined it would also be necessary to prove that the resulting list is a permutation of the original one: we leave this proof as an exercise to the reader.)

```
(goal sorted : ORDERED-LIST |-
 A{L:OrdList?}
 ((insertion-sort(L:OrdList?)) : OrdList) .)
```

The proof proceeds by structural induction on L; a case for each of the axioms that define OrdList—since it is a subsort of OrdList?—is generated. For the empty list and for a list with a single element the result follows immediately using auto; in the inductive step, however, after applying auto we arrive to:

```
================================
label-sel: sorted@3.1.1.1.1.1.1.1.1.1.0
================================
insert(V0#0*Int?,insert(V0#1*Int?,
   insertion-sort(V0#2*OrdList?))): OrdList
++++++++++++++++++++++++++++++++
The goal labeled sorted@3.1.1.1.1.1.1.1...
is not an identity
```

At this point, the ITP can no longer reduce the term and gets stuck.

Let us think about what remains to be proved. By the induction hypothesis, the list insertion-sort(V0#2*OrdList?) is sorted and, since we believe the equations for insert are correct, we would expect

```
insert(V0#1*Int?,
       insertion-sort(V0#2*OrdList))
```

to be sorted as well, even though it can be reduced no further; the same argument would apply to the insertion of V0#0*Int? as well. What we need is precisely this, an auxiliary result that states that inserting a new element into an ordered list using insert results in another ordered list. Such a lemma can be introduced in the ITP with the lem command and has to be proved for the reasoning to be sound.

```
(lem insert-orderedlist :
  A{E:Int? ; OL:OrdList?}
   ((((E:Int?): Int) &
     ((OL:OrdList?): OrdList))
   =>
   ((insert(E:Int?, OL:OrdList?)): OrdList))
.)
```

```
================================
label-sel: insert-orderedlist@0
================================
A{E:Int? ; OL:OrdList?}
   (((OL:OrdList? : OrdList) &
     (E:Int? : Int))==>
   (insert(E:Int?,OL:OrdList?): OrdList))
```

```
================================
label: sorted@3.1.1.1.1.1.1.1.1.1.0
================================
insert(V0#0*Int?,insert(V0#1*Int?,
   insertion-sort(V0#2*OrdList?))): OrdList
++++++++++++++++++++++++++++++++
```

The syntax for lemmas is the same as for goals, except for the fact that no module is specified. Note that the statement to be proved in this case is not just an equality, but an implication where the symbol & is used to separate conjunctions in the antecedent. Also, note that the lemma has become the current goal. (From now on we will only show the current subgoal.)

Again, we can think of proving the lemma with the ind command and, as happened for the main goal, auto discharges the case generated for the empty list. However, in the case of the singleton list, it merely transforms the goal into

```
================================
label-sel: insert-orderedlist@2.1.1.1.1...
================================
insert(E*Int?,V1#0*Int? :[]): OrdList
```

If we take a look at the equations for `insert`, the reason why this term cannot be reduced becomes apparent: the two equations that might apply are conditional and depend on whether `E*Int <= V1#0*Int` or `E*Int > V1#0*Int`. Thus, to discharge this subgoal it is necessary to reason by cases according to whether `E*Int? <= V1#0*Int?` is `true` or not. For that, the ITP offers the command `split`:

```
(split on (E*Int? <= V1#0*Int?) .)
```

This prompts the ITP to replace the working subgoal with the following ones:

```
================================
label-sel:
 insert-orderedlist@2.1.1.1.1.1.1.1.1.0
================================
insert(E*Int?,V1#0*Int? :[]): OrdList

================================
label:
 insert-orderedlist@2.1.1.1.1.1.1.1.2.0
================================
insert(E*Int?,V1#0*Int? :[]): OrdList
```

Note that the goals themselves are the same; the difference lies in the internal modules these goals are about: for the first one, the equation `E*Int? <= V1#0*Int? = true` has been added—which can be checked using `show-all`—while in the second the inequality is `false`. Now, both subgoals can be automatically discharged with `auto`.

```
(auto .)
(auto .)
```

The situation for the inductive step is similar: after applying `auto` we are left with

```
================================
label-sel: insert-orderedlist@3.1.1...
================================
insert(E*Int?,V1#0*Int? : V1#1*Int? :
                  V1#2*OrdList?): OrdList
```

Again, we need to reason by cases.

```
(split on (E*Int? <= V1#0*Int?) .)
(auto .)
(auto .)
```

This time, however, the second `auto` does not discharge the goal but presents us with

```
================================
label-sel: insert-orderedlist@3.1.1...
================================
V1#0*Int? : insert(E*Int?,V1#1*Int? :
                V1#2*OrdList?): OrdList
```

Under the current assumption that `E*Int? <= V1#0*Int?` is `false` and the induction hypothesis that the list `insert(E*Int?,V1#1*Int? : V1#2*OrdList?)` is ordered, it is clear that the equality holds. What we need is yet another lemma that makes this observation a general and explicit statement.

```
(lem insert-sortedlist-aux :
  A{E1:Int? ; E2:Int? ; OL:OrdList?}
  ((((E1:Int?): Int) & ((E2:Int?): Int) &
    ((OL:OrdList?): OrdList) &
    ((E1:Int? <= E2:Int?) = (true)) &
    ((E1:Int? : OL:OrdList?): OrdList) =>
   ((E1:Int? : insert(E2:Int?, OL:OrdList?))
                            : OrdList))) .)
```

As usual, we prove it by induction on the structure of the list. The case generated for the empty list is trivially discharged with `auto` whereas for the singleton list a case analysis is needed (`---` introduces a comment):

```
(ind on OL:OrdList? .)

--- case 1
(auto .)
-----------
--- case 2
(auto .)
(split on (E2*Int? <= V2#0*Int?) .)
--- case true
(auto .)
--- case false
(auto .)
```

Unfortunately (and surprisingly!) the last `auto` cannot discharge the corresponding subgoal, but presents us with

```
================================
label-sel: insert-sortedlist-aux@2.1.1...
================================
E1*Int? : V2#0*Int? : E2*Int? :[]: OrdList
```

So what is going on? This step corresponds to the case in which we have the singleton list `V2#0*Int? : []`; by one of the hypothesis of the lemma, `E1*Int? : V2#0*Int?` is sorted, and since this is the second branch of the `split`, `V2#0*Int? < E2*Int?` is `true`. It is then clear that the list

```
E1*Int? : V2#0*Int? : E2*Int? :[]
```

is sorted, so why can't the ITP prove it? The reason is that nowhere is explicitly stated that `E1*Int? <= V2#0*Int?` is `true`; this information has to be extracted from the fact that `E1*Int? : V2#0*Int?` is sorted, by using the command `ctor-term-split`:

```
(ctor-term-split on
        (E1*Int? : V2#0*Int? : []) .)
```

This prompts the ITP to make explicit all assumptions about the terms involved, based on the membership axioms that define the sort `OrdList`; the resulting subgoal can then be easily proved with `auto`.

```
================================
label-sel: insert-sortedlist-aux@2.1.1...
================================
((((V2#0*Int? :[]: OrdList) &
   (V2#0*Int? : Int)) &
   (E1*Int? : Int)) &
   (E1*Int? <= V2#0*Int? = true)) ==>
(E1*Int? : V2#0*Int? : E2*Int? :[]: OrdList)
```

The rest of the proof of the lemma proceeds along lines that should be familiar by now; no new auxiliary results are needed and we just present the necessary commands: we encourage the reader to try to obtain them by himself.

```
--- case 3
(auto .)
(split on (E2*Int? <= V2#0*Int?) .)
--- case true
(auto .)
--- case false
(auto .)
(ctor-term-split on (E1*Int? : V2#0*Int? :
              V2#1*Int? : V2#2*OrdList?) .)
(auto .)
```

Once the last `auto` in the script above has been executed, the ITP prompts us with the goal at which point the proof was interrupted to introduce the lemma `insert-sortedlist-aux`; with this result at our disposal we can discharge it with `auto`. This brings forward the goal `sorted@3.1.1.1.1.1.1.1.1.1.0` and, similarly, the ITP can finally prove it with `auto` by using `insert-orderedlist` and complete the proof.

3.3 Quantifiers

Even though all the examples considered so far have consisted of universally quantified formulas, the ITP can also deal with existential quantifiers: we illustrate its use with an example borrowed from the PVS's tutorial [5, Section 4.3].

We wish to prove that any postage requirement of 8 cents or more can be met solely with 3 and 5 cent stamps, i.e., is the sum of some multiple of 3 and some multiple of 5. As such, this is simply a property about natural numbers and the corresponding specification is trivial. (The module `INT` is imported instead of `NAT` because the proof needs the subtraction operator, which is not available in `NAT`.)

```
fmod STAMP is
  protecting INT .
endfm
```

Using `E` as the existential quantifier, the property is then stated as

```
(goal stamps : STAMP |- A{I:Nat}
  (E{Three:Nat ; Five:Nat}
    ((I:Nat + 8) =
     ((3 * Three:Nat) + (5 * Five:Nat)))) .)
```

and the proof proceeds by induction on `I:Nat`:

```
(ind on I:Nat .)
```

```
================================
label-sel: stamps@1.0
================================
E{Three:Nat ; Five:Nat}
   (0 + 8 = 3 * Three:Nat + 5 * Five:Nat)

================================
label: stamps@2.0
================================
A{V0#0:Nat}(
  (E{Three:Nat ; Five:Nat}
     (V0#0:Nat + 8 =
          3 * Three:Nat + 5 * Five:Nat))
  ==>
  (E{Three:Nat ; Five:Nat}
     (s V0#0:Nat + 8 =
          3 * Three:Nat + 5 * Five:Nat)))
```

Clearly, letting Three:Nat and Five:Nat both be 1 fulfills the base case; such instantiation is communicated to the ITP through the e-inst command:

```
(e-inst with ((Three:Nat <- (1)) ;
              (Five:Nat <- (1))) .)
```

This replaces the goal stamps@1.0 with the following three ones: while the first is but the instantiated goal, the two others make sure that the terms used in the substitution have the right sorts.

```
================================
label: stamps@1.0
================================
0 + 8 = 3 * 1 + 5 * 1

================================
label-sel: stamps@1.1.0
================================
1 : Nat

================================
label: stamps@1.2.0
================================
1 : Nat
```

All three cases are discharged with auto:

```
(auto .)
(auto .)
(auto .)
```

For the inductive step stamps@2.0, we need to find Three:Nat and Five:Nat such that

```
s V0#0:Nat + 8= 3 * Three:Nat + 5 * Five:Nat
```

assuming that there exist natural numbers T:Nat and F:Nat such that

```
V0#0:Nat + 8 = 3 * T:Nat + 5 * F:Nat
```

Then, if F:Nat is 0 we can take Five:Nat to be 2 and T:Nat to be equal to Three:Nat - 3; otherwise, we obtain the result by making Five:Nat equal to F:Nat - 1 and Three:Nat to T:Nat + 1.

The ITP script mimics quite closely the proof above. First, auto transforms the variable V0#0:Nat in stamsp@2 into a constant and adds the induction hypothesis to the specification, removing it from the implication:

```
================================
label-sel: stamps@2.0
================================
E{Three:Nat ; Five:Nat}
 (s V0#0*Nat + 8 = 3*Three:Nat + 5*Five:Nat)
```

Since the induction hypothesis is *not* a formula in membership equational logic, it cannot be added to the STAMP module and hence does not show up in the show-all command. However, the ITP internally keeps a link between the module and the formula which, at present, can only be revealed with the Maude command cont[3] and that can be used in this case to find out that the induction hypothesis has been labeled as hyp-0. We can now use e-inst to remove the existential quantifier from it

```
(e-inst hyp-0 .)
```

which results in a new module that contains the equation

```
V0#0*Nat + 8 =
   3 * Three!hyp-0*Nat + 5 * Five!hyp-0*Nat
```

as can be checked with show-all.

The rest of the proof is straightforward: we simply have to distinguish cases according to whether Five!hyp-0*Nat is zero or not, and instantiate Three:Nat and Five:Nat accordingly.

```
(split on ((Five!hyp-0*Nat) <= 0) .)

--- Five!hyp-0*Nat <= 0 = true (i.e it is 0)
(e-inst with
 ((Three:Nat <- (_-_(Three!hyp-0*Nat, 3))) ;
```

[3] This command shows the term representing the ITP's internal state [3, Chapter 11]

```
  (Five:Nat <- (2))) .)
(auto .)
(auto .)
(auto .)

--- Five!hyp-0*Nat <= 0 = false
(e-inst with
 ((Three:Nat <- (_+_(Three!hyp-0*Nat, 2))) ;
  (Five:Nat <- (_-_(Five!hyp-0*Nat, 1)))) .)
(auto .)
(auto .)
(auto .)
```

There is also a command `a-inst` for universal quantifiers: the examples we currently have are more involved and can be found at the ITP's webpage.

4 Constructor and defined memberships and the `ind` command

The behavior of the `ind` command (and for the same reasons the `ctor-split` and `ctor-term-split` commands) deserves a closer look: how does the ITP generate the inductive subgoals? In the `LIST` module, the operators `[]` and `_:_` are declared to be constructors of the sorts `List` and `NeList`, respectively, by means of the attribute `ctor`; since `NeList` is a subsort of `List`, this automatically makes `_:_` a constructor of `List` as well. Then, when asked to prove a goal by structural induction on a variable of sort `List`, the `ind` command generates two subgoals:

- The first one, that corresponds to the base case, is obtained by replacing the variable with the constant `[]` in the goal.
- The second one, which is somewhat more obscure and corresponds to the inductive step, is built as an implication where the antecedent and the consequent arise from the original goal by replacing the variable with a fresh one in the first case, and with a term constructed using `_:_` in the second.

In general, the situation is a bit more complicated and requires the notion of a constructor membership. In Maude, an operator declaration

$$\texttt{op}\ f : s_1 \cdots s_n\ \texttt{->}\ s_0$$

is logically equivalent to a declaration

$$\texttt{op}\ f : [s_1] \cdots [s_n]\ \texttt{->}\ [s_0]$$

at the kind level ($[s_i]$ is the kind to which s_i belongs), together with a membership axiom

$$\texttt{mb}\ f\texttt{(}x_1\texttt{:}s_1\texttt{,}\ \ldots\texttt{,}x_n\texttt{:}s_n\texttt{)}\ \texttt{:}\ s_0\ \texttt{.}$$

The ITP, in addition, distinguishes those operator declarations that contain the `ctor` attribute from those that do not, and tags the membership axioms associated to the latter with the label `metadata "dfn"`. Hence, the `LIST` module is interpreted by the ITP as:

```
fmod LIST-MB is
  protecting INT .

  sorts NeList List .
  subsort NeList < List .

  op [] : -> [List] .
  op _:_ : [Int] [List] -> [NeList] .
  op _++_ : [List] [List] -> [List] .

  var I : Int .
  vars L L' : List .

  mb [] : List .
  mb (I : L) : NeList .
  mb (L ++ L') : List [metadata "dfn"] .

  eq [] ++ L = L .
  eq (I : L) ++ L' = I : (L ++ L') .
endfm
```

The *defined memberships* of a functional module are the memberships in the transformed module that are tagged with the label `metadata "dfn"`; the remaining ones are called *constructor memberships*. When the `ind` command is used to reason by structural induction on a variable of sort s, it generates the goals that correspond to the inductive cases from the constructor memberships associated to s and to all its subsorts.

The defined memberships in a specification provide in a handy manner helpful information for the ITP to reason with, but in "good" specifications they *should* actually be redundant and deducible from the rest of the specification (for example, it is clear that the concatenation of two lists should also be a list). For this situation to actually hold, the equations that define the operations have to thoroughly consider all possibilities so that every term eventually reduces to a canonical form to which a constructor membership applies. This is the sufficient completeness problem that we illustrate in the next section; see also Section 7.

4.1 Defined memberships revisited

To stress how defined memberships are treated by the ITP, let us extend the `LIST-MB` module with an operator `empty?` that checks whether a list is empty or not. Admittedly, since we already have a sort `NeList` to distinguish nonempty lists, the use of the operator `empty?` in this example is a bit contrived; let us stick with it for the sake of the argument.

```
fmod LIST-MB is
  protecting INT .

  sorts NeList List .
  subsort NeList < List .

  op [] : -> [List] .
  op _:_ : [Int] [List] -> [NeList] .
  op _++_ : [List] [List] -> [List] .
  op empty? : List -> Bool .

  var I : Int .
  vars L L' : List .

  mb [] : List .
  mb (I : L) : NeList .
  mb (L ++ L') : List [metadata "dfn"] .

  eq [] ++ L = L .
  eq (I : L) ++ L' = I : (L ++ L') .

  eq empty?([]) = true .
  eq empty?(I : L) = false .
endfm
```

We can now try to show that the term `empty?`(L) is of sort `Bool` for every list L.

```
(goal empty-bool : LIST-MB |-
    A{L:List} ((empty?(L:List)) : Bool) .)
```

As expected, since the specification includes the declaration

```
  op empty? : List -> Bool .
```

the goal can be discharged by just using `auto`. What would have happened, however, if we had forgotten to add the equation

```
  eq empty?([]) = true .
```

to the specification? Obviously, this would have left undefined `empty?` over the empty list with the undesired consequence that now not all terms of the form `empty?`(L) are of sort `Bool`. Nonetheless, the ITP would have still proved it! This shows that even though the defined memberships of a functional module are not used to generate inductive cases, they are nevertheless applied whenever possible. And this is precisely the sufficient completeness problem: to guarantee that the equations for the defined operators in the specification consider all possible cases so that defined memberships would become irrelevant from a proof-theoretical point of view. In this situation, the defined membership

```
  mb empty?(L) : Bool [metadata "dfn"] .
```

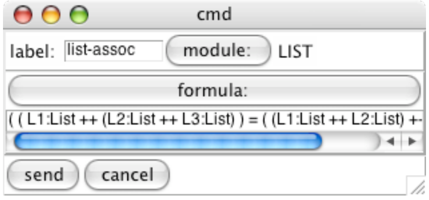
associated to the declaration of `empty?` is *not* redundant and cannot be derived from the rest of the specification.

5 A graphical interface for the ITP

Since the ITP is but a Maude specification, the standard way of interacting with the tool is by loading Maude in a shell and typing the commands to be executed. Currently, however, the ITP is being endowed with a graphical interface that will spare the users from the need to interact directly with Maude and to learn the ITP syntax. Though not yet completed,[4] we offer here a glimpse of what the interface will look like.

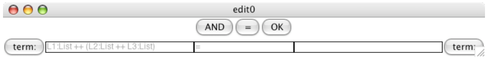
Selection of the functional module to be worked upon will be made through the use of

[4] We expect to have it ready in the early future.



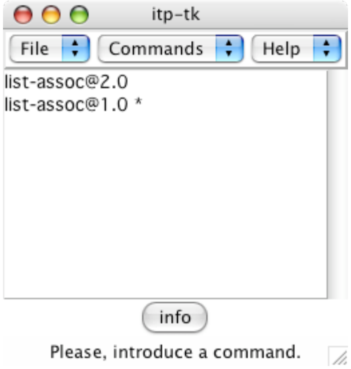
label: list-assoc
module: LIST
formula:
((L1:List ++ (L2:List ++ L3:List)) = ((L1:List ++ L2:List) +-

Figure 1: Introducing a goal



L1:List ++ (L2:List ++ L3:List) | = |

Figure 2: Introducing a formula



list-assoc@2.0
list-assoc@1.0 *

Please, introduce a command.

Figure 3: Tree of pending goals

menus. Then, a series of pop-up windows like the one shown in Figure 1 will guide the user to introduce the goal to be proved. In particular, Figure 2 shows how formulas will be entered through a formula editor that will avoid the error-prone process of typing them directly in the shell.

Once the goal has been introduced, the flow of the proof is controlled by a window like that shown in Figure 3. There the user can choose which ITP command to apply from the middle combo box, with the resulting subgoals building up the proof tree—through which the user can navigate—in the center of the window. The current subgoal can be selected by clicking over it and appears marked with a *; pressing then the `info` button will open a window showing the concrete goal as well as the functional module in which it has to be proved.

6 ASIP: algebraic semantics of imperative semantics

The ASIP project is based on Goguen and Malcolm' work on algebraic semantics of imperative programs [7]. It aims to provide a version of the ITP that may be used to formally specify and verify software; in this approach, the semantics of imperative semantics is defined by specifying a class of abstract machines and giving equational axioms which specify the effect of programs on such machines.

Goguen and Malcolm's original work used OBJ, but the fact that OBJ was not a theorem prover and that some things had to be done manually that should be done automatically led Goguen to point out (in [6], a work upon which [7] is based) that "it would be useful to construct a verification interface for OBJ to generate proof scores that use only techniques already shown correct." The ASIP+ITP tool is an extension of the ITP in which such verification interface is made available; a detailed presentation of the tool is provided in [4].

7 SCC: a sufficient completeness checker

SCC [8] is an experimental tool for checking the *sufficient completeness* of partial specifications written in Maude. Sufficient completeness is the property that operations are defined on all valid inputs. It is an important property both for developers of specifications, to check that they have not missed a case while defining the operations, and to inductive theorem provers, to check the soundness of a proposed induction scheme. The SCC tool has been written in Maude and relies on Maude's reflective capabilities and the ITP tool. The SCC tool is included in the latest distribution of the ITP tool and can be executed both in stand-alone mode and through appropriate commands during an ITP session. When applied to a module, it returns a set of equations that, if valid, guarantee its sufficient completeness.

For example, for the version of the `LIST-MB` module in Section 4 that contains the declaration of `empty?` at the sort level, but with the equation `empty?([]) = true` missing, the result would be:

```
(scc LIST-MB .)

================================
CTOR-LIST-MB$1.0
================================

|- true = false

++++++++++++++++++++++++++++++++
```

Though not very informative, this output is enough to point out that the specification is not sufficiently complete.

8 Conclusions

In this tutorial, we have given a quick overview of the ITP tool, an interactive, rewriting-based theorem prover that can be used to prove inductive properties of membership equational specifications. Some of the inference rules implemented in the ITP tool borrows from techniques introduced by Goguen for proving properties of order-sorted equational specifications using the OBJ system [6].

The ITP is still an experimental tool, but the results obtained so far are quite encouraging. The ITP tool is the only theorem prover at present that supports reasoning about membership equational logic specifications. The key feature of membership equational logic is its capability to define sorts by means of membership axioms. This allows the specifications of very precise types, like sorted lists, and of operations that reflect that typing.

The powerful integration of term rewriting with a decision procedure for linear arithmetic with uninterpreted function symbols, while also available in other rewriting-based theorem provers like the Rewriting Rule Laboratory [9], has been easily and efficiently implemented in the ITP by exploiting the reflective design of the tool and the reflective capabilities of the Maude system. This fact has encouraged us to plan to add other decision procedures to our tool in the near future.

Finally, the graphical interface, currently under construction, will definitely help the already growing ITP's user community as a much needed help to embark on more complex verification challenges.

Acknowledgments. We thank Marina Egea for very detailed on a previous draft.

References

[1] R. Bruni and J. Meseguer. Generalized rewrite theories. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Automata, Languages and Programming. 30th International Colloquium, ICALP 2003*, volume 2719 of *LNCS*, pages 252–266. Springer, 2003.

[2] M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications.* CSLI Publications, 2000.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude manual (version 2.1.1). `http://maude.cs.uiuc.edu/manual/`, 2005.

[4] M. Clavel and J. Santa-Cruz. ASIP+ITP: A verification tool based on algebraic semantics. In this volume.

[5] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida*, 1995.

[6] J. A. Goguen. OBJ as a theorem prover with applications to hardware verification. In *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 218–267. Springer-Verlag, 1989.

[7] J. A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs.* MIT Press, 1996.

[8] J. Hendrix. The SCC tool's home page. `http://maude.cs.uiuc.edu/tools/scc/`

[9] D. Kapur and M. Subramaniam. New uses of linear arithmetic in automated theorem proving by induction. *Journal of Automated Reasoning*, 16(1-2):39–78, 1996.

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

[11] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *12th International Workshop, WADT'97*, volume 1376 of *LNCS*, pages 18–61. Springer, 1998.

A ITP commands

- `(goal` *Label* `:` *Entailment* `.)`. This command introduces a goal *Label*to be proved with the ITP. The expression *Entailment* has the form *IdMod* `|- A{`x_1 : $s1, \ldots, x_n$: s_n`}` *expression*, with *IdMod* the name of the module the goal is about, $x_1, \ldots, x_n$ variables, and *expression* a Horn clause built with `&` and `=>`.

- `(lem` *Label* `:` *LEntailment* `.)`. This command introduces a lemma, named *Label*, to be proved with the ITP. The expression *LEntailment* has the form `|- A{`x_1 : $s1, \ldots, x_n$: s_n`}` *expression*, with $x_1, \ldots, x_n$ variables and *expression* a Horn clause built with `&` and `=>`.

- `(auto .)`. It first transforms all the variables into fresh constants and then reduces the terms in the goal as much as possible by using the equations in the module as rewrite rules. When necessary, decision procedures are applied to check if the conditions of conditional equations are satisfied.

- `(c-ind on Nat-`*Term* `.)`. It generates two subgoals from the current one. The first one consists in the original goal assuming that the value of `Nat-`*Term* is `0`. The second subgoal states that if the original one holds when the value of `Nat-`*Term* is less that `N`, then it also holds when the value of `Nat-`*Term* is `N`.

- `(ctor-split on` *Const* `.)`. The goals are generated from the constructor memberships for the sort of *Const* and all its subsorts, in the same way as for `ind`, but no induction hypothesis is generated.

- `(e-inst` *Label* `.)`. The existential formula *Label*, which is a hypothesis, is "instantiated" with values that satisfy it.

- `(e-inst with` *Substitution* `.)`. It instantiates the current existential goal with the given *Substitution*.

- `(ind on` *Var* `.)`. The goals are generated from the constructor memberships for the sort of *Var* and all its subsorts. Base cases correspond to unconditional memberships of the form `mb` $t : s$ `.`, and give rise to new subgoals by replacing the *Var* in the current goal with t. The inductive steps correspond to conditional memberships.

- `(scc` *IdMod* `.)`. It first calls on the functional module named *IdMod* the function `checkCompleteness`, which implements the SCC's sufficient completeness analizer. Then, it converts the resulting proof obligations into a set of goals, which are all associated with the constructor submodule of *IdMod*. Finally, it eliminates from the state of the proof those goals that can be proved automatically using the ITP's `auto` command.

- `(show-all .)`. It outputs the active module in the ITP's module database.

- `(split on (Bool-`*Term*`) .)`. It splits the current goal in two: one in which `Bool-`*Term* is assumed to be `true` and another one in which it is assumed to be `false`.