Introducing the ITP Tool: a Tutorial*

Manuel Clavel
(Universidad Complutense de Madrid, Spain
clavel@sip.ucm.es)

Miguel Palomino
(Universidad Complutense de Madrid, Spain
miguelpt@sip.ucm.es)

Adrián Riesco
(Universidad Complutense de Madrid, Spain
adririesco@yahoo.es)

Abstract: We present a tutorial of the ITP tool, a rewriting-based theorem prover that can be used to prove inductive properties of membership equational specifications. We also introduce membership equational logic as a formal language particularly adequate for specifying and verifying semantic data structures, such as ordered lists, binary search trees, priority queues, and powerlists. The ITP tool is a Maude program that makes extensive use of the reflective capabilities of this system. In fact, rewriting-based proof simplification steps are directly executed by the powerful underlying Maude rewriting engine. The ITP tool is currently available as a web-based application that includes a module editor, a formula editor, and a command editor. These editors allow users to create and modify their specifications, to formalize properties about them, and to guide their proofs by filling and submitting web forms.

Key Words: inductive theorem proving, semantic data structures, membership equational logic, ITP
Category: F.3.1, F.4.2

1 Introduction

The ITP tool is a theorem prover that can be used to prove properties of membership equational specifications, as well as incompletely specified algorithms on them, as a way to support incremental development of specifications. As an introduction to the ITP tutorial, we recall in Section 1.1 the basic concepts underlying the equational specification of data structures; then, in Section 1.2 we highlight the advanced concepts provided by membership equational logic to specify semantic data structures; finally, in Section 1.3, we discuss the models of the membership equational specifications with respect to which the properties are verified in the ITP tool.

* Research supported by Spanish MEC Projects TIC2003-01000, TIN2005-09207-C03-03, and by Comunidad de Madrid Program S-0505/TIC/0407.

1.1 Specification and Verification of Data Structures

The equational specification of data structures starts with the declaration of an alphabet of symbols with which to build terms to represent the elements of the data structure. These symbols are called the *constructor* symbols, and (ground) *constructor-terms* are those built only with constructor symbols. Since non-trivial data structures contain data of different types, contructors are typically declared along with the *sorts* of their arguments and of their results.

Example 1. Consider an equational specification of lists of integers. To represent lists of integers as terms, we can declare the constant symbol `nil` and the binary operator `cons` as constructors; the constant `nil` is a term of sort `List`, and the operator `cons` takes terms of sort `Int` and `List` and builds terms of sort `List`. For the sake of the example, we suppose that ...,`-1`,`0`,`1`,... have been also declared as constant symbols of the sort `Int`. In this specification, the empty list [] is represented by the term `nil` and the list $[1, 2]$ is represented by the term `cons(1, cons(2, nil))`.

Interesting data structures provide operations over elements of the data type. Their equational specifications include the declaration of the symbols that represent these operations, along with their definitions. The union of the alphabets of constructor symbols and of operation symbols is called the *signature* of the specification. The clauses that define the operations of a data structure are called the *axioms* of the specification.

Example 2. Consider an equational specification of lists of integers with an operation that returns the concatenation of two lists. To represent this operation we can add to the alphabet in Example 1 a binary symbol `append` and introduce in the specification the following equations:

$$\forall\{L\}(\texttt{append}(\texttt{nil}, L) = L)$$
$$\forall\{I, L, L'\}(\texttt{append}(\texttt{cons}(I, L), L') = \texttt{cons}(I, \texttt{append}(L, L'))) ,$$

where I is a variable over the sort `Int`, and L and L' are variables over the sort `List`.

Equational specifications have many possible interpretations or *models*, which are first-order structures M in which

- sorts s are interpreted as sets $s^{\mathcal{M}}$;
- constants c of the sort s are interpreted as elements $c^{\mathcal{M}}$ in the set $s^{\mathcal{M}}$; and
- operators f are interpreted as functions $f^{\mathcal{M}}$ over the elements in the sets interpreting the sorts of their arguments, and which return elements in the sets interpreting the sorts of their results.

While the different interpretations have in common that they satisfy the axioms included in the specification and their logical *consequences*, they are different because each one can satisfy other properties, which cannot be derived from those axioms using the inference rules of equational logic.

Example 3. All possible interpretations of the specification of lists of integers introduced in Example 2 satisfy that

$$\forall\{I, L\}(\texttt{append}(\texttt{cons}(I, \texttt{nil}), \texttt{nil}) = \texttt{cons}(I, \texttt{nil}))\,,$$

where I is a variable over the sort `Int`. However, some of them may satisfy that

$$\texttt{append}(\texttt{append}(L, L'), L'') = \texttt{append}(L, \texttt{append}(L', L''))\,, \tag{1}$$

that is, that the operation `append` is associative, while others may satisfy its negation

$$\neg(\texttt{append}(\texttt{append}(L, L'), L'') = \texttt{append}(L, \texttt{append}(L', L'')))\,,$$

where L, L' and L'' are variables over the sort `List`.

Verifying an equational specification of a data structure consists in proving that the required properties are satisfied at least by the "interesting" models. Consequently, in the formulation of a verification task, it has to be clearly stated which are those models that should satisfy the property.

Example 4. The *inductive* models of the specification introduced in Example 2 certainly verify (1). In these models the sets interpreting the sorts are inductively generated by the constructors. For example, $\mathcal{M}$ is an inductive model for this specification if and only if $\texttt{List}^{\mathcal{M}}$ (and, analogously, $\texttt{Int}^{\mathcal{M}}$) is defined as follows:

– $\texttt{nil}^{\mathcal{M}} \in \texttt{List}^{\mathcal{M}}$.

– $\texttt{cons}^{\mathcal{M}}(I, L) \in \texttt{List}^{\mathcal{M}}$ if $I \in \texttt{Int}^{\mathcal{M}}$ and $L \in \texttt{List}^{\mathcal{M}}$.

– Nothing else belongs to $\texttt{List}^{\mathcal{M}}$.

1.2 Membership Equational Logic

A distinguishing feature of the specification of *semantic* data structures (such as ordered lists, binary search trees, priority queues, and powerlists) from the specifications of other data structures (such as lists, sets, queues, trees, etc.) is that the constructors for semantic data structures are *partial*, that is, ground constructor-terms not necessarily represent legal data.

Example 5. Consider now an equational specification of *ordered* lists of integers. As in Example 1, we can use `cons` and `nil` as constructors to represent lists; notice, however, that

- `cons(1, nil)` is both a list and an ordered list; and
- `cons(2, cons(1, nil))` is a list, but not an ordered list.

That is, `cons` and `nil` are only partial for ordered lists.

Membership equational logic (MEL) is an expressive version of equational logic; a full account of its syntax and semantics can be found in [Meseguer, 1998, Bouhoula et al., 2000]. MEL has been designed to ease the task of specifying semantic data structures. To handle partiality, MEL introduces the distinction between *kinds* and *sorts*, which must be always associated to kinds. The idea is that kinds can include "illegal" terms, while sorts only include "legal" terms.

Example 6. Let us specify ordered lists of integers in MEL. We first declare a kind `Bool?` with an associated sort `Bool` (for legal boolean values), a kind `Int?` with an associated sort `Int` (for legal integers), and a kind `List?` with three associated sorts, `NeList` (for legal non-empty lists), `List` (for legal lists) and `OList` (for legal ordered lists). Then, we declare that `nil` is a constant symbol of the kind `List?`, and that `cons` is a binary operator symbol that takes terms of the kind `Int?` and terms of the kind `List?` and constructs terms of the kind `List?`. For the sake of the example, we suppose that `true` and `false` have been declared as constant symbols of the kind `Bool?`, and we also suppose that ...,`-1`,`0`,`1`,... have been declared as constant symbols of the kind `Int?`, along with the standard arithmetic operations and relations (which are represented as boolean functions). This signature can be declared in the *Diet Maude* specification language as follows:

```
kind Bool? = [Bool] .
kind Int? = [Int] .
kind List? = [NeList, List, OList] .
op nil : -> List? .
op cons : Int? List? -> List? .
```

Diet Maude is a syntactic *sugar-free* dialect of Maude [Clavel et al., 2005], a high-performance interpreter for MEL (and its rewriting logic extension). In Diet Maude, variables range over kinds (whose names must be included in the variables' names), and kinds must be explicitly declared with their associated sorts. In the above signature, `cons(1, nil)`, and `cons(2, cons(1, nil))` are well-formed terms since they have a kind, namely, `List?`. But they do not have a sort, since we have not defined yet which terms represent legal non-empty lists, lists, and/or ordered lists.

To define which well-formed terms have a sort (and not only a kind), MEL provides (conditional) *membership equational axioms*. A membership equational axiom is a universally quantified *membership assertion*,

$$\forall\{\mathbf{x}\}(t(\mathbf{x}) : s),$$

where $t(\mathbf{x})$ is a *pattern* (that is, a well-formed term with variables in $\mathbf{x}$), and s is a sort (which must be associated to the kind of the pattern). As an axiom, it declares that any well-formed term that is an instance of the pattern $t(\mathbf{x})$ *has the sort* s. Membership equational axioms can also be conditional,

$$\forall\{\mathbf{x}\}\Big(\Big(\bigwedge_i w_i(\mathbf{x}) : s_i \wedge \bigwedge_j u_j(\mathbf{x}) = v_j(\mathbf{x})\Big) \;\rightarrow t(\mathbf{x}) : s\Big),$$

where conditions are conjunctions of equalities and membership assertions. A membership assertion $w_i(\mathbf{x}) : s_i$ in the condition of a conditional membership axiom restricts the instances of $t(\mathbf{x})$ that have the sort s to be (a subset of) those for which the corresponding instances of $w_i(\mathbf{x})$ have the sort s_i. An equality $u_j(\mathbf{x}) = v_j(\mathbf{x})$ in a conditional membership axiom behaves like an equality in a conditional equation in equational logic.

Example 7. Consider the signature introduced in Example 6 to represent ordered lists of integers. We can define the terms that represent legal non-empty lists, legal lists, and legal ordered lists using the following membership axioms. Let I, I' be variables of the kind `Int?` and let L be a variable of the kind `List?`.

– A term of the form $\texttt{cons}(I, L)$ represents a legal non-empty list if I represents a legal integer and L represents a legal list:

$$\forall\{I, L\}((I : \texttt{Int} \wedge L : \texttt{List}) \rightarrow \texttt{cons}(I, L) : \texttt{NeList})\,.$$

– The constant `nil` represents a legal list.

$$\texttt{nil} : \texttt{List}\,.$$

– A term that represents a legal non-empty list represents a legal list as well:

$$\forall\{L\}(L : \texttt{NeList} \;\rightarrow\; L : \texttt{List})\,.$$

– The constant `nil` represents a legal ordered list.

$$\texttt{nil} : \texttt{OList}\,.$$

– A term of the form $\texttt{cons}(I, \texttt{nil})$ represents a legal ordered list if I represents a legal integer:

$$\forall\{I\}(I : \texttt{Int} \rightarrow \texttt{cons}(I, \texttt{nil}) : \texttt{OList})\,.$$

– A term of the form $\texttt{cons}(I, \texttt{cons}(I', L))$ represents a legal ordered list if I and I' represent legal integers, L represents a legal list, $\texttt{cons}(I', L)$ represents a legal ordered list, and $I \leq I'$:

$$\begin{aligned}&\forall\{I, I', L\}(\\ &\quad (I : \texttt{Int} \wedge I' : \texttt{Int} \wedge L : \texttt{List} \wedge \texttt{cons}(I', L) : \texttt{OList} \wedge I \leq I' = \texttt{true})\\ &\quad \rightarrow \texttt{cons}(I, \texttt{cons}(I', L)) : \texttt{OList})\,.\end{aligned}$$

The above membership axioms can be declared in Diet Maude as follows:

```
cmb cons(I:Int?, L:List?): NeList
    if I:Int? : Int /\ L:List? : List .

mb nil : List .
cmb L:List? : List if L:List? : NeList .

mb nil : OList .
cmb cons(I:Int?, nil): OList if I:Int? : Int .
cmb cons(I:Int?, cons(I':Int?, L:List?)): OList
    if I:Int? : Int /\ I':Int? : Int /\ L:List? : List /\
       cons(I':Int?, L:List?): OList /\ I:Int? <= I':Int? = true .
```

In this context, the term `cons(1, nil)` has sorts `NeList`, `List` and `OList`, while the term `cons(2, cons(1, nil))` has only sorts `NeList` and `List`. That is, the term `cons(1, nil)` represents both a (non-empty) legal list and a legal ordered list, while the term `cons(2, cons(1, nil))` represents a (non-empty) legal list but not a legal ordered list.

To define the value of the operations over the elements of the data type, MEL provides (conditional) *equational axioms.* An equational axiom is a universally quantified equality,

$$\forall\{\mathbf{x}\}(t(\mathbf{x}) = t'(\mathbf{x}))\,,$$

where $t(\mathbf{x})$ and $t'(\mathbf{x})$ are patterns (which must have the same kind): $t(\mathbf{x})$ is called the left-hand side, which typically consists of a term representing an operation call, and $t'(\mathbf{x})$ the right-hand side of the equation. As an axiom, it declares that any well-formed term that is an instance of the left-hand side *is equal to* the corresponding instance of the right-hand side. Equational axioms can also be conditional,

$$\forall\{\mathbf{x}\}\Big(\Big(\bigwedge_i w_i(\mathbf{x}) : s_i \wedge \bigwedge_j u_j(\mathbf{x}) = v_j(\mathbf{x})\Big) \rightarrow t(\mathbf{x}) = t'(\mathbf{x})\Big)\,,$$

where conditions are conjunctions of equalities and membership assertions.

Example 8. Consider the operation that returns the tail of a list of integers. We first extend the signature introduced in Example 7 with a unary operation symbol `tail` that takes terms of the kind `List?` and returns terms also of the kind `List?`. Then, we define that

– the `tail` of a list constructed by `cons`-ing an integer to a list is equal to this list:
$$\forall\{I, L\}(\mathtt{tail}(\mathtt{cons}(I, L)) = L)\,,$$
where I is a variable of the kind `Int?` and L is a variable of the kind `List?`.

This equational axiom can be declared in Diet Maude as follows:

```
eq tail(cons(I:Int?, L:List?)) = L:List? .
```

Finally, MEL specifications have many possible interpretations or models, which are first-order structures M in which

– kinds k are interpreted as sets $k^{\mathcal{M}}$;

– sorts s of the kind k are interpreted as subsets $s^{\mathcal{M}} \subseteq k^{\mathcal{M}}$;

– constants c of the kind k are interpreted as elements $c^{\mathcal{M}}$ in the set $k^{\mathcal{M}}$; and

– operators f are interpreted as functions $f^{\mathcal{M}}$ over the elements in the sets interpreting the kinds of their arguments, and which return elements in the sets interpreting the kinds of their results.

– sort-membership is interpreted as membership in the set interpreting the sort.

The different interpretations have in common that they satisfy the axioms included in the specification and their logical *consequences*. To prove that a property is a logical consequence we can use the MEL inference system [Meseguer, 1998]. This is a sound and complete calculus that extends equational logic with rules for proving memberships. In particular, to prove that a term t has a sort s we can

– check that t is an instance of (the pattern of) a membership axiom and, in the case of a conditional membership, prove that the conditions are fulfilled, or

– prove that t is equal to another term t' and that t' has the sort s.

Example 9. Consider the specification of ordered lists with the `tail` operation introduced in Example 8. We can prove that

$$\forall\{I, L\}((I : \mathtt{Int} \wedge L : \mathtt{List}) \;\rightarrow\; \mathtt{tail}(\mathtt{cons}(I, L)) : \mathtt{List}) \qquad (2)$$

since, for any terms I and L, $\texttt{tail}(\texttt{cons}(I, L))$ is equal to L and L is, by assumption, of the sort `List`. Thus, by soundness of the calculus, (2) is a logical consequence and it holds in all the models of the specification.

1.3 The ITP Tool

The ITP tool is a theorem-proving *assistant*. It can be used to interactively verify properties of MEL specifications with respect to its *ITP-models*. An ITP-model is a model of the specification such that:

– it is an inductive model, in the sense that the sets interpreting the sorts in the model are inductively generated by the membership axioms defining the sorts in the specification;

– it is a standard model of the theory of arithmetic for the integer numbers.

Example 10. Let $\mathcal{M}$ be a model of the specification of ordered lists introduced in Example 7. Let us suppose that $\mathcal{M}$ is also a standard model for the theory of arithmetic, with $\texttt{Int}^{\mathcal{M}}$ being the set of integer numbers $\mathbb{Z}$, and $\texttt{<=}^{\mathcal{M}}$ the inequality relation $\leq$. Then, $\texttt{NeList}^{\mathcal{M}}$ is inductively generated by the membership axioms defining `NeList` if and only if $\texttt{NeList}^{\mathcal{M}}$ is defined as follows:

– $\texttt{cons}^{\mathcal{M}}(I, L) \in \texttt{NeList}^{\mathcal{M}}$ if $I \in \mathbb{Z}$ and $L \in \texttt{List}^{\mathcal{M}}$.

– Nothing else belongs to $\texttt{NeList}^{\mathcal{M}}$.

Similarly, $\texttt{List}^{\mathcal{M}}$ is inductively generated if and only if:

– $\texttt{nil}^{\mathcal{M}} \in \texttt{List}^{\mathcal{M}}$.

– $L \in \texttt{List}^{\mathcal{M}}$ if $L \in \texttt{NeList}^{\mathcal{M}}$.

– Nothing else belongs to $\texttt{List}^{\mathcal{M}}$.

Finally, $\texttt{OList}^{\mathcal{M}}$ is inductively generated if and only if:

– $\texttt{nil}^{\mathcal{M}} \in \texttt{OList}^{\mathcal{M}}$.

– $\texttt{cons}^{\mathcal{M}}(I, \texttt{nil}^{\mathcal{M}}) \in \texttt{OList}^{\mathcal{M}}$ if $I \in \mathbb{Z}$.

– $\texttt{cons}^{\mathcal{M}}(I, \texttt{cons}^{\mathcal{M}}(I', L)) \in \texttt{OList}^{\mathcal{M}}$ if $I, I' \in \mathbb{Z}$, $L \in \texttt{List}^{\mathcal{M}}$, $\texttt{cons}^{\mathcal{M}}(I', L) \in \texttt{OList}^{\mathcal{M}}$, and $I \leq I'$.

– Nothing else belongs to $\texttt{OList}^{\mathcal{M}}$.

By default, the ITP tool assumes that the sets interpreting the sorts are also *freely* generated [Enderton, 2000], that is, two different ground constructor-terms always denote different elements. However, users can "customize" the tool for verifying properties about models that do not satisfy the "freeness" requirement; of course, some of the commands, whose soundness is based on this requirement, would not be then available.

An important feature of the proposed semantic framework is that it supports proofs by structural induction and complete induction. Another interesting feature is that incompletely specified operations can be reasoned about so as to support incrementality. That is, unlike in most reasoning systems including RRL [Kapur and Zhang, 1995] and ACL2 [Kaufmann et al., 2000], operations do not have to be completely specified before inductive properties about them can be verified mechanically.

Example 11. Consider an extension of the specification of ordered lists introduced in Example 7 that includes an operation `length` to calculate the length of a list. For the sake of the example, suppose that `length` is incompletely specified.

```
op length : List? -> Int? .
eq length(cons(I:Int?, L:List?)) = 1 + length(L:List?) .
```

Despite the fact that the length of an empty list is not defined, the following property can be verified:

$$\begin{aligned}(\forall\{L\}(L:\texttt{List} \ \rightarrow\ \texttt{length}(L):\texttt{Int}) \ \wedge\ \texttt{0} \mathrel{\texttt{<=}} \texttt{length}(\texttt{nil}) = \texttt{true})\\ \rightarrow\ \forall\{L\}(L:\texttt{List} \ \rightarrow\ \texttt{length}(L) \mathrel{\texttt{>=}} \texttt{0} = \texttt{true})\,,\end{aligned} \tag{3}$$

where L is a variable of the kind `List?`. The proof goes as follows. Assume that

$$\forall\{L\}(L:\texttt{List} \ \rightarrow\ \texttt{length}(L):\texttt{Int}) \tag{4}$$

and that

$$\texttt{0} \mathrel{\texttt{<=}} \texttt{length}(\texttt{nil}) = \texttt{true}\,. \tag{5}$$

Then, by induction on L,

$$\forall\{L\}(L:\texttt{List} \ \rightarrow \texttt{length}(L) \mathrel{\texttt{>=}} \texttt{0} = \texttt{true})$$

holds if

$$\texttt{length}(\texttt{nil}) \mathrel{\texttt{>=}} \texttt{0} = \texttt{true} \tag{6}$$

holds and, for any term L in the kind `List?`,

$$\forall\{I\}(I:\texttt{Int} \wedge L:\texttt{List} \ \rightarrow \texttt{length}(\texttt{cons}(I,L)) \mathrel{\texttt{>=}} \texttt{0} = \texttt{true}) \tag{7}$$

holds whenever

$$\texttt{length}(L) \mathrel{\texttt{>=}} \texttt{0} = \texttt{true} \tag{8}$$

does. The base case (6) is a consequence, in the theory of arithmetic, of (4) and (5). The inductive case (7) can be reduced to proving that

$$\texttt{1} \mathbin{\texttt{+}} \texttt{length}(L) \mathbin{\texttt{>=}} \texttt{0} = \texttt{true} \tag{9}$$

holds since, for any terms I and L of the kinds, respectively, `Int?` and `List?`,

$$(\texttt{length}(\texttt{cons}(I, L)) \mathbin{\texttt{>=}} \texttt{0}) = (\texttt{1} \mathbin{\texttt{+}} \texttt{length}(L) \mathbin{\texttt{>=}} \texttt{0})$$

is a consequence of the axiom defining `length`. But, (9) is a consequence, in the theory of arithmetic, of (4) and (8).

Finally, notice that the class of the ITP-models includes the *initial* model of the specification,[1] but possibly also many other models. This "tolerance" provides the extra freedom that is needed to inductively reason about incompletely specified operations.

Example 12. The initial model of the specification of ordered lists introduced in Example 11 also satisfies property (3), simply because

$$\texttt{0} \mathbin{\texttt{<=}} \texttt{length}(\texttt{nil}) = \texttt{true}$$

does not hold in the initial model. In fact, it also satisfies

$$\begin{aligned}(\forall\{L\}(L : \texttt{List} \;\rightarrow\; \texttt{length}(L) : \texttt{Int}) \;\wedge\; \texttt{0} \mathbin{\texttt{>}} \texttt{length}(\texttt{nil}) = \texttt{true})&\\ \rightarrow\; \forall\{L\}(L : \texttt{List} \;\rightarrow\; \texttt{length}(L) \mathbin{\texttt{>=}} \texttt{0} = \texttt{true})&\end{aligned} \tag{10}$$

simply because, again,

$$\texttt{0} \mathbin{\texttt{>}} \texttt{length}(\texttt{nil}) = \texttt{true} \tag{11}$$

does not hold in the initial model. However, (10) cannot be verified, as expected, in the class of the ITP-models since, for those in which (11) holds, then

$$\texttt{length}(\texttt{nil}) \mathbin{\texttt{>=}} \texttt{0} = \texttt{true}$$

cannot be true.

2 Getting started

The ITP tool is a Maude program. It comprises over 8.000 lines of Maude code that make extensive use of the reflective capabilities of the system. In fact, rewriting-based proof simplification steps are directly executed by the powerful

[1] In the initial model [Meseguer, 1998], sorts are interpreted as the smallest sets satisfying the axioms in the theory, and equality is interpreted as the smallest congruence satisfying those axioms.

underlying Maude rewriting engine, by transforming the Diet Maude specifications into Maude admissible modules.

The ITP tool is currently available as a web-based application that includes a *module editor*, a *formula editor*, and a *command editor*. These editors allow users to create and modify their specifications, to formalize properties about them, and to guide the proofs by filling and submitting web forms. The web application also offers a goal viewer, a script viewer, and a log viewer. They generate web pages that allow the user to check, print, and save the current state of a proof, the commands that have guided it, and the logs generated in the process by the Maude system.

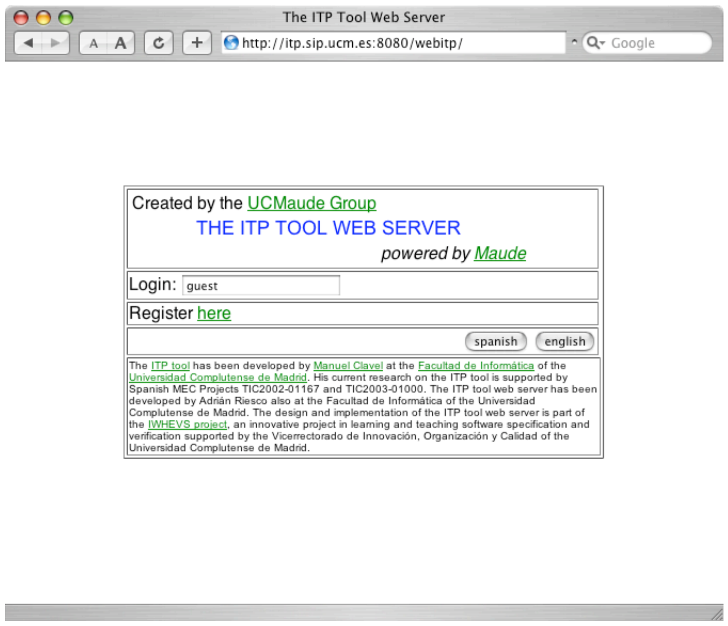
The ITP Web tool can be executed in two different ways: either as a remote or as a local application. It comprises 2.000 lines of Maude code, 3.000 lines of JSP, and 7.500 lines of Java. The only requirements to run the remote application are a computer with an Internet connection and JDK 1.4.1 installed (it should be available on most computers), and a browser. The remote application can be accessed at the URL `http://itp.sip.ucm.es:8080/webitp/`, which displays the initial window shown in Figure 1. Running the ITP tool as a local application is more demanding. In addition to JDK 1.4.1 and a browser, it is also necessary to have the Tomcat server installed, as well as Maude and the files containing the specification of the ITP. The concrete details of the files needed and the steps to follow to complete the installation can be found at `http://maude.sip.ucm.es/itp/`.

2.1 Introducing a module

The screen captured in Figure 2 shows the ITP Web main menu: from it, Diet Maude modules can be created (create button) and edited (open button), and ITP scripts can be loaded (load). To give a taste of the tool, let us create the module corresponding to the specification of lists of integers introduced in Example 2 which contained an operation `append` to concatenate two lists.

To create a module from scratch, we push the create button, which takes us to the module editor whose appearance is shown in Figure 3. A new module `tmp-id` is opened which, by default, contains kinds for quoted identifiers, booleans, integers (with the standard operations), and strings. The kind `Int?` contains the sorts `Zero` (for the number 0), `NzNat` (for natural numbers different from 0), `Nat` (for natural numbers), `NzInt` (for integer numbers different from 0), and `Int` (for integer numbers). There are three main actions that can be executed from the module editor: modifying (modify column), inserting (insert column), and deleting (delete column) an element from the module.

To insert a kind, sort, operator, membership, or equation, we have to select the corresponding option from the drop-down list below insert and then push select. The next step depends on the selection. For inserting a kind, we only need



Created by the UCMaude Group

THE ITP TOOL WEB SERVER

powered by Maude

Login: guest

Register here

The ITP tool has been developed by Manuel Clavel at the Facultad de Informática of the Universidad Complutense de Madrid. His current research on the ITP tool is supported by Spanish MEC Projects TIC2002-01167 and TIC2003-01000. The ITP tool web server has been developed by Adrián Riesco also at the Facultad de Informática of the Universidad Complutense de Madrid. The design and implementation of the ITP tool web server is part of the IWHEVS project, an innovative project in learning and teaching software specification and verification supported by the Vicerrectorado de Innovación, Organización y Calidad of the Universidad Complutense de Madrid.

Figure 1: Initial screen of the ITP Web

to write its name; for inserting a sort, in addition to its name, the kind to which it belongs has to be specified: this is done by selecting one of the currently declared kinds from a drop-down list which is dynamically updated. The insertion of a new operator is guided by the menu in Figure 4. The operator's name is written below operator. Then, the types of the arguments are selected from a drop-down list which contains the available kinds, and inserted (or removed) by pressing add (delete). Finally, the type of the result is specified in the same manner using the list below result. There is also a check box ctor that allows the operator to be flagged as a free constructor. The procedure for adding a membership or equation is similar in both cases: a term (or two, for an equation) has to be written and a sort selected from those available. Conditions, if required, are added in a similar way. Proceeding in this manner we introduce the whole specification `LIST`, as shown in Figure 5. Once completed, we can press save to save the resulting module in a file.[2] We can also press next to load the module in the ITP database and return to the ITP Web main menu.

[2] In fact, save generates a web page with the ITP commands that have created the module. To really *save* the module, save the content of this web page as a file.

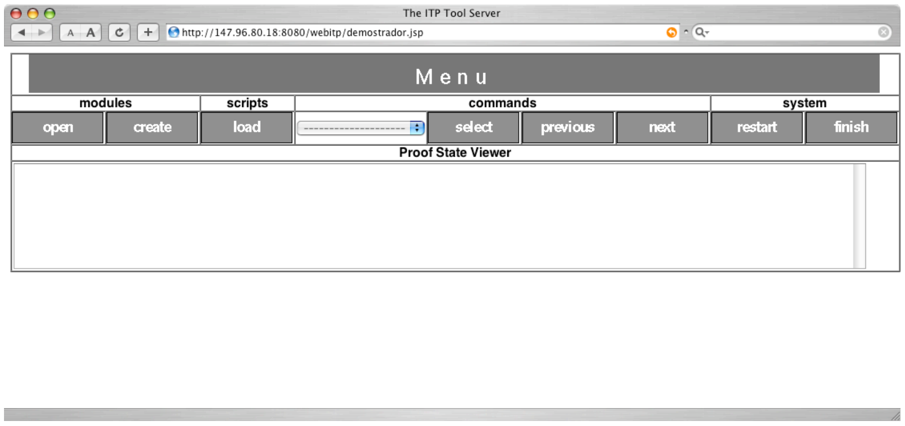
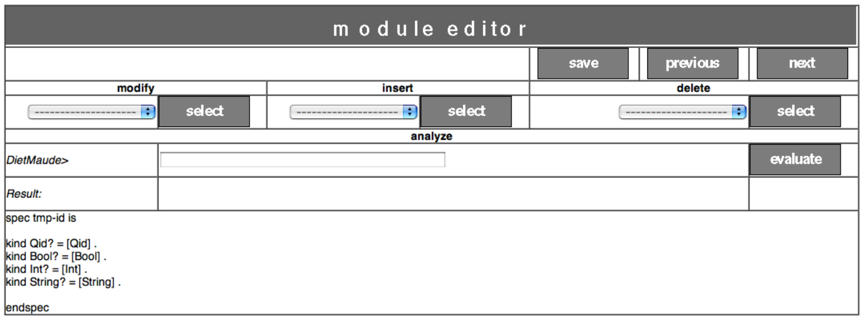
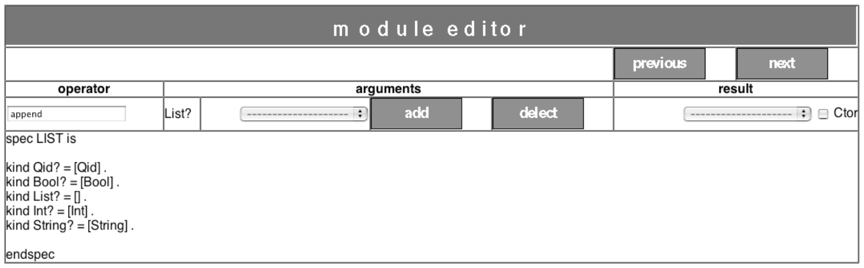


Figure 2: Main menu



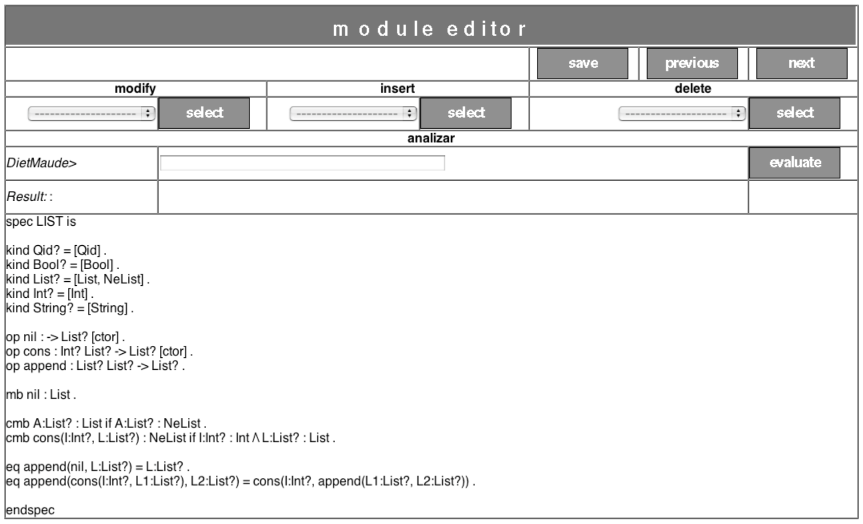
```
spec tmp-id is

kind Qid? = [Qid] .
kind Bool? = [Bool] .
kind Int? = [Int] .
kind String? = [String] .

endspec
```

Figure 3: Module editor



```
spec LIST is

kind Qid? = [Qid] .
kind Bool? = [Bool] .
kind List? = [] .
kind Int? = [Int] .
kind String? = [String] .

endspec
```

Figure 4: Inserting an operator



```
spec LIST is

kind Qid? = [Qid] .
kind Bool? = [Bool] .
kind List? = [List, NeList] .
kind Int? = [Int] .
kind String? = [String] .

op nil : -> List? [ctor] .
op cons : Int? List? -> List? [ctor] .
op append : List? List? -> List? .

mb nil : List .

cmb A:List? : List if A:List? : NeList .
cmb cons(I:Int?, L:List?) : NeList if I:Int? : Int ∧ L:List? : List .

eq append(nil, L:List?) = L:List? .
eq append(cons(I:Int?, L1:List?), L2:List?) = cons(I:Int?, append(L1:List?, L2:List?)) .

endspec
```

Figure 5: Specification of lists of integers

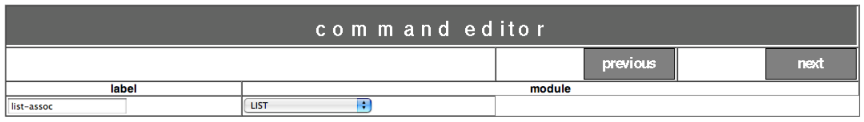
2.2 A first proof

A property that `LIST` should verify is that `append` is associative:

$$\forall\{L, L'L''\}(L : \texttt{List} \wedge L' : \texttt{List} \wedge L'' : \texttt{List}) \\ \rightarrow \texttt{append}(\texttt{append}(L, L'), L'') = \texttt{append}(L, \texttt{append}(L', L''))\,, \quad (12)$$

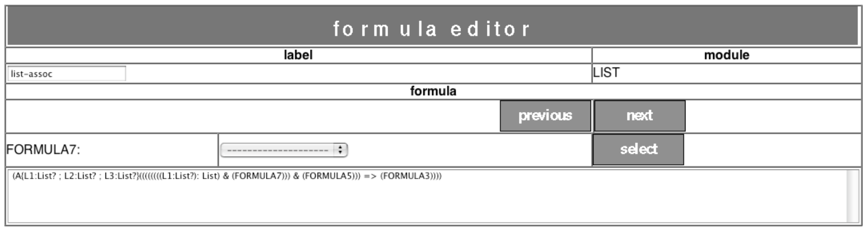
where L, L', and L'' are variables of the kind `List?`. This goal can be introduced by choosing the command goal from the drop-down list beside select, in the main menu; initially, this is the only option available, but once a goal has been introduced other commands will appear in the list. The first step consists in choosing the goal's label and the module is going to refer to. In this case, we choose the name `list-assoc` and the module `LIST`, as shown in Figure 6. The second step consists in introducing the formula to be verified. For this, the tool takes us to the formula editor. Formulas are constructed in a top-down, left-to-right fashion, as illustrated by the partially built formula in Figure 7. The formula editor contains a drop-down list from which the (sub)formula's top structure is chosen: this can be an equality (`=`), membership (`:`), implication (`=>`), conjunction (`&`), or existential (`E`) or universal quantification (`A`). Once completed, we can press next to return to the main menu: the goal's label will appear in the proof state viewer, as shown in Figure 8.

We can try to prove (12) by *structural induction*. For this, we select the sort-ind command, that has now been included, along several other options, in



list-assoc

Figure 6: Naming a goal



list-assoc LIST

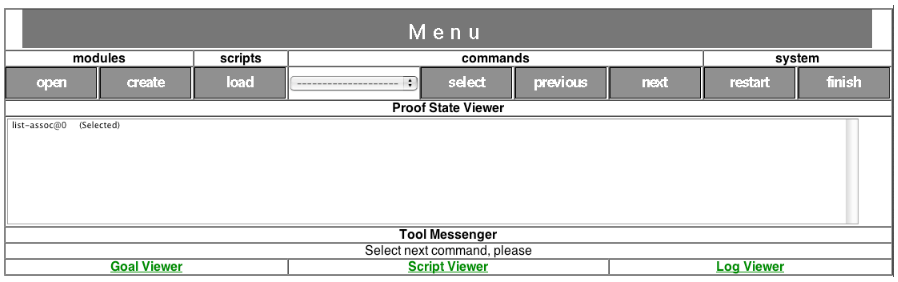
FORMULA7:

```
(A{L1:List? ; L2:List? ; L3:List?}((((((((L1:List?): List) & (FORMULA7))) & (FORMULA5))) => (FORMULA3))))
```

Figure 7: Building a formula

the drop-down list beside select. The command editor then asks to select from a drop-down list of variables which one is to be used for the induction (`L:List?` in this case); notice that the list only includes those variables for which the goal hypothesis contains a membership assertion. The command generates then an inductive subgoal for each of the memberships that specify the sort of the variable, and it chooses one of them as the working subgoal and tags it with `(Selected)`. (Actually, the working subgoal can be changed using the command sel.) The output is as in Figure 9.

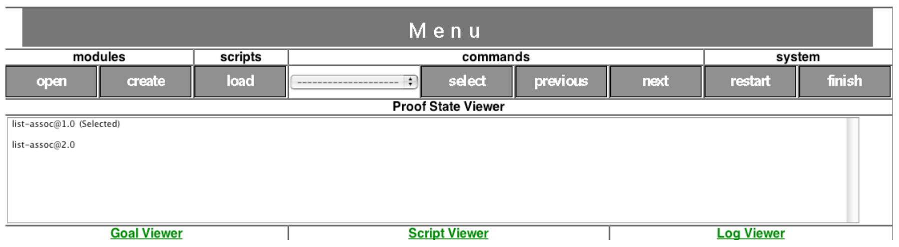
We can use the goal viewer to obtain information about the selected subgoal: its label, the property to be proved, the hypotheses and lemmas available (if



```
list-assoc@0    (Selected)
```

Select next command, please

Figure 8: A goal just introduced



```
list-assoc@1.0 (Selected)
list-assoc@2.0
```

Figure 9: Subgoals

any), the "fresh" constants introduced (if any), and the original module. In this case, the goal viewer shows us the following information:

```
Label: append-assoc@1.0
Formula:
 A{L':List? ; L'':List?}(
   ((L':List? : List)&(L'':List? : List))
    ==>(append(append(nil,L':List?),L'':List?)
        = append(nil,append(L':List?,L'':List?)))).
```

Notice that this is the subgoal corresponding to the base case in the inductive proof: the module `LIST` should verify (12) when L is `nil`. At this point, we can try to automatically prove `append-assoc@1.0` with the command auto, that simplifies the goal, using the axioms in the original module along with the available hypotheses and lemmas, and discharges it when it discovers an inconsistency or reaches an identity. The command succeeds, the subgoal is discharged, and the ITP presents us with the remaining subgoal generated by the induction. The goal viewer presents us the following information:

```
Label: append-assoc@2.0
Formula:
 A{V0#0:Int? ; V0#1:List?}(
    (((V0#1:List? : List))&(V0#0:Int? : Int) &
      (A{L':List? ; L'':List?}(
          ((L':List? : List)&(L'':List? : List))
           ==>(append(append(V0#1:List?,L':List?),L'':List?)
               = append(V0#1:List?,
                        append(L':List?,L'':List?))))))
   ==>
    (A{L':List? ; L'':List?}(
        ((L':List? : List)&(L'':List?: List))
          ==>(append(append(cons(V0#0:Int?,V0#1:List?),L':List?),
                  L'':List?)
          = append(cons(V0#0:Int?,V0#1:List?),
                      append(L':List?,L'':List?)))))).
```

This is the subgoal corresponding to the inductive case: the module LIST should verify (12) when L is cons(V0#0:Int?,V0#1:List?), assuming that it verifies (12) when L is V0#1:List?. We can also try to prove this subgoal automatically with the auto command and, again, the ITP succeeds and this completes the proof. The script viewer shows the script of our proof, which can be saved in a file for later use.

3 A script safari

We continue now exploring the ITP commands and illustrate how they are used in several scripts.

3.1 More on induction and simplification

In addition to the command sort-ind to reason by structural induction, the ITP also provides a nat-ind command to reason by *induction over the natural numbers*. This command takes a term of the kind Int? as argument and generates three subgoals from the original goal (the first one added as a lemma in the second and third): one which requires to prove that the term is of sort Nat; one which states that the goal holds for the term being equal to 0; and another one which states that the goal holds for the term being equal to n, assuming that it holds for the term being less than n. To illustrate its use, let us consider a module LIST-LENGTH that extends LIST with an operation to determine the length of a list.

```
op length : List? -> Int? .
eq length(nil) = 0 .
eq length(I:Int? : L:List?) = 1 + length(L:List?) .
```

In the module LIST-LENGTH, the associativity of the operation append can alternatively be proved by selecting nat-ind, that asks to introduce the term to use for the induction (in this case, length(L1:List?)), as shown in Figure 10. This selection gives rise to three subgoals. The first one, as mentioned above, simply requires to prove that the term length(L1:List?) is of sort Nat. The goal viewer now shows the following information:

```
Label: append-assoc-bis@0@1.0
Formula: A{L:List?}((L:List? : List)==>(length(L:List?): Nat)).
```

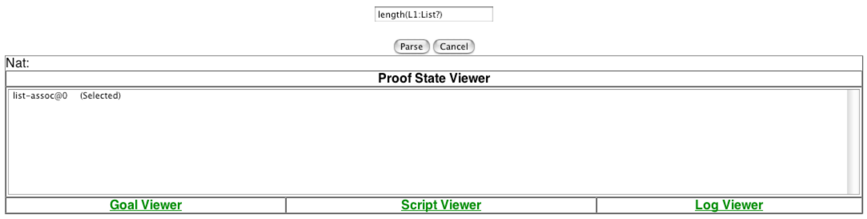


Figure 10: Induction over natural numbers

This subgoal is discharged straightforwardly by one application of the induction scheme over lists (sort-ind) and two uses of the auto command. The goal viewer shows the following information about the first (and selected) of the remaining subgoals, which states that (12) holds for any terms L, L', L'' of the sort `List` such that `length`(L) is equal to 0.

```
Label: append-assoc-bis@1.0
Formula:
A{L:List? ; L':List? ; L'':List?}(
  (((L:List? : List))&(L':List? : List)&(L'':List? : List)
     & (length(L:List?)= 0))
   ==>(append(append(L:List?,L':List?), L'':List?)
       = append(L:List?,append(L':List?,L'':List?)))).
Lemmas:
cmb length(L:List?): Nat if L:List? : List[lem-append-assoc-bis@0].
```

The lemma `lem-append-assoc-bis@0` corresponds to `append-assoc-bis@0@1.0`, already proved. In the case of `append-assoc-bis@1.0`, using auto seems to have no effect except for the transformation of the variables into "fresh" constants and the simplication of the goal, by adding the formulas in the antecedent as hypotheses, as shown by the goal viewer:

```
Label: append-assoc-bis@1.0
Formula: append(append(L*List?,L'*List?),L''*List?)
          = append(L*List?,append(L'*List?, L''*List?)).
Hypotheses:
mb L''*List? : List[hyp-0].
mb L'*List? : List[hyp-1].
mb L*List? : List[hyp-2].
eq length(L*List?)= 0[hyp-3].
Lemmas:
cmb length(L:List?): Nat if L:List? : List[lem-append-assoc@0].
```

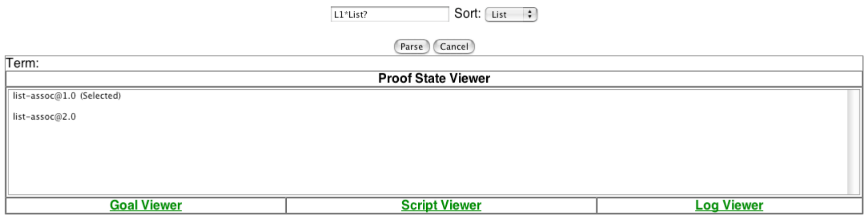


Figure 11: Term splitting

```
New constants:
op L''*List? : -> List? .
op L'*List? : -> List? .
op L*List? : -> List? .
```

Since the additional information provided by the hypotheses is not enough to further reduce any of the terms in the simplified goal, to proceed with the proof we must supply the ITP with guidelines to follow. In this case, it is useful to make a *case analysis on the structure* of `L*List`. For this, we can use the command `term-split` which asks for a term to perform the splitting on as well as for its sort, as shown in Figure 11. This command replaces the subgoal `append-assoc-bis@1.0` with the two following ones, corresponding to the constructors for `List`:

```
Label: append-assoc-bis@1.1.0
Formula: (L*List? = nil)
            ==>(append(append(L*List?,L'*List?),L''*List?)
                = append(L*List?,append(L'*List?,L''*List?))).

Label: append-assoc-bis@1.2.0
Formula: (((L*List? = cons(V1#0*Int?,V1#1*List?))
           &(V1#1*List? : List))&(V1#0*Int? : Int))
            ==>(append(append(L*List?,L'*List?),L''*List?)
                = append(L*List?,append(L'*List?,L''*List?))).
```

At this point, `append-assoc-bis@1.1.0` and `append-assoc-bis@1.2.0` have the same hypotheses, lemmas, and new constants as `append-assoc-bis@1.0`.

Now we can try to prove the subgoal `append-assoc-bis@1.1.0`, corresponding to the case when the list `L*List?` is `nil`, and `auto` easily succeeds. It adds the equality in the antecedent as a new equational hypothesis; then, it simplifies both sides of the equality in the consequent using the axioms, hypotheses, and lemmas as rewrite rules; and finally, it discharges the simplified equality, because two terms syntactically identical are trivially equal.

The other subgoal generated deserves closer attention. The list `L*List?` is now built using `cons` 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 subgoal `append-assoc-bis@1.0`, that is, the one in which `length(L*List?)` is equal to `0` (as stated in its hypothesis `[hyp-3]`), and this is inconsistent with `L*List?` being constructed with `cons`. In this case, the `auto` command adds the equalities and memberships in the antecedent as new hypotheses:

```
eq L*List? = cons(V1#0*Int?,V1#1*List?) .
mb V1#1*List? : List .
mb V1#0*Int? : Int .
```

and it simplifies the hypothesis `[hyp-3]` to

```
eq 1 + length(V1#1*List?)= 0 .
```

Then it realizes, using the internal decision procedure for linear arithmetic with uninterpreted function symbols [Clavel et al., 2004], that the term `0` cannot be equal to `1 + length(V1#1*List?)`, since:

– the lemma `[lem-append-assoc-bis@0]` guarantees that, for any term L of sort `List`, `length(`L`)` returns a term greater or equal than `0`, and,

– by hypothesis, `V1#1*List?` is a term of the sort `List`;

and it discharges the subgoal, not because both sides of the equality in the consequent can be reduced to a common term, but because the axioms, hypotheses, and lemmas form an inconsistent specification.

We are now left with the proof of the third subgoal generated by nat-ind, labeled `append-assoc-bis@2.0`, which states that (12) holds for any terms L, L', L'' such that `length(`L`)` is equal to n, assuming that it holds for any terms L_1, L_1', L_1'' such that `length(`L_1`)` is less than n.

```
Label: append-assoc-bis@2.0
Formula:
A{V0#0:Nat}(
  (A{L:List? ; L':List? ; L'':List?}(
      ((((length(L:List?)< V0#0:Nat = true)
         &(L:List? : List))&(L':List? : List))&(L'':List? : List))
        ==>(append(append(L:List?,L':List?),L'':List?)
            = append(L:List?,append(L':List?, L'':List?)))))
  ==>
```

```
 (A{L:List? ; L':List? ; L'':List?}(
     ((((length(L:List?)= V0#0:Nat)
        &(L:List? : List))&(L':List? : List))&(L'':List? : List))
        ==>(append(append(L:List?,L':List?),L'':List?)
           = append(L:List?,append(L':List?, L'':List?)))))).
```

The proof proceeds exactly like that for `append-assoc-bis@1.0`. 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.)

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

3.2 Lemmas and more

This example is more involved than the previous ones and, beside to further illustrate the use of the sort definitions, it will serve to introduce two additional ITP commands: lem, to introduce auxiliary lemmas, and bool-split for boolean case analysis.

Consider an extension `LIST-SORT` of the specification of ordered lists introduced in Example 7 that includes an operation `insertion-sort` to order a non-empty list by inserting, at the right position, its first element into the list that results from (recursively) ordering its tail.

```
  op insert : Int? List? -> List? .
  op insertion-sort : List? -> List? .

  eq insertion-sort(nil) = nil .
  eq insertion-sort(I:Int? : L:List?)
     = insert(I:Int?, insertion-sort(L:List?)) .

  eq insert(I:Int?, nil) = I:Int? : nil .
  ceq insert(I:Int?, J:Int? : L:List?)
      = I:Int? : J:Int? : L:List?
      if I:Int? <= J:Int? = true .
  ceq insert(I:Int?, J:Int? : L:List?)
      = J:Int? : insert(I:Int?, L:List?)
      if I:Int? > J:Int? = true .
```

A property that `LIST-SORT` should verify is that `insert-sort` returns an ordered list:[3]

$$\forall\{L\}(L : \texttt{List} \ \rightarrow \texttt{insert-sort}(L) : \texttt{OList})\,, \tag{13}$$

where L is a variable of the kind `List?`.

3.2.1 Goals which are lemmas

To start with, however, we will settle for a simpler result, namely that inserting an element into an ordered list results in another ordered list:

$$\forall\{I, L\}(I : \texttt{Int} \wedge L : \texttt{OList} \ \rightarrow \texttt{insert}(I, L) : \texttt{OList})\,, \tag{14}$$

where I is a variable of the kind `Int?` and L is a variable of the kind `List?`. This property is likely to be needed as a lemma in order to prove (13). We can try to prove (14) by structural induction on `L:List?` with the sort-ind command. The case generated for the empty list is trivial and is discharged by auto; however, in the case of the singleton list, it merely transforms the goal into

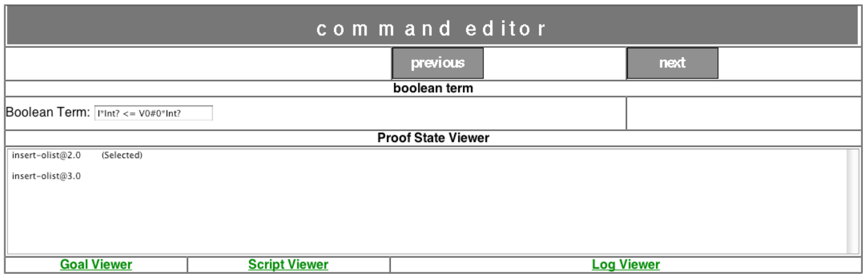
```
Label: insert-olist@2.0
Formula: insert(I*Int?,cons(V0#0*Int?,nil)): OList.

Hypotheses:
mb I*Int? : Int[hyp-1].
mb V0#0*Int? : Int[hyp-0].

New constants:
op I*Int? : -> Int? .
op V0#0*Int? : -> Int? .
```

If we take a look at the equations for `insert`, the reason why the term

```
insert(I*Int?,cons(V0#0*Int?,nil))
```

cannot be further reduced becomes apparent: the two equations that might apply are conditional and depend on whether `I*Int? <= V0#0*Int?` or `I*Int? > V0#0*Int?` is equal to `true`. Thus, to discharge the subgoal `insert-olist@2.0` it is necessary to *reason by cases*. For that the ITP Web tool offers the command bool-split, which asks for a boolean term, as shown in Figure 12, and replaces the subgoal with two new ones. As it can be checked with the goal viewer, in both subgoals the formulas to be proved are the same; the difference lies in

[3] Notice that 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.



Boolean Term: I*Int? <= V0#0*Int?

```
insert-olist@2.0      (Selected)

insert-olist@3.0
```

Figure 12: Boolean split

the hypotheses associated to them: for the first one, the equation `I*Int? <= V0#0*Int? = true` has been added while in the second the inequality `I*Int? <= V0#0*Int?` is equal to `false`. Now, both subgoals can be automatically discharged with auto.

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

```
Label: insert-olist@3.0
Formula:
insert(I*Int?,cons(V0#1*Int?,cons(V0#2*Int?,V0#0*List?))): OList.

Hypotheses:
mb I*Int? : Int[hyp-6].
mb V0#0*List? : List[hyp-0].
mb V0#1*Int? : Int[hyp-1].
mb V0#2*Int? : Int[hyp-2].
mb cons(V0#2*Int?,V0#0*List?): OList[hyp-3].
cmb insert(I:Int?,cons(V0#2*Int?,V0#0*List?)): OList
    if I:Int? : Int[hyp-5].
eq V0#1*Int? <= V0#2*Int? = true[hyp-4].

New constants:
op I*Int? : -> Int? .
op V0#0*List? : -> List? .
op V0#1*Int? : -> Int? .
op V0#2*Int? : -> Int? .
```

Again, we need to reason by cases and apply bool-split over `I*Int? <= V0#1*Int?`. This time, however, only the first auto succeeds while the second does not discharge the goal and presents us with:

```
Label: insert-olist@3.2.0
Formula:
cons(V0#1*Int?,insert(I*Int?,cons(V0#2*Int?,V0#0*List?))): OList.
```

along with the hypotheses and lemmas inherited from `insert-olist@3.0`, plus the hypothesis corresponding to this case:

```
eq I*Int? <= V0#1*Int? = false[case--false].
```

Assuming that `insert` is well-defined, it is clear that

```
    cons(V0#1*Int?,insert(I*Int?,cons(V0#2*Int?,V0#0*List?)))
```

is an ordered list, given that:

- `V0#1*Int?` is less than `I*Int?`, by hypothesis `[case--false]`; and
- `cons(V0#1*Int?, cons(V0#2*Int?,V0#0*List?))` is an ordered list, since, by hypothesis `[hyp-4]`, `V0#1*Int?` is less than or equal to `V0#2*Int?` and, by hypothesis `[hyp-3]`, `cons(V0#2*Int?,V0#0*List?)` is an ordered list.

What we need is a lemma that makes this observation a general and explicit statement about `insert`:

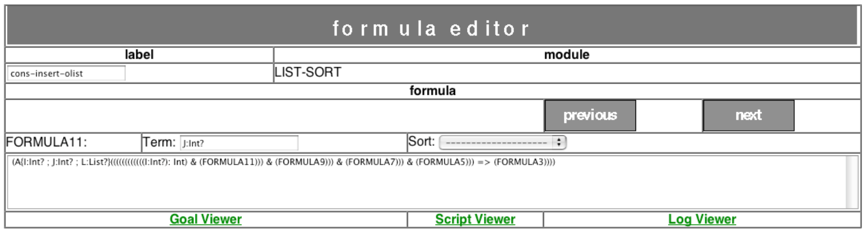
$$\begin{aligned}\forall\{I,J,L\}(((I : \mathtt{Int} \wedge J : \mathtt{Int} \wedge L : \mathtt{List} \\ \wedge\ I \mathbin{\texttt{<=}} J = \mathtt{true} \wedge \mathtt{cons}(I,L) : \mathtt{OList}) \\ \rightarrow \mathtt{cons}(I,\mathtt{insert}(J,L)) : \mathtt{OList}),\end{aligned} \tag{15}$$

where I and J are variables of the kind `Int?`, and L is a variable of the kind `List?`.

3.2.2 Lemmas which are goals

To introduce a lemma, we choose lem in the drop-down list beside select and this takes us to a formula editor like that for goals; see Figure 13. Back to the main menu, after introducing (15), the proof state viewer shows that (15) has been added as a new (and selected) subgoal `cons-insert-olist@0`; the goal viewer shows that (15) has also been added to the subgoal `insert-olist@3.2.0` as a new lemma.

As usual, we prove `cons-insert-olist@0` by induction on the structure of the list `L:List?`. The case generated for the base case is trivially discharged with auto whereas for the inductive case a boolean case analysis on `J*Int? <= V0#0*Int?` is needed. Unfortunately, the command auto cannot discharge the subgoal corresponding to the case when `J*Int? <= V0#0*Int?` is `true`, but presents us with:



label	module
cons-insert-olist	LIST-SORT

FORMULA11: Term: J:Int? Sort: --------------------

```
(A{I:Int? ; J:Int? ; L:List?}((((((((((((I:Int?): Int) & (FORMULA11))) & (FORMULA9))) & (FORMULA7))) & (FORMULA5))) => (FORMULA3))))
```

Figure 13: A lemma

```
Label: cons-insert-olist@2.2.0
Formula: cons(I*Int?,cons(V0#0*Int?,cons(J*Int?,nil))): OList.
```

and a number of hypothesis, including:

```
mb I*Int? : Int[hyp-1].
mb J*Int? : Int[hyp-2].
mb V0#0*Int? : Int[hyp-0].
mb cons(I*Int?,cons(V0#0*Int?,nil)): OList[hyp-3].
eq J*Int? <= V0#0*Int? = false[case--false].
```

However, it is then clear that the term

```
cons(I*Int?,cons(V0#0*Int?,cons(J*Int?,nil)))
```

is an ordered list, given that:

– `I*Int?` is less than or equal to `V0#0*Int?`, since, by hypothesis `[hyp-3]`, `cons(I*Int?,cons(V0#0*Int?,nil))` is an ordered list; and

– `V0#0*Int?` is less than `J*Int?`, by hypothesis `[case--false]`.

So why can't the ITP prove it? The reason is that nowhere is explicitly stated that `I*Int? <= V0#0*Int?` is equal to `true`. Now, this information can be extracted from the fact that `cons(I*Int?,cons(V0#0*Int?,nil))` is a term of sort `OList` by using the command term-split over this term. This prompts the ITP to make explicit all assumptions about `cons(I*Int?,cons(V0#0*Int?,nil))`, based on the membership axioms that define the sort `OList` and the fact that `cons` and `nil` have been declared as free constructors; the resulting subgoal can then be easily proved with auto.

We can try to prove the subgoal corresponding to the case when `J*Int? <= V0#0*Int?` is `false` using the same script: auto, term-split over the term `cons(I*Int?,cons(V0#0*Int?,V0#1*List?`, and auto again. However, this time, the last auto cannot discharge the subgoal, but presents us with:

```
Label: cons-insert-olist@2.2.1.0
Formula:
cons(V1#1*Int?,cons(V1#2*Int?,insert(J*Int?,V1#0*List?))): OList.
```

and a number of hypothesis, including:

```
Hipotesis:
mb J*Int? : Int[hyp-4].
mb V1#2*Int? : Int[hyp-9].
mb cons(V1#2*Int?,V1#0*List?): OList[hyp-10].
cmb cons(I:Int?,insert(J:Int?,V1#0*List?)): OList
    if I:Int? <= J:Int? = true /\ cons(I:Int?,V0#1*List?): OList
    /\ J:Int? : Int /\ I:Int? : Int[hyp-2].
eq V1#1*Int? <= V1#2*Int? = true[hyp-14].
eq J*Int? <= V1#2*Int? = false[case--false].
```

Again, it is clear that

```
cons(V1#1*Int?,cons(V1#2*Int?,insert(J*Int?,V1#0*List?)))
```

is an ordered list, given that:

- `V1#1*Int?` is less than or equal to `V1#2*Int?`, by hypothesis `[hyp-14]`; and
- `cons(V1#2*Int?,insert(J*Int?,V1#0*List?))` is an ordered list, by application of the induction hypothesis `[hyp-2]`, under the hypothesis `[case-false]`, `[hyp-10]`, `[hyp-4]`, and `[hyp-9]`.

So why can't the ITP prove it? Because it cannot prove that

```
insert(J*Int?,V1#0*List?)
```

is a term of sort `List`, which is a condition in the membership axiom that defines when terms of the form $\texttt{cons}(I, \texttt{cons}(J, L))$ are ordered lists. But, of course, `insert` satisfies this lemma:

$$\forall\{I, L\}(I : \texttt{Int} \wedge L : \texttt{List} \ \rightarrow \texttt{insert}(I, L) : \texttt{List}), \tag{16}$$

which can be easily proved by structural induction.

3.2.3 Lemmas to prove goals

Suppose that (16) has been proved as a goal, and that the proof script has been saved in a file `insert-list.itp`. Then, we can automatically insert (16) as a lemma in the subgoal `cons-insert-olist@2.2.1.0` by simply loading the file with the `load` button. With this additional information, the command `auto` succeeds in the proof of `cons-insert-olist@2.2.1.0`, and the proof of the lemma is completed.

We can now try again to discharge the subgoal `insert-olist@3.2.0` with the command `auto`. The command succeeds this time, thanks to the lemma `lem-cons-insert-olist@0`. This completes the proof of the fact that inserting an element into an ordered lists produces an ordered list. By clicking now on script viewer, we can save the script for future use. In fact, we will need this result as an auxiliary lemma for the main theorem we are interested in, that is, that the list returned by `insertion-sort` is ordered.

After introducing (13) in the usual way by means of the formula editor, we can try to prove it by structural induction. The base case is trivial and is discharged by the command `auto`. However, in the inductive case, `auto` simply transforms the goal into:

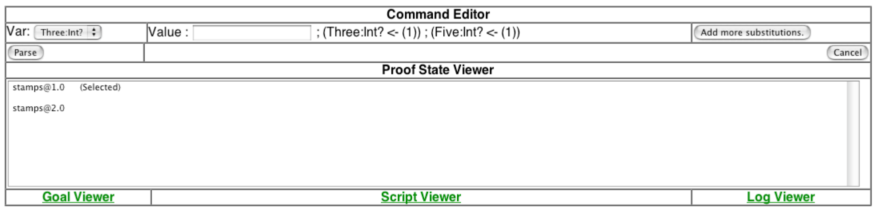
```
Label: insertion-sort-olist@2.0
Formula: insert(V0#0*Int?,insertion-sort(V0#1*List?)): OList.

Hypotheses:
mb V0#0*Int? : Int[hyp-0].
mb V0#1*List? : List[hyp-1].
mb insertion-sort(V0#1*List?): OList[hyp-2].

New constants:
op V0#0*Int? : -> Int? .
op V0#1*List? : -> List? .
```

It is clear that `insert(V0#0*Int?,insertion-sort(V0#1*List?))` is an ordered lists, since we know that inserting an element into an ordered lists leaves it ordered, and, by induction hypothesis `[hyp-2]`, `insertion-sort(V0#1*List?)` is an ordered list. What we need then is to introduce the lemma `insert-olist` in the subgoal `insertion-sort-olist@2.0`, by loading its proof script. After that, the command `auto` succeeds, as expected.

3.3 Quantifiers too

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 [Crow et al., 1995, 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



```
; (Three:Int? <- (1)) ; (Five:Int? <- (1))
```

```
stamps@1.0    (Selected)

stamps@2.0
```

Figure 14: Existential instantiation

some multiple of 5:

$$\forall\{I\}((I : \texttt{Nat}) \to \exists\{\mathit{Three}, \mathit{Five}\}(\mathit{Three} : \texttt{Nat} \land \mathit{Five} : \texttt{Nat} \\ \land\, I + 8 = (3 \times \mathit{Three}) + (5 \times \mathit{Five}))) \,, \tag{17}$$

where I, Five, and Three are variables of the kind `Int?`.

As such, this is simply a property about natural numbers and the corresponding specification is trivial and adds nothing to the default module; we just change its name to `STAMP`:

```
spec STAMP is
  kind Qid? = [Qid] .
  kind Bool? = [Bool] .
  kind Int? = [Int] .
  kind String? = [String] .
endspec
```

After introducing (17) with the formula editor, the proof proceeds by induction on `I:Int?`. The subgoal corresponding to the base case is:

```
Label: stamps@1.0
Formula: E{Five:Int? ; Three:Int?}
    (((0 + 8 = 3 * Three:Int? + 5 * Five:Int?)
     &(Three:Int?  : Nat))&(Five:Int? : Nat)).
```

Clearly, letting `Three:Int?` and `Five:Int?` both be `1` fulfills the base case; such instantiation is communicated to the ITP Web tool through the exist-inst-subs command. This takes us to a screen in which we specify the values to assign to the existential variables, as shown in Figure 14.

The command exist-inst-subs replaces the goal with the following one:

```
Label: stamps@1.0
Formula: ((0 + 8 = 3 * 1 + 5 * 1) & (1 : Nat)) & (1 : Nat).
```

To unfold the conjunction we need to apply the conj command twice, after which each of the conjuncts is immediately discharged with auto.

For the inductive step we need to find natural numbers *Three* and *Five* such that

$$(1 + I) + 8 = 3 * \mathit{Three} + 5 * \mathit{Five}$$

assuming that there exist natural numbers Three' and Five' such that

$$I + 8 = 3 * \mathit{Three}' + 5 * \mathit{Five}'$$

Notice that, if Five' is 0 we can take *Five* to be 2 and *Three* to be equal to $\mathit{Three}' - 3$; otherwise, we obtain the result by making *Five* equal to $\mathit{Five} - 1$ and *Three* to $\mathit{Three}' + 2$.

The ITP script mimics quite closely the proof above. First, auto transforms the subgoal `stamps@2.0` corresponding to the inductive case into:[4]

```
Label: stamps@2.0
Formula:
  E{Five:Int? ; Three:Int?}
    (((s V0#0*Int? + 8 = 3 * Three:Int? + 5 * Five:Int?)
     &(Three:Int? : Nat))&(Five:Int? : Nat)).
Hipotesis:
mb V0#0*Int? : Nat[hyp-0].
E{Five:Int? ; Three:Int?}(
   ((V0#0*Int? + 8 = 3 * Three:Int? + 5 * Five:Int?)
   &(Three:Int? : Nat))&(Five:Int? : Nat)) [hyp-1]

New constants:
op V0#0*Int? : -> Int? .
```

with the induction hypothesis labeled as `[hyp-1]`. Note, however, that `[hyp-1]` cannot be directly used by the command `auto` as *executable* additional information, that is, as information that can be used in the rewriting-based simplication steps. To remove the existential quantifier from `[hyp-1]`, we can use the command exist-inst-hyp which results in a new subgoal with the same formula to be proved but with the following *executable* hypotheses:

```
mb Five!hyp-1*Int? : Nat[hyp-2].
mb Three!hyp-1*Int? : Nat[hyp-2].
eq V0#0*Int? + 8
   = 3 * Three!hyp-1*Int? + 5 * Five!hyp-1*Int?[hyp-2].
```

[4] Note that the ITP internal constructors for the natural numbers are the constant 0, to represent the number 0, and the unary operator s, to represent the succesor of any number.

These hypotheses explicitely state that `Three!hyp-1*Int?` and `Five!hyp-1*Int?` are the *witnesses* of `[hyp-1]`, which is therefore removed from the list of available hypotheses. The rest of the proof is straightforward: we simply have to distinguish cases according to whether `Five!hyp-1*Int?` is zero or not, and instantiate `Three:Int?` and `Five:Int?` accordingly. The sequence of commands is: first, bool-split on `Five!hyp-1*Int? <= 0`; then, for the `true`-case, exist-inst-subst (with `Five:Int?` and `Three:Int?` instantiated, respectively, to `2` and `Three!hyp-1*Int? - 3`), `cnj` (twice), and `auto` (thrice); finally, for the `false`-case, exist-inst-subst (with `Five:Int?` and `Three:Int?` instantiated, respectively, to `Five!hyp-1*Int? - 1` and `Three!hyp-1*Int? + 2`), `cnj` (twice), and `auto` (thrice).

4 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. As an introduction to the tool, we have presented membership equational logic and discussed its adequacy for the specification and verification of semantic data structures, such as ordered lists, binary search trees, priority queues, and powerlists. An interesting feature of the proposed semantic framework is that incompletely specified operations can be reasoned about so as to support incrementality. That is, unlike in most reasoning systems including RRL [Kapur and Zhang, 1995] and ACL2 [Kaufmann et al., 2000], operations do not have to be completely specified before inductive properties about them can be verified mechanically.

The ITP tool is currently available as a web-based application that includes a module editor, a formula editor, and a command editor. These editors allow users to create and modify their specifications, to formalize properties about them, and to guide their proofs by filling and submitting web forms. 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 powerful integration of term rewriting with a decision procedure for linear arithmetic with uninterpreted function symbols [Clavel et al., 2004], while also available in other rewriting-based theorem provers like RRL [Kapur and Zhang, 1995], 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. Another interesting extension of the tool is the implementation of the cover set induction method [Kapur and Subramaniam, 1996], a feature already available in RRL [Kapur and Zhang, 1995].

Acknowledgments. We thank Deepak Kapur and José Meseguer for many interesting discussions on the subject of inductive theorem proving; our current understanding of the semantics underlying the ITP tool has greatly benefited from their questions, criticisms, and suggestions to previous proposals. We also thank Marina Egea, Narciso Martí-Oliet, and Alberto Verdejo for very detailed comments on a several draft version of this tutorial.

References

[Bouhoula et al., 2000] Bouhoula, A., Jouannaud, J.-P., and Meseguer, J. (2000). Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132.

[Clavel et al., 2005] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. (2005). Maude manual (version 2.2). `http://maude.cs.uiuc.edu/manual/`.

[Clavel et al., 2004] Clavel, M., Palomino, M., and Santa-Cruz, J. (2004). Integrating decision procedures in reflective rewriting-based theorem provers. In Antoy, S. and Toyama, Y., editors, *Fourth International Workshop on Reduction Strategies in Rewriting and Programming*, pages 15–24. Technical report AIB-2004-06, Department of Computer Science, RWTH, Aachen.

[Crow et al., 1995] Crow, J., Owre, S., Rushby, J., Shankar, N., and Srivas, M. (1995). A tutorial introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida*. `http://www.csl.sri.com/papers/wift-tutorial/`.

[Enderton, 2000] Enderton, H. B. (2000). *A Mathematical Introduction to Logic, Second Edition.* Academic Press.

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

[Kapur and Zhang, 1995] Kapur, D. and Zhang, H. (1995). An overview of rewrite rule laboratory (rrl). *J. Computer and Mathematics with Applications*, 29(2):91–114.

[Kaufmann et al., 2000] Kaufmann, M., Manolios, P., and Moore, J. S. (2000). *Computer-Aided Reasoning: An Approach.* Kluwer Academic Press.

[Meseguer, 1998] Meseguer, J. (1998). Membership algebra as a logical framework for equational specification. In Parisi-Presicce, F., editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3 - 7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer-Verlag.