The ITP Tool

Manuel Clavel
Department of Philosophy
University of Navarre

October 13, 2000

Abstract

The ITP tool is an inductive theorem prover for proving properties of Maude functional modules, 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 in rewriting logic of the formal inference system that it implements. This document presents the main features of the ITP tool and describes how to obtain and install it, and how to use it effectively.

Contents

1 Overview

This document introduces the ITP tool which is an interactive inductive theorem prover for proving properties of Maude functional modules, i.e., specifications in membership equational logic with an initial algebra semantics.

An important feature of the ITP tool is that it is written entirely in Maude and is in fact and *executable specification* in rewriting logic of the formal inference system that it implements. This inference system treats equational specifications as *data* — for example, it may add an induction hypothesis as a new equational axiom. This makes a *reflective* design — in which Maude functional modules become data at the metalevel — ideally suited for the ITP tool. Indeed, the fact that rewriting logic is a reflective logic [9, 8], and that Maude efficiently supports reflective rewriting logic computations is systematically exploited in this tool.

Rewriting logic reflection has one important additional advantage, namely, the modular and declarative treatment of *proof tactics* [7, 9]. Such tactics can be implemented by rewrite rules at the metalevel of the Maude module implementing the ITP inference system, and can be changed at will without any modification to the inference system itself.

The very high level of abstraction at which the ITP tool has been developed has made it relatively easy to build, makes understanding their implementation much easier, and will make their maintenance and future extensions much easier than if a conventional implementation, say in C, C++, or Java, had instead been chosen. Thanks to the high performance of the Maude engine, these important benefits are achieved without sacrificing performance. Indeed, the ITP tool is very responsive in interactive mode, with input/output time typically dominating over computation time.

The remainder of this document is organized as follows. Section 2 explains how to obtain and install the ITP tool. Section 3 is a tutorial on how to use interactively the ITP tool to prove inductive properties of Maude functional modules. We include three appendices at the end of the document. Appendix A introduces the ITP inference system. Appendix B explains the reflective design of the ITP tool. Finally, Appendix C contains the ITP grammar.

Aknowledgements

The ITP tool is part of a set of proving tools developed by Francisco Durán, Steven Eker, and José Meseguer for Maude equational specifications. The Maude group is completed with Patrick Lincoln, Narciso Martí-Oliet, and José Francisco Quesada, all of whom have contributed suggestions, solutions, and encouragement. The theoretical foundations of these tools have benefited very much from the work done by Adel Bouhoula, Jean-Pierre Jouannaud and José Meseguer on theorem proving techniques for membership equational logic [3, 17]. The ITP tool has also profited from an ongoing collaboration with José Meseguer and David Basin on inductive (meta-)theorem proving techniques for membership equational logic and rewriting logic [2]. I thank Narciso Martí-Oliet for his detailed comments on an earlier draft of this paper.

The design and implementation of the ITP tool has been partially supported by the Advanced Software Enrichment Project of the Information-Technology Promotion Agency, Japan (IPA), by DARPA and NASA through Contract

NAS2-98073, and by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114.

2 Installation

2.1 Getting the ITP Tool

A compressed `tar` file of the ITP sources and related files may be obtained at `http://maude.csl.sri.com/tools/itp/`

Having obtained a compressed `tar` version, save it into a file `itp.tar.gz` say, and then extract its contents with

```
gzip -d itp.tar.gz
tar xvf itp.tar
```

You should then have the following:

- `README` file;
- `Changes` index with latest changes;
- `itp-tool` Maude module, implementing the ITP tool;
- `itp-tool-aux` Maude module, imported by `itp-tool`;
- `itp-tactics` Maude module, imported by `itp-tool`;
- `meta-prelude` Maude module, imported by `itp-tactics`;
- `itp-rules` Maude module, implementing the ITP inference system;
- `examples/` subdirectory, containing the proof examples described in Sections 3.5 and other examples;
- `docs/` subdirectory, containing the files needed to create a version of this manual.

2.2 Requirements

The ITP tool is written entirely in Maude. The Maude system is freely available at `http://maude.csl.sri.com`. To use the ITP tool you need version 1.0.3 or later.

3 Tutorial

3.1 Introduction

This is a tutorial on how to use the ITP tool as an interactive inductive theorem prover for equational specifications in Maude. Maude [4, 5] is a high-level language and high-performance system supporting both equational and rewriting logic specifications.

In Maude there are three kinds of modules: *functional modules*, *system modules*, and *object-oriented modules*. Functional modules are equational specifications, system modules are rewriting logic specifications, and both, functional modules and system modules, have an initial semantics. Object-oriented modules are reduced to a special case of system modules for which a convenient syntax is used.

The Maude system, its documentation, a tutorial, a collection of examples, some case studies, and related papers are available on the Maude web page at `http://maude.csl.sri.com`.

3.2 Functional Modules, Proofs, and Interaction

Maude functional modules are equational specifications. We assume that the reader is familiar with the basic methodology of equational specifications. Very briefly, they are used to declare different kinds of data together with the operations that act upon them. The behavior of the operations is described by means of conditional equations. It is useful to distinguish two kinds of operations: constructors, that are used to construct or generate data, and the remaining operations.

The version of equational logic that provides the logical basis for functional modules is *membership equational logic*. Membership equational logic is an expressive version of equational logic supporting sorts, subsorts, partiality, and overloading of function symbols. A full account of the syntax and semantics of membership equational logic can be found in [3, 17]. (Appendix A.1 includes a brief introduction to membership equational logic.)

Functional modules are assumed to be confluent and terminating, and their operational semantics is *equational simplification*, that is, rewriting of terms until a normal form is obtained.[1]

The mathematical semantics of equational specifications — based on initial algebras [13] — gives precise meaning to functional modules and supports proving properties about them. The proof theory of the underlying equational logic may be applied to prove properties of functional modules. However, most of their interesting properties are properties of their initial or *standard* models, rather than properties of *all* its models. In general, pure equational deduction is inadequate for proving these properties, and it is necessary to use inductive techniques, sometimes complemented by other techniques.

The ITP tool implements a sound inference system for proving properties of functional modules. This inference system includes a "structural induction" principle, that allows proving that a property holds in the initial model of a functional module by proving that it holds for each constructor whenever it holds for the arguments of that constructor.

This tutorial is concerned with showing how to prove theorems about the initial models of functional modules. A theorem to be proved is submitted to the ITP tool as the initial goal, and then this goal is succesively transformed by controlled rewriting — using the ITP inference rules as rewrite rules — into different sets of subgoals, until no subgoals are left if the proof succeeds.

[1] The Maude system does not check the confluence and termination properties. There is however a set of tools that can be used to prove these properties under certain restrictions. `http://maude.csl.sri.com/tools` provides the latest version of these tools.

The user interacts with the ITP tool by invoking predefined theorem proving commands. The ITP commands are implemented in the ITP tool as strategies that control the application of the ITP inference rules. Hence *proof scripts* are sequences of controlled applications of rewrite rules that perform specific theorem proving tasks.

3.3 Getting Started

The ITP implementation is separated into several modules, organized in a module hierarchy. The top module in this hierarchy is the `itp-tool` module. Assuming that you are running Maude, type "`in itp-tool`" in Maude to start the ITP tool. This loads the `itp-tool` module and the rest of the modules upon which it depends, and initializes the read-eval-print loop for interacting with the ITP tool. The full path name must be given if `itp-tool` is not in the current working directory. All communication with the ITP tool must be enclosed in parenthesis (see [4] for more information about the `LOOP-MODE` module in Maude and the read-eval-print loop that it provides.)

While you are developing a proof, we recommend to type each proof command in a file and then enter it into Maude by copy-and-paste, thus ensuring that you have a complete record of your proof. To restart a proof, type "`loop init .`" (this message is not enclosed in parenthesis, since it is directed to the Maude system itself.)

3.4 A Simple Functional Module

We begin with a simple classical example, the natural numbers in Peano notation. To construct the data we use a constant `0` and a succesor operation `s`. Therefore, `0` and `s` are constructor operators and are declared with the corresponding `ctor` attribute.

```
fmod NAT is
including BOOL .
sort Nat .
op 0 : -> Nat  [ctor] .
op s : Nat -> Nat  [ctor] .
op _+_ : Nat Nat -> Nat .
vars N M L : Nat .
eq N + 0 = N .
eq N + s(M) = s(N + M) .
endfm
```

In this example, the only other operation defined over natural numbers is addition, written in mixfix syntax, and equationally defined in a recursive way by structural induction on its second argument.

3.5 Some Basic Proofs

We prove here that associativity and commutativity hold for the `_+_` function in the initial model of `NAT` — isomorphic to the usual set of natural numbers with addition. (There are *nonstandard* models of the equational theory `NAT` for which addition is neither associative nor commutative.) This will illustrate not just the basic proof commands but also the typical proof process.

3.5.1 Associativity of Addition

Main Goal: `(N + (M + L)) = ((N + M) + L)`

This goal can be entered to the ITP by typing

```
(goal
fmod NAT is
including BOOL .
sort Nat .
op 0 : -> Nat [ctor] .
op s : Nat -> Nat [ctor] .
op _+_ : Nat Nat -> Nat .
vars N M L : Nat .
eq (N + 0) = (N) .
eq (N + s(M)) = (s(N + M)) .
endfm
|- {N ; M ; L}((N + (M + L)) = ((N + M) + L)) .
)
```

This establishes a new goal to be proved. The ITP's response is to print the current proof state:

```
=================================
1
=================================

|-ind {N ; M ; L} (N + (M + L) = (N + M) + L)
```

! Notice that we have slightly modified the module `NAT`: terms in its equations are now enclosed in parentheses, and the module `BOOL` is explicitly imported. These modifications are required by the ITP interface to correctly parse functional modules. (See Appendix C.1 for more details.)

Until we have finished a proof, the proof state always looks like a sequence of subgoals like this:

```
++++++++++++++++++++++++++++++++++
==================================
⟨label_1⟩
==================================

|-⟨mod_1⟩ φ_1
⋮
==================================
⟨label_n⟩
==================================

|-⟨mod_n⟩ φ_n
```

where ϕ_i is among the theorems that we need to prove to establish the initial goal, $\langle mod_i \rangle$ indicates the *mode* in which ϕ_i must hold, namely, either in the initial model (`ind`) or in all models (`vrt`) of a functional module, and $\langle label_i \rangle$ is a string of natural numbers that is used to label each subgoal. To remember in which functional module a subgoal must be proved, type "`(show (`⟨*label*⟩`) .)`"

Let us now get back to the associativity equation:

```
(N + (M + L)) = ((N + M) + L).
```

Properties of functions defined by recursion are best established by induction. There are three variables that we could induct on: `N`, `M`, and `L`. Since `_+_` is recursively defined by structural induction on the second argument, `L` is the correct one. The choice of `L` as induction variable is indicated by typing

```
(ind (1) on L .)
```

This instructs the ITP tool to apply its induction inference rule on the variable `L` in subgoal `1`. The new proof state contains two subgoals, namely, the base case (`0`) and the induction step (`s`):

```
++++++++++++++++++++++++++++++++++
==================================
1 . 1
==================================

|-ind {N:1}({N ; M} (N + (M + N:1) = (N + M) + N:1)
        ==> {N ; M} (N + (M + s(N:1)) = (N + M) + s(N:1)))
==================================
1 . 2
==================================

|-ind {N ; M} (N + (M + 0) = (N + M) + 0)
```

The variable `N:1` in the induction step is a new variable of sort `Nat` generated by the ITP tool.

Let us try to solve subgoal `1 . 2` automatically:

```
(simp (1 . 2). )
```

This command tells the ITP tool to apply a proof strategy `simplify` to subgoal `1 . 2`. The strategy `simplify` tries to "simplify" a subgoal using the ITP inference rules, in particularly, equational simplification. In this case, subgoal `1 . 2` is solved completely (thanks to the equation `eq N + 0 = N`) and dissapears. We are then left with subgoal `1 . 1` which has not changed form the previous state.

```
+++++++++++++++++++++++++++++++++
=================================
1 . 1
=================================

|-ind {N:1}({N ; M} (N + (M + N:1) = (N + M) + N:1)
        ==> {N ; M} (N + (M + s(N:1)) = (N + M) + s(N:1)))
```

Now, let us try to solve automatically the remaining subgoal.

```
(simp (1 . 1). )
```

This subgoal is solved completely (thanks to the equation `eq (N + s(M)) = (s(N + M))` and the induction hypothesis), and we obtain the desired result:

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

3.5.2 Commutativity of Addition

Main Goal: `((N + M) = (M + N))`
We start the proof as usual:

```
(goal
fmod NAT is
...
endfm
|- {N ; M}((N + M) = (M + N)) .
)
```

In this case, there are two variables that we could induct on: `N` and `M`. As before, since `_+_` is recursively defined by structural induction on the second argument, we induct on `M`.

```
(ind (1) on M .)
```

The new proof state contains two subgoals that correspond to the base case (`0`) and the induction step (`s`):

```
+++++++++++++++++++++++++++++++++
=================================
1 . 1
=================================

|-ind {N:1} ({N} (N + N:1 = N:1 + N)
```

```
              ==> {N} (N + s(N:1) = s(N:1) + N))
==================================
1 . 2
==================================

|-ind {N} (N + 0 = 0 + N)
```

Let us try to simplify subgoal `1 . 2`:

```
(simp (1 . 2) .)
```

In this case the subgoal `1 . 2` is not completely solved and we are left with the following state:

```
+++++++++++++++++++++++++++++++++
==================================
1 . 1
==================================

|-ind {N:1} ({N} (N + N:1 = N:1 + N)
              ==> {N} (N + s(N:1) = s(N:1) + N))
==================================
1 . 2
==================================

|-vrt N* = 0 + N*
```

The constant `N*` is a new constant of sort `Nat` generated by the ITP tool.

In order to further simplify subgoal `1 . 2`, a lemma suggests itself:

-First Lemma: `(0 + N) = (N)`

For this lemma the canonical proof procedure:

```
(goal
fmod NAT is
...
endfm
|- {N}((0 + N) = (N)) .
)

(ind (1) on N .)
(simp (1 . 2) .)
(simp (1 . 1) .)
```

leads to the desired message `q.e.d`.

Now we restart the proof of the **Main Goal**, but we begin by adding the **First Lemma** to `NAT`.

```
(goal
fmod NAT is
...
endfm
|- {N ; M}((N + M) = (M + N)) .
)

(lem {N}((0 + N) = (N)) to (1) .)
```

This tells the ITP tool that the equation `0 + N = N` can be used in all future proofs by simplication. It also generates a new subgoal `1 . 2` that corresponds to this lemma. Notice that the initial goal is relabelled to `1 . 1`.

```
++++++++++++++++++++++++++++++++
=================================
1 . 1
=================================

|-ind {N ; M} (N + M = M + N)

=================================
1 . 2
=================================

|-ind {N} (0 + N = N)
```

To solve the new subgoal `1 . 2`, we apply the above proof script for proving the **First Lemma**, with the appropiate relabelling of its subgoals.

```
(ind (1 . 2) on N .)
(simp (1 . 2 . 2) .)
(simp (1 . 2 . 1) .)
```

This time the base case of the **Main Goal** is solved automatically:

```
(ind (1 . 1) on M .)
(simp (1 . 1 . 2) .)

++++++++++++++++++++++++++++++++
=================================
1 . 1 . 1
=================================

|-ind {N:1}({N} (N + N:1 = N:1 + N)
            ==> {N} (N + s(N:1) = s(N:1) + N))
```

The induction step however merely simplifies to:

```
(simp (1 . 1 . 1) .)

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

```
=================================
1 . 1 . 1
=================================

|-vrt s(N:1* + N*) = s( N:1*) + N*
```

We then need another lemma:

-Second Lemma: `(s(M) + N) = (s(M + N))`

To prove this lemma we need also the **First Lemma**. Thus we begin its proof by adding this lemma to `NAT`, with the corresponding proof scritp:

```
(goal
fmod NAT is
...
endfm
|- {N ; M}((s(M) + N) = (s(M + N))) .
)

(lem {N}((0 + N) = (N)) to (1) .)
(ind (1 . 2) on N .)
(simp (1 . 2 . 2) .)
(simp (1 . 2 . 1) .)
```

Thus, the canonical proof procedure

```
(ind (1 . 1) on N .)
(simp (1 . 1 . 2) .)
(simp (1 . 1 . 1) .)
```

leads to the desired message `q.e.d`.

Now we can restart again the original proof, but adding this time the **First** and **Second Lemma** with the corresponding proof scripts:

```
(goal
fmod NAT is
...
endfm
|- {N ; M}((N + M) = (M + N)) .
)

(lem {N}((0 + N) = (N)) to (1) .)
(ind (1 . 2) on N .)
(simp (1 . 2 . 2) .)
(simp (1 . 2 . 1) .)

(lem {N ; M}((s(M) + N) = (s(M + N))) to (1 . 1) .)
(ind (1 . 1 . 2) on N .)
(simp (1 . 1 . 2 . 2) .)
(simp (1 . 1 . 2 . 1) .)
```

and then solve the **Main Goal** using the canonical proof procedure:

```
(ind (1 . 1 . 1) on M .)
(simp (1 . 1 . 1 . 2) .)
(simp (1 . 1 . 1 . 1) .)

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

This finishes our introduction to performing proofs in ITP. The examples should have familiarized you with:

- the procedure for proving a theorem in the ITP tool: state a goal; proceed with its proof until a lemma is required; prove that lemma; restart the original proof including now this lemma as a new equation.
- a specific procedure that often works well for functional modules: induction followed by simplication via the `simplify` strategy.

3.6 A Summary of Commands

As illustrated in Section 3.5, the user interacts with the ITP tool by invoking predefined theorem proving commands. A proof begins by typing:

- "`(goal` ⟨*module*⟩ `|-` ⟨*formula*⟩ `.)`"

where ⟨*module*⟩ is, essentially, any Maude functional module (see Appendix C.1 for more details), and ⟨*formula*⟩ is any atomic sentence in membership equational logic over the signature of ⟨*module*⟩ (Appendix C.2 includes the ITP's syntax for formulae).

The ITP commands are implemented inside the ITP tool as strategies that control the application of the ITP inference rules as rewrite rules. Hence proof scripts are sequences of controlled applications of rewrite rules that perform specific theorem proving tasks. If a command cannot be applied, the ITP tool prints the message "`The requested strategy cannot be applied`".

We list below the basic ITP theorem proving commands, indicating in each case, in parentheses, which inference rule it applies. The ITP inference rules are listed in Appendix A.3. The ⟨*label*⟩ inside a command always indicates the subgoal to which the command is applied. (See Appendix C.3 for more details on the syntax of commands.)

- "`(ind (`⟨*label*⟩`) on` ⟨*var*⟩ `.)`" [rule 1] inducts on the variable ⟨*var*⟩.
- "`(vrt (`⟨*label*⟩`) .)`" [rule 2] reduces proving a formula in the initial model to proving it in the variety of all models.
- "`(cns (`⟨*label*⟩`) .)`" [rule 3] reduces proving a universal formula to proving an instance of it in which the variables are replaced by new constants.
- "`(imp (`⟨*label*⟩`) .)`" [rule 4] reduces proving a conditional formula to proving that the consequent holds under the assumption that the antecedents hold.

- "`(rwr (`⟨*label*⟩`) .)`" [rule 5] simplifies a ground equality by reducing to canonical form its left- and right-hand sides.
- "`(lem` ⟨*formula*⟩ `to (`⟨*label*⟩`) .)`" [rule 6] adds a lemma to a proof.

Very often you will like to simplify automatically a formula by invoking the command

- "`(simp (`⟨*label*⟩`) .)`"

as shown in the proofs in Section 3.5. Finally, recall that at any point of the proof you can see in which functional module a formula must be proved by typing

- "`(show (`⟨*label*⟩`) .)`"

A The ITP Logic

In this section we introduce the inference system implemented in the ITP tool for proving inductive properties of Maude functional modules. Recall that Maude functional modules are membership equational theories satisfying the additional requirement of being Church-Rosser, terminating and sort-decreasing [3]. We begin with an introduction to membership equational logic, its syntax and its semantics. Then, we explain in detail the structural induction principle supported by the ITP. Finally, we present the entire ITP inference system.

A.1 Membership Equational Logic

Membership equational logic is an expressive version of equational logic supporting sorts, subsorts, partiality, and overloading of function symbols. A full account of the syntax and semantics of membership equational logic can be found in [3, 17]. Here we define the basic notions needed for this paper.

A signature in membership equational logic is a triple $\Omega = (K, \Sigma, S)$ with K a set of *kinds*, Σ a K-kinded signature $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^*\times K}$, and $S = \{S_k\}_{k\in K}$ a pairwise disjoint K-kinded family of sets. We call S_k the set of *sorts* of kind k. The pair (K, Σ) is what is usually called a many-sorted signature of function symbols; however we call the elements of K *kinds* because each kind k now has a set S_k of associated *sorts*, which in the models will be interpreted as subsets of the carrier for the kind. Also, as usual, we denote by T_Σ the K-kinded algebra of ground Σ-terms, and by $T_\Sigma(X)$ the algebra of Σ-terms on the K-kinded set of variables X.

The atomic formulae of membership equational logic are either equations $t = t'$, where t and t' are Σ-terms of the same kind, or *membership assertions* of the form $t : s$, where the term t has kind k and $s \in S_k$. Sentences are Horn clauses on these atomic formulae, i.e., sentences of the form

$$\forall(x_1, \ldots, x_m).\, A_1 \wedge \ldots \wedge A_n \Rightarrow A_0\,,$$

where each A_i is either an equation or a membership assertion, and each x_j is a K-kinded variable. For example, Figure 1 gives a set of membership equational axioms specifying the natural numbers. In the functional module `NAT` of

$$\begin{array}{rcl} & & \texttt{0:Nat}, \\ \forall(N).\, N\texttt{:Nat} & \Rightarrow & \texttt{s(}N\texttt{):Nat} \end{array}$$

Figure 1: Membership equational axioms for the natural numbers.

Section 3.4, these two axioms exactly correspond to the (constructor) operator declarations for `0` and `s`, which are membership axioms in disguise.

A theory in membership equational logic is a pair (Ω, E), where E is a finite set of sentences in membership equational logic over the signature Ω. The way in which partiality is expressed in membership equational logic is by the fact that terms always have a kind, but may not have a sort. Terms for which a sort cannot be established from the axioms E correspond to *undefined* or *error* elements.

We employ standard semantic concepts from many-sorted logic. Given a signature $\Omega = (K, \Sigma, S)$, an Ω-algebra is a many-kinded Σ-algebra (a K-indexed-set $A = \{A_k\}_{k \in K}$ together with a collection of appropriately kinded functions interpreting the function symbols in Σ) together with an assignment to each sort $s \in S_k$ of a subset $A_s \subseteq A_k$. Hence, sorts can be thought of as unary predicates that semantically denote subsets of the appropriate kind. An algebra A and a (kind-respecting) valuation σ satisfy an equation $t = t'$ iff $\sigma(t) = \sigma(t')$, where we overload notation by identifying σ with its unique homomorphic extension to Σ-terms. We write $A, \sigma \models t = t'$ to denote such a satisfaction. Similarly, $A, \sigma \models t : s$ holds iff $\sigma(t) \in A_s$.

Note that an Ω-algebra is nothing but a K-kinded first-order model with function symbols Σ and an alphabet of unary predicates $\{S_k\}_{k \in K}$. Therefore, the satisfaction relation can be extended to Horn and first-order formulae ϕ over these atomic formulae in the standard way. We write $A \models \phi$ when ϕ is satisfied for all valuations σ, and then say that A is a model of ϕ. Similarly, a theory (Ω, E) in membership equational logic is simply a Horn theory for the associated signature, when Ω is viewed as a first-order K-kinded signature. As usual, for ϕ a first-order sentence in the language of Ω, we write $(\Omega, E) \models \phi$, when all the models of the set E of sentences are also models of ϕ.

Theories in membership equational logic have initial models [17]. This provides the basis for reasoning by induction, as is explained in detail in Section A.2. We write $(\Omega, E) \models\!\!\!\simeq \phi$ to denote that the initial model of the membership equational logic theory (Ω, E) is also a model of ϕ.

A.2 The ITP Induction Principle

Since membership equational logic is a subset of equational Horn logic (indeed, they can be shown to be equivalent [17]), it follows immediately that any theory (Ω, E) has a unique (up to isomorphism) initial model [13]. The following is an induction principle for reasoning about properties of sorts, with respect to this model.

Definition 1 (Induction over sort definitions) *Let (Ω, E) be a theory in membership equational logic and let s be a sort in some S_k. Let $\{C_1, \ldots, C_n\}$*

be those sentences in E that specify s, i.e., those C_i of the form

$$\forall(x_1, \ldots, x_{p_i}).\, A_1 \wedge \ldots \wedge A_{q_i} \Rightarrow A_0\,, \tag{1}$$

where, for some t of kind k, A_0 is $t : s$.

For τ a first-order formula with free variable x of sort s over the signature Ω, an induction principle for (Ω, E) with respect to $x : s$ and $\tau(x)$ *is the formula*

$$\psi_1 \wedge \ldots \wedge \psi_n \Rightarrow \forall x : s.\, \tau(x) \tag{2}$$

where, for $1 \leq i \leq n$ and C_i of the form (1), ψ_i is

$$\forall(x_1, \ldots, x_{p_i}).\, [A_1]_\tau \wedge \ldots \wedge [A_{q_i}]_\tau \Rightarrow [A_0]_\tau \tag{3}$$

and, for $0 \leq j \leq q_i$

$$[A_j]_\tau = \begin{cases} \tau(t) & \text{if } A_j = t : s, \text{ for } t \text{ of kind } k \\ A_j & \text{otherwise.} \end{cases}$$

For a given membership equational theory (Ω, E), the above definition provides an induction schema (*ind*), given by (2), in many-kinded first-order logic over the signature Ω. Note that for $q_i = 0$, the nullary conjunction in the antecedent of (3) is true and the implication can be replaced with the succedent $\tau(t)$.

In the initial model of a membership equational theory, sorts are interpreted as the smallest sets satisfying the axioms in the theory, and equality is interpreted as the smallest congruence satisfying those axioms. Alternatively, the sets interpreting sorts can be characterized as being inductively generated in stages. This corresponds to the fixedpoint characterization of the least Herbrand model of a collection of Horn clauses [18], and the induction principle we have given formalizes induction over the stages in which the set is inductively defined [1]. By induction over the stages of the inductive definition of a sort s, which amounts to an induction over the proof that some ground term of kind k is of sort s, we can establish that reasoning in the membership equational theory (Ω, E), augmented by (*ind*), is sound.

Theorem 1 (Soundness) *Let (Ω, E) be a membership equational theory. If $(\Omega, E \cup \{(ind)\}) \vdash \tau$, then $(\Omega, E) \models\!\simeq \tau$.*

As an example, consider the membership equational theory for the natural numbers previously given in Figure 1. Definition 1 gives rise to the following induction principle (*ind*) over the sort `Nat` with respect to this theory.

$$[\tau(\mathtt{0}) \;\wedge \forall(N).(\tau(N) \Rightarrow \tau(\mathtt{s}(N)))] \Rightarrow\; \forall N : \mathtt{Nat}.\, \tau(N).$$

Note that other induction principles are possible. In particular (*ind*) takes *all* of the sentences that specify membership in s as constituting the inductive definition of s. In some cases, a subset of the sentences (sometimes called *generators* or *constructors*) is sufficient to characterize an inductive definition. Of course additional proof obligations then arise, e.g., sufficient completeness of the chosen subset (see [3, 6]).[2]

[2] In fact, the inference rule (*ind*) implemented in the ITP tool only consideres as constituting the inductive definition of a sort s the subset of sentences defining s that have been "marked" by the user with the attribute `ctor`. Therefore, the proofs of ITP are always understood to be relative to satisfying the proof obligation of "sufficient generation" for the declared constructors. However, these proof obligations can also be satisfied by induction using ITP, according to the general method explained in [6, Section 4.3].

A.3 The ITP Inference System

The following rules 1–6 provide a sound inference system — the ITP inference system — for proving that a universally quantified atomic formula of the form $(\forall x)t = t'$, or of the form $(\forall x)t : s$, is an inductive property of a given functional module M.

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

1. **Induction over sort definitions**

$$\frac{M \models\!\!\simeq \forall(x).\, p}{M \models\!\!\simeq \psi_1 \cdots M \models\!\!\simeq \psi_n}$$

where $\forall(x).\, p$ is an atomic sentence in membership equational logic, x is a variable of sort s in M, s is defined by a set of Horn clauses $\{C_1, \ldots, C_n\}$ of the form (1), and, for $1 \leq i \leq n$ and C_i of the form (1), ψ_i is defined as in (3).

2. **Proof in Variety**

$$\frac{M \models\!\!\simeq p}{M \vdash p},$$

where p is a sentence in membership equational logic.

3. **Constants Lemma**

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

where $\forall(x_1, \ldots, x_n).p$ is a sentence in membership equational logic and, for $1 \leq i \leq n$, x_i has sort s_i in M and c_i is a new constant that does not occur in M. Note that `op_:->_.` is the syntax for declaring constants in Maude.

4. **Implication Elimination**

$$\frac{M \vdash \forall(X_1).t_1 = t'_1 \wedge \cdots \wedge \forall(X_{n-1}).t_{n-1} = t'_{n-1} \Rightarrow \forall(X_n).t_n = t'_n}{M \cup \{\texttt{eq}\ t_1 = t'_1. \cdots \texttt{eq}\ t_{n-1} = t'_{n-1}.\} \vdash \forall(X_n).t_n = t'_n}$$

if for $i \in \{1, \ldots, n-1\}$, $(\mathcal{V}ar(t_i) \cup \mathcal{V}ar(t'_i)) \subseteq X_i$. Note that `eq_=_.` is the syntax for introducing equations in Maude.

$$\frac{M \vdash \forall(X_1).t_1 : s_1 \wedge \cdots \wedge \forall(X_{n-1}).t_{n-1} : s_{n-1} \Rightarrow \forall(X_n).t_n : s_n}{M \cup \{\texttt{mb}\ t_1 : s_1. \cdots \texttt{mb}\ t_{n-1} : s_{n-1}.\} \vdash \forall(X_n).t_n : s_n}$$

if for $i \in \{1, \ldots, n-1\}$, $\mathcal{V}ar(t_i) \subseteq X_i$. Note that `mb_:_.` is the syntax for introducing membership axioms in Maude.

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

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

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

6. **Equational and Membership Lemmas**

$$\frac{M \models p}{M \models \forall(X).t = t' \qquad M \cup \{\texttt{eq}\ t = t'.\} \models p}$$

$$\frac{M \models p}{M \models \forall(X).t : s \qquad M \cup \{\texttt{mb}\ t : s.\} \models p}$$

where $\forall(X).t = t'$, $\forall(X).t : s$, and p are atomic sentences in membership equational logic.

The explicit induction style of the ITP rules of inference is similar to that used in other explicit induction approaches in equational theorem proving such as those of Larch [14], and those developed for OBJ specifications [11].

B The ITP Implementation

In Appendix A.3 we have introduced the ITP inference system for proving inductive properties of Maude functional modules. Here we explain how to combine the use of rewriting logic as a logical framework [10, 15, 16] and the reflective capabilities of Maude to implement the ITP tool. ([6] provides further details on building theorem proving tools in Maude.)

Figure 2 illustrates our reflective design for the ITP tool. The functional module M, about which we want to prove inductive theorems, is at the object level. The ITP inference system uses M as data and therefore is specified as a system module `ITP-RULES` at the metalevel. Finally, different *proof tactics* to guide the application of the rewrite rules in `ITP-RULES` are strategies, which are defined at the meta-metalevel in a module `ITP-TACTICS`. Strategies are needed here, since the ITP inference rules are specified as rewrite rules that are not Church-Rosser or terminating. Hence, it is important to have some way of controlling the application of these rewrite rules in order to steer rewriting in the desired direction. Maude users can control rewriting by specifying, at the metalevel, their own rewriting strategies. (See [4, 9] for more details on defining strategies in Maude.)

Operationally, to use the ITP tool, the user submits as an initial goal the pair formed by a functional module and the atomic sentence in membership equational logic which is to be proved, and then this goal is successively transformed by rewriting — using the ITP inference rules as rewrite rules — into different sets of subgoals, until (in the case of a successful proof) no subgoals are left. The application of the ITP inference rules as rewrite rules is controlled by the user using strategies.

To the working logician or computer scientist, a reflective tool like the ITP may seem complicated and not particularly user-friendly since there is quite a

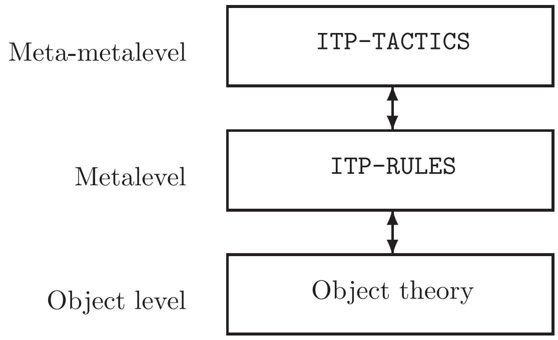

Figure 2: Reflective design for an inductive theorem prover.

bit of encoding involved in stating a theorem and in carrying out its proof. In particular, reasoning can involve several levels (object, meta, meta-meta, ...). As illustrated by the examples in Section 3, the ITP interface — fully specified in Maude — avoids many of the practical problems of working with such a reflective hierarchy. This interface hides all the levels of encoding from the user. Hence the user can actually abstract away many of the representation details and focus on the essential structure of proofs of theorems.

Finally, note that building the ITP tool using different levels of reflection results in a modular design with a clean separation between the logical and the control components. For example, we can simply extend the tool by specifying additional inference rules in the module `ITP-RULES` without changing the strategies defined in the module `ITP-TACTICS` and vice versa. Thanks to the high performance of the Maude engine these important benefits in ease of development, understandability, maintainability, and extensibility, are achieved without sacrificing performance. In fact, the ITP is very responsive in interactive mode, with input/output time typically dominating over computation time.

C The ITP Reference Manual

C.1 The Syntax of Admissible Modules

Admissible modules are terms of sort `PreModule` in the module `PREMODULE` below. For more information about tokens, bubbles and metaparsing in Maude see [4, Section 2.7.6]. The module `QID-LIST` is part of the standard library of predefined modules in Maude (see [4, Appendix D]).

```
fmod PREMODULE is
including QID-LIST .
sorts Token  NeTokenList Bubble
      GatherId Attribute AttributeSet
      Decl DeclList PreModule .

subsort Attribute < AttributeSet .
```

```
subsort Decl < DeclList .

*** token
op token : Qid -> Token
            [special
             (id-hook  Bubble (1 1)
              op-hook  qidBaseSymbol (<Qids> : -> Qid))] .

*** bubble
op bubble : QidList -> Bubble
             [special
              (id-hook Bubble (1 -1 ( ) )
               op-hook qidListSymbol
                       (__ : QidList QidList -> QidList)
               op-hook qidBaseSymbol (<Qids> : -> Qid)
               )] .

*** neTokenList
op neTokenList : QidList -> NeTokenList
                  [special(
                   id-hook Bubble (1 -1 ( ) )
                   op-hook qidListSymbol
                           (__ : QidList QidList -> QidList)
                   op-hook qidBaseSymbol (<Qids> : -> Qid)
                   id-hook Exclude (.))] .

*** premodule
op fmod_is_endfm : Token DeclList -> PreModule .
op __ : DeclList DeclList -> DeclList [assoc gather(e E)] .

op including_. : Token -> Decl .
op protecting_. : Token -> Decl .
op sort_. : Token -> Decl .
op sorts_. : NeTokenList -> Decl .
op subsort_<_. : Token Token -> Decl .

ops assoc comm ctor : -> Attribute .
op id:(_) : Bubble -> Attribute .
op prec_ : MachineInt -> Attribute .
op E : -> GatherId  .
op e : -> GatherId  .
op & : -> GatherId  .
op __ : GatherId GatherId -> GatherId [assoc] .
op gather(_) : GatherId -> Attribute  .

op __ : AttributeSet AttributeSet
        -> AttributeSet [assoc comm] .

op op_:->_. : Token Token  -> Decl .
op op_:_->_. : Token NeTokenList Token -> Decl .
```

```
op op_:->_[_]. : Token Token AttributeSet -> Decl .
op op_:_->_[_]. : Token NeTokenList Token AttributeSet
                   -> Decl .
op ops_:->_. : NeTokenList Token -> Decl .
op ops_:_->_. : NeTokenList NeTokenList Token -> Decl .
op ops_:->_[_]. : NeTokenList Token AttributeSet -> Decl .
op ops_:_->_[_]. : NeTokenList NeTokenList Token AttributeSet
                   -> Decl .

op var_:_. : Token Token -> Decl .
op vars_:_. : NeTokenList Token -> Decl .

op mb(_):_. : Bubble Token -> Decl .
op cmb(_):_if(_). : Bubble Token Bubble -> Decl .

op eq(_)=(_). : Bubble Bubble -> Decl .
op ceq(_)=(_)if(_). : Bubble Bubble Bubble -> Decl .

endfm
```

C.2 The Syntax of Admissible Goals

Admissible goals are terms of sort `Goal` in the module `GOAL` below.

```
fmod GOAL is
including PREMODULE .
sort VarQ Formula Goal .

subsort Token < VarQ .

*** *** formula
op none : -> VarQ .
op _;_ : VarQ VarQ -> VarQ [assoc id: none] .

op (_)=(_) : Bubble Bubble -> Formula .
op (_):_ : Bubble Token -> Formula .
op _=>_ : Formula Formula -> Formula .
op _&_ : Formula Formula -> Formula [comm assoc] .
op {_}(_) : VarQ Formula -> Formula .

*** goal
op _|-_ : PreModule Formula -> Goal [none] .

endfm
```

C.3 The Syntax of Admissible Commands

Admissible commands are terms of sort `PreCommand` in the module `PRECOMMAND` below.

```
fmod PRECOMMAND is
```

```
including GOAL .
sorts IntString PreCommand .
subsort MachineInt < IntString .

*** label
op none : -> IntString .
op _._ : IntString IntString -> IntString
          [assoc id: none gather(e E)] .

*** precommand
op goal_. : Goal -> PreCommand .
op ind(_)on_. : IntString Token -> PreCommand .
op vrt(_). : IntString -> PreCommand .
op cns(_). : IntString -> PreCommand .
op imp(_). : IntString -> PreCommand .
op rwr(_). : IntString -> PreCommand .
op lem_to(_). : Formula IntString -> PreCommand .
op simp(_). : IntString -> PreCommand .
op show(_). : IntString -> PreCommand .
endfm
```

References

[1] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, Amsterdam, 1977.

[2] D. Basin, M. Clavel, and J. Meseguer. Rewriting logic as a metalogical framework. Manuscript, 2000.

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

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. Submitted for publication. `http://maude.csl.sri.com`, 2000.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. A Maude tutorial. Presented at *European Joint Conference on Theory and Practice of Software (ETAPS)*, Berlin, Germany, March 25, 2000. `http://maude.csl.sri.com`, 2000.

[6] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *Proc. of the CafeOBJ Symposium'98*, 1998. `http://maude.csl.sri.com`.

[7] M. Clavel and J. Meseguer. Internal strategies in a reflective logic. In B. Gramlich and H. Kirchner, editors, *Proc. CADE-14 – Workshop on Strategies in Automated Deduction*, pages 1–12, 1997.

[8] M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. Submitted for publication, 2000.

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

[10] Manuel Clavel, Francisco Durán, Steven Eker, José Meseguer, and Mark-Oliver Stehr. Maude as a formal meta-tool. In J. Wing and J. Woodcock, editors, *FM'99 — Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1684–1703. Springer-Verlag, 1999.

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

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

[13] Joseph Goguen and José Meseguer. Models and equality for logical programming. In H. Ehrig, G. Levi, R. Kowalski, and U. Montanari, editors, *Proceedings TAPSOFT'87*, volume 250 of *Lecture Notes in Computer Science*, pages 1–22. Springer-Verlag, 1987.

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

[15] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, 1993. To appear in D. Gabbay, ed., *Handbook of Philosophical Logic*, Kluwer Academic Publishers.

[16] N. Martí-Oliet and J. Meseguer. General logics and logical frameworks. In D. Gabbay, editor, *What is a Logical System?*, pages 355–392. Oxford University Press, 1994.

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

[18] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23:733–42, 1976.