Maude ITP 2.0 Tutorial

Joe Hendrix[1], José Meseguer[1] and Ralf Sasse[1]

University of Illinois at Urbana-Champaign, USA.
{jhendrix,meseguer,rsasse}@cs.uiuc.edu

Abstract. The present tutorial describes the main features, commands, and proof techniques supported by the Maude ITP 2.0 inductive theorem prover.

1 Introduction

Inductive theorem proving is one of the most successful verification techniques for proving complex properties about software algorithms. Many different inductive theorem provers have been developed over the years including ACL2 [13], Coq [2], HOL [9], Isabelle [15], Larch [10], the Maude ITP [8,5,3], PVS [16], RRL [12] and SPIKE [1]. These tools support a wide variety of different techniques, logics, and technical approaches.

One common characteristic of inductive theorem provers is that they are almost always *interactive* theorem provers, and proving challenging theorems requires trained user intervention. The advantage of user interaction is that the user can direct the theorem prover to show theorems that cannot be proven by fully automatic techniques. However, it is important from the user's perspective that the theorem prover not require too much input. It is often the case that the user already "knows" the theorem is true, and wants the prover to perform the necessary steps to prove it. However, with current technology the prover will often need direction on problems that appear trivial to the user. Many inductive theorem proof attempts are abandoned when the user decides the theorem prover requires too much involvement.

Techniques to improve and/or reduce user interaction have been a major line of research in inductive theorem proving, and there are at least three different research directions aimed at improving a user's experience with an inductive theorem prover:

1. Better techniques and heuristics for generating *induction schemas*;
2. More powerful and better integrated *automated reasoning* algorithms and decision procedures to eliminate cases generated from the chosen induction scheme.
3. Improvements to the logic and proof assistant to help the user *understand* the current state of the proof and interact with the prover in a more natural way.

Let us first mention different improvements that were made to the earlier version of the Maude ITP, which was developed by Manuel Clavel, as part of Joe Hendrix's thesis [11], and how they relate to these three areas. We then explain in that context the other relevant features of the ITP, and discuss *extensibility*. These improvements are summarized below.

Induction schemas. The ITP has traditionally supported two forms of induction: structural induction and induction over the less than relation $<$ on the natural numbers. This has been extended with an additional induction scheme: *coverset induction* [18]. Coverset induction generates induction schemes by analyzing recursive calls in an operation defined by a complete set of terminating rewrite rule. Adding coverset induction to the ITP required developing a new form of coverset induction for membership equational logic.

In addition to extending coverset induction to membership equational logic, coverset induction was extended in several other directions. First, a command was added to the ITP to define an alternate set of memberships for representing the elements in a sort. These alternate set of memberships can be used by coverset induction to generate more appropriate induction schemes. Second, due to the experience gained with the Powerlist case study [11], we found it helpful to allow coverset induction to take additional patterns other than the one used to generate the induction scheme. These additional patterns are used to further specialize the subcases generated by the theorem prover. If used intelligently, they can help reduce a conjecture to a set of proof obligations that can be proven in a fully automatic way.

Automated Reasoning. The core automated tactics used by the Maude ITP have been improved by developing new commands and extending existing ones. The main new feature that has been added is the ability to prove that a commutative relation in the user's theory is an equivalence relation, and then automatically propagate facts implied by the transitivity of equivalence relation. This feature has lead to much simpler proofs in the Powerlist case study [11]. The other new command is a command `eq-split` which instantiates universally quantified variables in a goal to better match the left-hand sides of rules in the specification. This is essentially a form of constructor splitting that uses the left-hand side of rules in the specification to control the splitting process.

Previous Maude ITP commands have been improved in several ways: (1) the `auto` command has been improved to automatically split conjunctions into multiple subgoals which are then automatically simplified; (2) a new congruence closure algorithm has been added which fixes the spurious Maude metalavel warnings and improves the performance of the old algorithm; (3) commands to enable and disable rules have been added; and (4) the parsing of formulas to extract better inference rules from lemmas and the antecedents of a goal which we are proving has been improved.

User understandability. In order to help the user better understand what to do next, additional commands for displaying information about the current state of the proof and figure out which rules can be applied have been added. The new commands include a command `red` for evaluating arbitrary terms in the current theory, a command `show-hyps` for displaying the current hypotheses, and a command `show-rules with` for identifying all inference rules that contain a given operator. These commands help the user to figure out what existing lemmas one may want to apply as well as devise new lemmas to simplify the current goal.

Extensibility. The ITP system has been improved to make it a better platform for future extension. First of all, the ITP was updated to be compatible with the new Maude 2.3 metalavel. Furthermore, the ITP's source code has been refactored to be more readable and the different data-structures and invariants of the ITP have been documented. Finally, the command parsing component was reimplemented to generate better error messages and become more extensible. One benefit of this work has been that Ralf Sasse was able to port his JAVA+ITP tool [17] to the new version of ITP while making no changes to the source code of the ITP. Previously, JAVA+ITP required Java-specific changes to the ITP source code.

In the rest of this tutorial we discuss the main improvements to the ITP in more detail. We start by introducing some of the existing commands already supported by earlier versions of the ITP [8,5,3] in Section 2. In Section 3, we discuss some useful debugging commands that have been added. In Section 4, we describe the coverset induction feature, based on the theoretical presentation given in Joe Hendrix's thesis [11]. In Section 5, we describe our approach to propagating additional facts when a relation is known to be an equivalence relation.

2 The Maude ITP

The Maude ITP is an experimental interactive tool for proving properties of the initial algebra $T_{\mathcal{E}}$ of a membership equational logic (MEL) [14] specification $\mathcal{E}$ written in Maude [4]. The ITP has been written entirely in Maude, and is in fact an *executable* specification in MEL of the formal inference system that it implements. The ITP inference system treats MEL specifications as *data* — for example, the ITP command `imp` adds the hypotheses of the current goal as rules to the current goal's associated theory. This makes the ITP a *reflective* design, in which Maude equational specifications become data at the metalevel. Indeed, the fact that membership equational logic is a reflective logic [7] and that Maude efficiently supports reflective MEL computations is systematically exploited in this tool. A similar reflective design has been adopted to develop other formal tools in Maude [6]. Using reflection to implement the ITP tool has one important additional advantage, namely, the ease to rapidly extend it by integrating other tools implemented in Maude using reflection, as it is the case of the sufficient completeness checker, see [11].

An ITP session begins with the user providing a Maude theory $\mathcal{E}$ whose equations and rules have been oriented into a conditional rewrite membership system $\mathcal{R}$ along with a first-order MEL formula ϕ which the user wishes to prove holds in the initial algebra $T_{\mathcal{E}}$. The ITP is *interactive* and requires user input to discharge the formula ϕ. At each point in an ITP session, the ITP maintains the sequence of *goals* remaining to be proved. Each goal has an associated *formula* and an associated *theory* which extends the original theory with lemmas and assumed hypotheses introduced by the user. Once all the goals are discharged, then the original conjecture ϕ has been proven to be true.[1]

The ITP offers many different commands available to the user to aid in the task of proving the remaining goals. A tutorial of the older version of the ITP can be found in [8], and legacy code is preserved as far as possible. This is not a complete reference, but a tutorial, and we introduce those commands that a user will most likely need to use.

As the ITP is using Maude's metalevel, all commands have to be introduced within parentheses and are finished with a blank space and a period before the final right parenthesis is added. So whenever a command is shown here, you will need to wrap it accordingly before passing it to Maude. To give the command `command` to the ITP you have to actually write `(command .)` instead.

Before going into detail, let us mention that a list of all available commands can be found in Appendix A. To start an actual proof, the appropriate version of Maude (at least 2.3) needs to be running. Then the ITP-tool needs to be loaded, followed by the module about which we want to prove properties, given in regular Maude syntax. After that, the commands

```
select ITP-TOOL .
loop init-itp .
```

need to be given and the actual goal we want to prove needs to be loaded. The syntax for entering a goal is

`(goal` ***`goalname`*** `:` ***`modulename`*** `|-` ***`formula`*** `.)`

where the formula needs to have no free variables. Universal quantification uses `A{`***`vars-list`***`}` ***`formula`*** as the syntax, while existential quantification is written as `E{`***`vars-list`***`}` ***`formula`***. After entering the goal, the user only needs to give commands to discharge the goal. The available commands are described next.

ind and **ind***. The `ind` and `ind*` commands each take a variable x that is universally quantified in the current formula, and perform structural induction on x. The corresponding syntax is:

`(ind on` *`x`* `.)`   `(ind* on` *`x`* `.)`

The current formula should have the form $(\forall x)\ x : s \implies \phi$, or $(\forall x : s)\phi$, and the structural induction scheme is obtained from the constructor memberships from

[1] This has to be understood to be a *relative claim*, since certain proof obligations that are implicit in the ITP must be discharged with the help of other tools in the Maude formal tool environment for a full proof to be achieved. For example, structural induction uses the declared constructors to generate the induction scheme. But such a scheme would be unsound without a proof of sufficient completeness for the constructors, for which the SCC tool can be used [11].

the original specification. The difference between `ind` and `ind*` is that `ind*` will automatically call the `auto` command (explained later) for simplifying each subgoal, while `ind` will output the unsimplified subgoals. In many proof attempts, `ind*` will completely eliminate all the subgoals, so a single `ind*` command can often discharge the current goal without generating any new subgoals.

cnj. The `cnj` command splits a conjunction $A \& B$ into two subgoals, A and B.

cns. The `cns` command performs universal quantifier elimination when the current formula has the form $(\forall Y)\, \phi$. When this command is issued, the ITP introduces a fresh constant c_y for each variable $y \in Y$. This constant is added to the current module, each variable y appearing in the formula ϕ is replaced by c_y, and the outermost quantifier is dropped from the current goal.

e-inst. The `e-inst` command is the counterpart to `cns` for performing existential quantifier elimination. When the current formula has the form $(\exists Y)\, \phi$ and the user issues the command

```
(e-inst with sub .)
```

where *sub* is a substitution $\theta : Y \rightarrow T_{\Sigma}(X)$, the ITP replaces the current formula $(\exists Y)\, \phi$ with the formula $\phi\theta$. Note that this is of course sound, but may fail to be complete in case of an inappropriate instantiation. That is, it might be impossible to do the proof (or the statement may even become false) after instantiation, even though the original formula holds and could be proved with a different instantiation.

imp. The `imp` command can be used when the current formula is an implication $\phi \implies \psi$ to introduce additional inference rules into the current module. When invoked, the ITP parses ϕ to extract one or more (possibly conditional) equations to add to the module. Each such equation (oriented as a rule from left to right) is labeled as an hypothesis, numbered, and added to the current module. If the antecedent ϕ contains formulas that cannot be added to the module, then each one is added to the current goal as an auxiliary labeled formula which may be instantiated later by the user. After extracting and labeling the equations in ϕ the conditions in ϕ are eliminated and the current formula is replaced with ψ.

lem. The `lem` command is used to introduce lemmas to help prove the current formula. The syntax of the command is

```
(lem name : formula .)
```

When this command is issued, a new goal with the given name is added to the current list of goals with the given formula. In addition, the given formula is parsed with the same algorithm used by `imp` command, and one or more new (possibly conditional) sentences usable as rules may be added to the current module. If the formula cannot be parsed as a conjunction of admissible sentences usable as rules, it is added as a formula which may be used with the `a-inst` command described below.

a-inst. The `a-inst` command is primarily used for instantiating universally quantified labeled formulas added by `imp`, or by `lem` when introducing lemmas. Usually, these are formulas that cannot be executed, because the condition contains extra variables. The syntax of `a-inst` is

(a-inst *name* with *sub* .)

where *name* is a labeled formula $(\forall Y)\ \phi$ appearing in the current goal, and *sub* is a substitution $\theta : Y \to T_{\Sigma}(X)$. After this command is issued, the ITP parses $\phi\theta$ to extract new subformulas usable as rules, and adds them to the current module.

auto. The `auto` command is perhaps the most commonly used ITP command. It attempts to automatically discharge the current goal through a variety of tactics. If the current goal is a universally quantified formula, it uses the `cns` command to perform quantifier elimination. If the current goal is an implication, it assumes the hypotheses by using the `imp` command. Otherwise, the `auto` command rewrite all the terms in the current formula and the current hypotheses. If the terms in an equation are reduced to the same term $t = t$, then the equation is replaced with `true`. The current goal is eliminated if it reduces to `true`, or if one of the equations in the hypotheses is reduced to `true` $=$ `false`.

The `auto` command has been extended in two ways. The first is to perform equivalence propagation as described later in Section 5. The second extension occurs when the current formula is a conjunction $\phi_1 \wedge \cdots \wedge \phi_n$ with $n \geq 2$. Then, the new `auto` command will automatically split the conjunction into n separate goals with the formulas $\phi_1, \ldots, \phi_n$ respectively. It will then invoke the `auto` command on each subgoal. Previously the `auto` command would halt on conjunctions.

split. The `split on` *`bool-formula`* command allows splitting the current goal into two subgoals, given a formula with a boolean result value. In one of of the subgoals the given *`bool-formula`* is assumed to be true, in the other it is assumed to be false.

In this section we have introduced several of the commands which a user will most frequently need. In the remaining sections we will discuss several other commands that have been added to the ITP in version 2.0 in order to improve its induction features, core reasoning capabilities, and user interface.

Let us also show an example with the commands introduced so far.

Example 1. For a Maude module representing lists of natural numbers by means of a `cons` operator, where `nil` is the empty list, we would like to show that the standard `append` operator is associative.

```
fmod LIST-of-NAT is protecting NAT .
  sort List .
  op nil : -> List [ctor] .
  op cons : Nat List -> List [ctor] .
  op append : List List -> List .
  var N : Nat .
  vars L L' : List .
  eq append(nil, L) = L .
  eq append(cons(N, L), L') = cons(N, append(L, L')) .
endfm
```

Based on the above module we can introduce the desired goal to the ITP with this command:

```
select ITP-TOOL .
loop init-itp .
(goal list-append-assoc : LIST-of-NAT
  |- A{L:List ; L':List ; L'':List}
     ((append(append(L,L'),L'')) = (append(L,append(L',L'')))) .)
```

First we can apply structural induction on `L` with the command `(ind on L .)` to get two subgoals:

```
=================================
label-sel: list-append-assoc@1.0
=================================
A{L'':List ; L':List} append(append(nil,L':List),L'':List)
  = append(nil,append(L':List,L'':List))
```

The first subgoal can be discharged by `(auto .)` obviously, as after normalizing both sides with the equation for `append` of `nil` the two sides are syntactically equal.

```
=================================
label: list-append-assoc@2.0
=================================
A{V0#0:Nat ; V0#1:List}(A{L'':List ; L':List}
  append(append(V0#1:List,L':List),L'':List)
   = append(V0#1:List,append(L':List,L'':List)))
 ==>(A{L'':List ; L':List}
  append(append(cons(V0#0:Nat,V0#1:List),L':List),L'':List)
   = append(cons(V0#0:Nat,V0#1:List),append(L':List,L'':List)))
```

The second subgoal looks a little more complicated, but applying the rules for appending a natural number twice in the left-hand side and once in the right-hand side yields two terms that are exactly the terms of the induction hypothesis, under a wrapping `cons` application, thus the hypothesis can be applied and both terms in the equality of the conclusion become syntactically equal. This is also done with a single application of `(auto .)`. The observant reader will thus have noticed, that a single application of `(ind* on L .)` would have taken care of the original goal altogether.

3 Interface Commands

In this section we describe several commands which were added to the ITP for developing a proof strategy, and debugging failed proofs.

Enable/Disable. The `enable` and `disable` commands control the executability of the different rewrite rules and memberships in the current goal. The rules can either come from hypotheses in the current module, lemmas that were previously added, or labeled equations in the original user's module.

```
(enable rule-name .)      (disable rule-name .)
```

If `enable` is called with the name of a rule in the module labeled with the attribute `nonexec`, it will discard the `nonexec` attribute, thus enabling the rule during rewriting. Conversely, when the `disable` command is given for a rule that is not labeled with the `nonexec` attribute, it will add the `nonexec` attribute to the rule, thus disabling it when rewriting is used to simplify goals. These commands can be used in debugging to help identify non-terminating hypotheses or lemmas. They also can be used for *information hiding*. It is often useful to prove lemmas that state the essential properties of an operation, and then disable the operation's definition.

Reduction. It can be difficult to remember all the lemmas and hypotheses added to a module, and sometimes rules may fail to apply because a condition cannot be resolved. Unfortunately, there is no automatic way to fix the second problem, but to aid the debugging process, the command

```
(red term .)
```

has been added. This command computes the canonical form and least sort of an arbitrary term in the current module.

Showing rules. A large part of the success of inductive theorem provers stems from a user's ability to construct a set of terminating rules that yield unique normal forms for terms appearing in the current goal. In the ITP, the rules depend not only on the definitions in the user's module and lemmas, but also on the hypotheses added to the current proof attempt. Unlike the lemmas, the hypotheses are usually different for each subgoal, and so it is often helpful to see the current hypotheses assumed in the current goal. To do this, the command `(show-hyps .)` has been added to show the current hypotheses.

In addition to the hypotheses, it is often useful to see *all* of the rules related to a given symbol appearing in the current goal. Although the existing `show-all` command will display all of the rules, it can be tedious to sort through them in larger proofs to see the rules that are currently relevant. For more targeted searches, the command

```
(show-rules with op .)
```

has been added. This command will display the equations and memberships whose left-hand side references the operator *op*.

Selecting a diffferent goal. Sometimes it may be beneficial to check a goal different from the one selected by default. For that, the command

```
(sel goal-name .)
```

will make the goal with the given name the current proof goal. This command already existed in prior versions and turns out to be very useful.

4 Coverset Induction in the Maude ITP

For the theory behind coverset induction, see Joe Hendrix's thesis [11].

The Maude ITP now has two commands `cov` and `cov*` which apply coverset induction when invoked by the user. At the moment these commands should only be invoked if the given module in which we are reasoning does not make use of equational attributes in operators, such as associativity, commutativity and identity, except for commutativity of equivalence relations as explained in Section 5. A future ITP version will eliminate this restriction. The difference between the two commands is that `cov*` will automatically simplify all of the subgoals generated by coverset induction with the `auto` command, while `cov` will leave them unchanged. Each command takes the pattern as an argument with the syntax:

```
(cov on pattern)      (cov* on pattern)
```

where *pattern* is a term whose variables are universally quantified in the current formula.

One useful feature of coverset induction is that, in addition to generating potentially useful induction hypotheses, it specializes terms appearing in the current problem to match additional rules in the specification. This allows them to be simplified by rewriting. Splitting based on constructors is called *constructor splitting*, and has been quite useful in the powerlist case study [11]. The ITP already offers a command `ctor-term-split` to do this, but the command only replaces a single variable with its constructor memberships, and does not explicitly attempt to match a term against the left-hand sides of equations. As a consequence, `ctor-term-split` often had to be invoked several times to achieve the required matching. In contrast, a single coverset induction command would have done the job. Coverset induction also potentially introduced induction hypotheses even if they were not necessary. For this reason the commands `eq-split` and `eq-split*` were added. These commands essentially perform coverset induction, but do not add the induction hypotheses.

```
(eq-split on pattern .)      (eq-split* on pattern .)
```

The difference between these two commands is that `eq-split*` invokes the `auto` command to attempt to automatically discharge each subgoal, while `eq-split` does not modify the generated goals.

For more details on how most demanded variables and a subsumption test improve practical performance, see Joe Hendrix's thesis [11]. Allowing substitutions in subgoals to be further specialized by *additional patterns* is discussed next, followed by allowing users to define *alternative constructors* for sorts in the specification.

4.1 Additional Patterns

As shown in Joe Hendrix's thesis [11], many of the lemmas in the powerlist case study are discharged with a single `cov*` command. However, for many of the lemmas where this failed, they could be discharged automatically if one were to perform additional constructor splitting on terms appearing in the subgoal. This process requires multiple commands. To help the user, this process has been automated by introducing two coverset induction commands that take additional patterns, called *split patterns*, which are used for splitting the subgoals. They are not used to generate the induction hypothesis. The two commands have the following syntax:

```
(cov-split  on pattern split split-patterns)
(cov-split* on pattern split split-patterns)
```

where *pattern* is the term used for coverset induction and *split-patterns* is a semicolon-separated list of terms.

The `cov-split` (resp. `cov-split*`) commands can be thought of as performing coverset induction with the given pattern, and then `eq-split` (resp. `eq-split*`) on the induction cases. The experience gained with the powerlist case study has shown that virtually all of the lemmas involving coverset induction and `eq-split` could be solved in a single `cov-split*` command.

4.2 Alternative Constructors

The fourth and final extension to coverset induction that has been added to the ITP is the ability to define alternative constructor declarations with the command

$$\texttt{(ctor-def } \mathit{name} : \texttt{A}\{x : s\}$$
$$(\texttt{E}\{Y_1\}\, t_1 = x \,\texttt{\&}\, \mathit{cond}_1)\ \texttt{V}\ \ldots\ \texttt{V}\ (\texttt{E}\{Y_n\}\, t_n = x \,\texttt{\&}\, \mathit{cond}_n)\ .)$$

where each formula cond_i is a (possibly empty) conjunction of equations and memberships.

After giving this command to the ITP, the ITP creates a subgoal which requires the user to prove the given formula, and it adds a set of alternative memberships

$$M_{name} = \{\, t_1 : s \textbf{ if } \mathit{cond}_1 \qquad \ldots \qquad t_n : s \textbf{ if } \mathit{cond}_n \,\}$$

to the current goal. These memberships can then be used in lieu of the normal constructor memberships with sort s during the narrowing phase of coverset induction. In order to specify which alternative memberships to use, the following four commands have been added:

```
(cov using names on pattern)
(cov* using names on pattern)
(cov-split  using names on pattern split split-patterns)
(cov-split* using names on pattern split split-patterns)
```

where *names* is a semicolon-separated list with the names of constructor definition names. Each name in the list must be associated to memberships for a distinct sort, and when a name is provided the memberships in $\mathcal{E}$ for that name are replaced with the memberships M_{name} for the purposes of instantiating a variable using the inference rule given in [11].

Alternate constructors are used in the powerlist case study in several places. A key property of powerlists is that each powerlist with more than one element can be represented as either the *concatenation* $P \mid Q$ of two powerlists or the *interleaving* $P \times Q$ of two powerlists. In the Maude specification of powerlists, given in [11], a membership with $\mid$ as the main constructor is used, but an alternate set of constructors with $\times$ was proved to be correct, too. For operations that are most naturally defined using $\times$, the alternate constructors were used when performing coverset induction.

In some cases, one may want to make the alternative constructor definitions the default constructors. This can be done with the command

```
(set-default-ctor name.)
```

After issuing this command the memberships with the given name will be used for their associated sort whenever constructor narrowing occurs. The default set of memberships for a sort can be used by calling `set-default-ctor` with the name of the sort.

5 Equivalence Propagation

In addition to coverset induction, the ITP has been extended with specialized support for *equivalence relations*. As membership equational logic is functional and does not support relations other than the sort predicates, a *relation* in the context of this section is a binary function whose output kind is the kind used by the built-in Boolean type. A relation p is an *equivalence relation* over a sort $s \in S$ for our purposes if it is labeled with the `comm` attribute marking the symbol as commutative and satisfies the following two properties:

$$T_{\mathcal{E}} \models (\forall x : s)\ p(x, x) = \texttt{true}$$
$$T_{\mathcal{E}} \models (\forall x : s, y : s, z : s)\ p(x, y) = \texttt{true} \wedge p(y, z) = \texttt{true} \implies p(x, z) = \texttt{true}$$

where `true` refers to the operator for true in the predefined `BOOL` module and $(\forall x : s)\ \phi$ is syntactic sugar for the formula $(\forall x)\ x : s \implies \phi$.

Equivalence relations benefit from specialized automated reasoning support, because ordinary rewriting cannot deal with extra variable y in the condition of the transitivity axiom

$$p(x,z) \textbf{ if } p(x,y) \wedge p(y,z).$$

The solution to this has been to extend the ITP with two commands: a command `defequiv` for defining equivalence relations, and a command `equiv-propagate` for propagating facts implied by transitivity. In addition, the built-in *auto* command has been extended to also perform equivalence propagation in addition to its other tactics.

To indicate that a given operation is an equivalence relation, the following command has been added to the ITP:

```
(defequiv p on sort .)
```

The operation p must be labeled with the commutativity attribute. When this command is issued to the ITP, the ITP generates two subgoals — one for each equation that an equivalence relation must satisfy, and then records in the original goal that the operation p is an equivalence relation for arguments with a sort *sort*.

Once one or more equivalence relations are added using `defequiv`, equivalence propagation will automatically occur when the user calls the `auto` command, which automatically applies several tactics including rewriting, equivalence propagation, and hypothesis simplification to resolve the formula. In addition, the user may request equivalence propagation to occur with the command `(equiv-propagate .)`. When equivalence propagation is used with either command, for each predicate p that is an equivalence relation on s, the following rule is applied until completion.

$$\frac{p(t,u) = \texttt{true}, p(u,v) = \texttt{true} \in \mathcal{E} \text{ s.t. } p(t,v) \downarrow_{\mathcal{E}} \neq \texttt{true}}{\mathcal{E}' := \mathcal{E} \uplus \{\, p(t,v) = \texttt{true} \,\}}$$

where $\mathcal{E}$ denotes the theory containing the current module and any hypotheses assumed in the current goal, and $p(t,v) \downarrow_{\mathcal{E}}$ denotes the term obtained by rewriting $p(t,v)$ with the equations in $\mathcal{E}$ oriented as rules. After applying the rule, the current goal's theory $\mathcal{E}$ is replaced with $\mathcal{E}'$, and then the process is repeated until either: (1) the rule can no longer be applied, or (2) a conflict is detected because $p(t,u) = \texttt{true}$, $p(u,v) = \texttt{true}$ and $p(t,v) \downarrow_{\mathcal{E}} = \texttt{false}$. If a conflict is detected, then the current hypotheses are unsatisfiable, and the current goal is immediately discarded.

6 Conclusions

We have presented the most common commands for using the ITP. This is not a full reference, but hopefully should be enough for a user familiar with the basics of the Maude language [4] to use the ITP effectively. For the more advanced features discussed in Sections 4 and 5, the powerlist case study presented in [11] can be a very good source of examples and insight about their use.

Acknowledgements. The current 2.0 version of the Maude ITP was developed by Joe Hendrix [11] as a refactored implementation of the ITP developed by Manuel Clavel. It adds a number of new features such as coverset induction, alternative constructors, equivalence propagation, and new proof automation commands. It has been applied to a large and substantial case study, namely properties of powerlist functions [11]. We gratefully acknowledge the great help provided by Manuel Clavel in developing the earlier ITP functionality now incorporated in the 2.0 version, and in providing in-depth accounts of his implementation. The theoretical work on which the coverset induction feature is based is joint work by Manuel Clavel, Joe Hendrix, Deepak Kapur, and José Meseguer. We thank Manuel Clavel and Deepak Kapur for their contribution.

References

1. N. Berregeb, A. Bouhoula, and M. Rusinowitch. SPIKE-AC: A system for proofs by induction in associative-commutative theories. In H. Ganzinger, editor, *Proc. of RTA-96*, volume 1103 of *Lecture Notes in Computer Science*, pages 428–431. Springer, 1996.
2. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer, 2004.
3. M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications.* CSLI Publications, 2000.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude*, volume 4350 of *Lecture Notes in Computer Science.* Springer, 2007.
5. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *CAFE: An Industrial-Strength Algebraic Formal Method*. Elsevier, 2000. `http://maude.cs.uiuc.edu`.
6. M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In J. M. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1684–1703. Springer, 1999.
7. M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, horn logic with equality, and rewriting logic. *Electronic Notes Theoretical Computer Science*, 71, 2002.
8. M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science*, 12(11):1618–1650, 2006.
9. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.
10. J. V. Guttag, J. J. Horning, S. J. Garland, and K. Jones. *Larch: Languages and Tools for Formal Specification.* Springer, 1993.
11. J. Hendrix. *Decision Procedures for Equationally Based Reasoning.* PhD thesis, University of Illinois at Urbana-Champaign, 2008.

12. D. Kapur and H. Zhang. An overview of Rewrite Rule Laboratory (RRL). *Journal of Computer and Mathematics with Applications*, 29(2):91–114, 1995.
13. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, 2000.
14. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, pages 18–61. Springer LNCS 1376, 1998.
15. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
16. S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Proc. of CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.
17. R. Sasse and J. Meseguer. Java+ITP: A verification tool based on hoare logic and algebraic semantics. *Electronic Notes Theoretical Computer Science*, 176(4):29–46, 2007.
18. H. Zhang, D. Kapur, and M. S. Krishnamoorthy. A mechanizable induction principle for equational specifications. In E. Lusk and R. Overbeek, editors, *Proc. of CADE-9*, volume 310 of *Lecture Notes in Computer Science*, pages 162–181. Springer, 1988.

A List of ITP commands

- `a-inst` *name* `with` *sub*
- `auto`
- `cnj`
- `cns`
- `cov on` *pattern*
- `cov* on` *pattern*
- `cov-split on` *pattern* `split` *split-patterns*
- `cov-split* on` *pattern* `split` *split-patterns*
- `cov using` *names* `on` *pattern*
- `cov* using` *names* `on` *pattern*
- `cov-split using` *names* `on` *pattern* `split` *split-patterns*
- `cov-split* using` *names* `on` *pattern* `split` *split-patterns*
- `ctor-def` *name* `: A{`x `:` s`} (E{`Y_1`}` t_1 `=` x `&` $cond_1$`) V ... V (E{`Y_n`}` t_n`=`x `&` $cond_n$`)`
- `ctor-term-split` *var*
- `def-equiv` *p* `on` *sort*
- `disable` *rule-name*
- `e-inst with` *sub*
- `enable` *rule-name*
- `equiv-propagate`
- `eq-split on` *pattern*
- `eq-split* on` *pattern*
- `imp`
- `ind on` x

- `ind* on` *x*
- `lem` *name* `:` *formula*
- `red` *term*
- `sel` *goal-name*
- `set-default-constructor` *name*
- `show-all`
- `show-hyps`
- `show-rules with` *op*
- `split on` *bool-formula*