Program Verification: Lecture 12

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

Mathematical Proof of Associativity of Addition

We want to prove that the addition operation in the module

```
fmod NATURAL is
  sort Natural .
  op 0 : -> Natural [ctor] .
  op s : Natural -> Natural [ctor] .
  op _+_ : Natural Natural -> Natural .
  vars N M : Natural .
  eq N + 0 = N .
  eq N + s(M) = s(N + M) .
endfm
```

satisfies the associativity property,

$$(\forall \mathtt{N}, \mathtt{M}, \mathtt{L})\ \mathtt{N + (M + L)} = \mathtt{(N + M) + L}.$$

Mathematical Proof of Associativity of Addition (II)

We can prove the property by induction on `L`. That is, we prove it for $L = 0$ (base case) and then assuming that it holds for `L`, we prove it for `s(L)` (induction step).

Base Case: We need to show,

$$(\forall N, M)\ N + (M + 0) = (N + M) + 0.$$

We can do this trivially, by simplification with the equation

```
eq N + 0 = N .
```

Mathematical Proof of Associativity of Addition (II)

Induction Step: We think of L as a generic constant (typically written n in textbooks) and assume that the associativity equation (induction hypothesis (IH))

$$(\forall \texttt{N}, \texttt{M})\ \texttt{N + (M + L)} = \texttt{(N + M) + L}.$$

holds for that constant. Then we try to prove the equation,

$$(\forall \texttt{N}, \texttt{M})\ \texttt{N + (M + s(L))} = \texttt{(N + M) + s(L)}.$$

using the induction hypothesis. Again, we can do this by simplification with the equations E in NAT, and the induction hypothesis IH equation, since we have,

Mathematical Proof of Associativity of Addition (III)

$$\texttt{N + (M + s(L))} \longrightarrow_E \texttt{N + s(M + L)}$$

$$\longrightarrow_E \texttt{s(N + (M + L))} \longrightarrow_{IH} \texttt{s((N + M) + L).}$$

and

$$\texttt{(N + M) + s(L)} \longrightarrow_E \texttt{s((N + M) + L).}$$

q.e.d

Machine-Assisted Proof with Maude's ITP

Maude's ITP is an inductive theorem prover supporting proof by induction in Maude functional modules. It is a program written entirely in Maude by Manuel Clavel and Joe Hendix in which one can:

- load in Maude the functional module or modules one wants to reason about
- load the file `itp-tool.maude` and then type `loop init-itp .`
- enter named goal to be proved by the ITP enclosed in parentheses using the `goal` command.
- give commands, corresponding to proof steps, to prove that property, also enclosed in parentheses

Machine-Assisted Proof with Maude's ITP (II)

For example, suppose that we want to automatically prove the associativity of addition. We first load into Maude the module, say,

```
fmod NATURAL is
  sort Natural .
  op 0 : -> Natural [ctor] .
  op s : Natural -> Natural [ctor] .
  op _+_ : Natural Natural -> Natural .
  vars N M : Natural .
  eq N + 0 = N .
  eq N + s(M) = s(N + M) .
endfm
```

Then we load `itp-tool.maude` and type `loop init-itp .`

Machine-Assisted Proof with Maude's ITP (III)

We then enter our associativity goal by giving it a name (`assoc`), mentioning the module in which it should be proved (`NATURAL`) and making explicit the universal quantification with the letter `A` and curly brackets notation. Note the required use of "on-the-fly" variables; and the generous use of parentheses to help the ITP parser.

```
(goal assoc : NATURAL |- A{N:Natural ; M:Natural ;  L:Natural}
   ((N + (M + L)) = ((N + M) + L)) .)
```

The ITP then echoes, giving this goal an additional label ending (`@0`) to help the user keep track of where he/she is as the proof process unfolds and other (sub-)goals are generated.

```
=================================
label-sel: assoc@0
=================================
A{N:Natural ; M:Natural ; L:Natural}
N:Natural +(M:Natural + L:Natural) = (N:Natural + M:Natural)+ L:Natural
+++++++++++++++++++++++++++++++++
```

Machine-Assisted Proof with Maude's ITP (IV)

We can then try to prove goal `assoc@0` by induction on `L:Natural` by giving the command `(ind on L:Natural .)` The tool then generates two subgoals (one for the base case, and another for the induction step). The current, selected goal is labeled with `-sel`

```
=================================
label-sel: assoc@1.0
=================================
A{N:Natural ; M:Natural} N:Natural +(M:Natural + 0) =(N:Natural + M:Natural)+ 0

=================================
label: assoc@2.0
=================================
A{V0#0:Natural}
(A{N:Natural ; M:Natural}
N:Natural +(M:Natural + V0#0:Natural) =(N:Natural + M:Natural)+ V0#0:Natural)
```

```
 ==>
(A{N:Natural ; M:Natural}
N:Natural +(M:Natural + s(V0#0:Natural)) =(N:Natural + M:Natural)+ s(V0#0:Natura
+++++++++++++++++++++++++++++++++
```

Machine-Assisted Proof with Maude's ITP (V)

We can then try prove the above "base case" subgoal by using the ITP's `auto` tactic that —after turning the variables into constants by the constants lemma (more on this later) and doing implication elimination if necessary— tries to simplify the goal by applying equations in the module, until hopefully reaching an identity. This tactic succeeds, leaving the second goal.

```
Maude> (auto .)

=================================
label-sel: assoc@2.0
=================================
A{V0#0:Natural}
(A{N:Natural ; M:Natural}
N:Natural +(M:Natural + V0#0:Natural) =(N:Natural + M:Natural)+ V0#0:Natural)
```

```
==>
(A{N:Natural ; M:Natural}
N:Natural +(M:Natural + s(V0#0:Natural)) =(N:Natural + M:Natural)+ s(V0#0:Natura
+++++++++++++++++++++++++++++++++
```

Machine-Assisted Proof with Maude's ITP (VI)

We can likewise apply the `auto` tactic to the second goal, thus proving the associativity theorem.

```
Maude> (auto .)

q.e.d

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

Note that, in this case, both the constants lemma and implication elimination had to be invoked by `auto` before being able to simplify both sides of the conclusion using the induction hypothesis.

List Induction

So far, we have only used natural number induction. What about induction on other data structures? For example, what about list induction? Consider, for example, the following module defining a list `append` operator in terms of a list "cons" operator `_:_` for lists of natural numbers importing the `NAT` predefined module.

```
fmod MY-LIST is protecting NAT .
sort List .
op nil : -> List [ctor] .
op _:_ : Nat List -> List [ctor] .
op append : List List -> List .
vars N M : Nat .
vars L L1 L2 L3 : List .
eq append(nil, L) = L .
eq append(N : L1, L2) = (N : append(L1, L2)) .
endfm
```

List Induction (II)

The `nil` constant and the "cons" operator `_:_` are constructors that play a role analogous to zero and successor in `NAT`, and list "append" is the analogous of number addition.

In fact, it is also associative, that is, the above module satisfies the property,

$$(\forall \mathtt{L1}, \mathtt{L2}, \mathtt{L3})\ \mathtt{append(append(L1,L2),L3)} = \mathtt{append(L1,append(L2,L3))}.$$

List Induction (III)

The same scheme of proof used to prove associativity of addition can be used here as well, changing zero by `nil`, and successor by the "cons" operator `_:_`.

That is, if we want to do induction on `L1`, we must prove the base case for `nil`,

$$(\forall \mathtt{L2}, \mathtt{L3})\ \mathtt{append(append(nil,L2),L3)} = \mathtt{append(nil,append(L2,L3))}.$$

which follows trivially by simplification with the equation

```
eq append(nil, L) = L .
```

List Induction (IV)

And then we must prove the induction step by assuming that, considering `L1` as a generic list constant, we have the induction hypothesis equation,

$$(\forall \mathtt{L2}, \mathtt{L3})\ \mathtt{append(append(L1,L2),L3)} = \mathtt{append(L1,append(L2,L3))}.$$

that we try to use, along with the equations in the `MY-LIST` module, to prove by simplification the equation

$$(\forall \mathtt{L2}, \mathtt{L3})\ \mathtt{append(append((N\ :\ L1),L2),L3)} = \mathtt{append((N\ :\ L1),append(L2,L3))}.$$

where `N` is a generic natural constant,

List Induction (V)

All this can again be done by hand, and it works. But it can be automated using the Maude ITP prover by:

- an induction step on `L`, which generates two subgoals, followed by

- `auto` steps for the subgoals (which succeed)

After initializing the ITP and entering the `MY-LIST` module, we enter the main goal to the ITP. The screenshot shows the result of the `ind` step followed by the two `auto` steps, which complete the proof.

List Induction (VI)

```
Maude> (goal append-assoc :
    MY-LIST |- A{L1:List ; L2:List ; L3:List}
                ((append(L1, append(L2, L3)))
                 = (append(append(L1, L2), L3))) .)

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

Maude> (ind on L1:List .)

=================================
label-sel: append-assoc@1.0
=================================
A{L2:List ; L3:List}
```

```
append(nil,append(L2:List,L3:List)) = append(append(nil,L2:List),L3:List)

=================================
label: append-assoc@2.0
=================================
A{V0#0:Nat ; V0#1:List}
(A{L2:List ; L3:List} append(V0#1:List,append(L2:List,L3:List)) =
append(append(V0#1:List,L2:List), L3:List))
==>
(A{L2:List ; L3:List} append(V0#0:Nat :
V0#1:List,append(L2:List,L3:List)) =
append(append(V0#0:Nat : V0#1:List,L2:List),L3:List))
+++++++++++++++++++++++++++++++++

Maude> (auto .)

=================================
label-sel: append-assoc@2.0
=================================
A{V0#0:Nat ; V0#1:List}
(A{L2:List ; L3:List} append(V0#1:List,append(L2:List,L3:List)) =
```

```
append(append(V0#1:List,L2:List), L3:List))
==>
(A{L2:List ; L3:List}
append(V0#0:Nat : V0#1:List,append(L2:List,L3:List)) =
append(append(V0#0:Nat : V0#1:List,L2:List),L3:List))
+++++++++++++++++++++++++++++++++

Maude> (auto .)

q.e.d

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

Using Lemmas

Life is not always as easy as proving associativity of addition or of list append. Often, attempts at simplification using the `auto` tactic do not succeed. However, they suggest lemmas to be proved. Consider the following goal of proving commutativity of addition in our `NATURAL` module:

```
Maude> (goal comm : NATURAL |- A{N:Natural ; M:Natural}
         ((N + M) = (M + N)) .)

=================================
label-sel: comm@0
=================================
A{N:Natural ; M:Natural} N:Natural + M:Natural = M:Natural + N:Natural
+++++++++++++++++++++++++++++++++
```

Using Lemmas (II)

We can try to prove it by induction on `M:Nat`

```
Maude> (ind on M:Natural .)



==================================
label-sel: comm@1.0
==================================
A{N:Natural} N:Natural + 0 = 0 + N:Natural

==================================
label: comm@2.0
==================================
A{V0#0:Natural}(A{N:Natural}
N:Natural + V0#0:Natural = V0#0:Natural + N:Natural)
==>
(A{N:Natural} N:Natural + s(V0#0:Natural) = s(V0#0:Natural)+ N:Natural)
++++++++++++++++++++++++++++++++++
```

Using Lemmas (III)

When we apply the `auto` tactic to this first goal we get,

```
Maude> (auto .)

=================================
label-sel: comm@1.0
=================================
N*Natural = 0 + N*Natural
+++++++++++++++++++++++++++++++++
```

Using Lemmas (IV)

What we can do is to assume the unsimplified equation yielded by `auto` as a lemma in the proof of our main goal. We can do this by giving this lemma a label and adding it to the module of goal `comm@1.0` as follows:

```
Maude> (lem 0-comm : A{N:Natural}((0 + N) = (N)) .)

=================================
label-sel: 0-comm@0
=================================
A{N:Natural} 0 + N:Natural = N:Natural

=================================
label: comm@1.0
=================================
N*Natural = 0 + N*Natural
```

```
=================================
label: comm@2.0
=================================
A{V0#0:Natural}(A{N:Natural}
N:Natural + V0#0:Natural = V0#0:Natural + N:Natural)
==>
(A{N:Natural} N:Natural + s(V0#0:Natural) = s(V0#0:Natural)+ N:Natural)
+++++++++++++++++++++++++++++++++
```

Using Lemmas (V)

Adding this lemma creates a new goal `0-comm@0`, that is, a new proof obligation that we need to discharge. We can do so by proving the lemma by induction on `N:Natural`, using the `auto` tactic to eliminate the two generated subgoals, which brings us back to the original unproved subgoal:

```
Maude> (ind on N:Natural .)

=================================
label-sel: 0-comm@1.0
=================================
0 + 0 = 0

=================================
label: 0-comm@2.0
=================================
A{V1#0:Natural}
```

```
0 + V1#0:Natural = V1#0:Natural
==>
0 + s(V1#0:Natural) = s(V1#0:Natural)

=================================
label: comm@1.0
=================================
N*Natural = 0 + N*Natural

=================================
label: comm@2.0
=================================
A{V0#0:Natural}
(A{N:Natural} N:Natural + V0#0:Natural = V0#0:Natural + N:Natural)
==>
(A{N:Natural} N:Natural + s(V0#0:Natural) = s(V0#0:Natural)+ N:Natural)
+++++++++++++++++++++++++++++++++

Maude> (auto .)

=================================
```

```
label-sel: 0-comm@2.0
=================================
A{V1#0:Natural} 0 + V1#0:Natural = V1#0:Natural
==>
0 + s(V1#0:Natural) = s(V1#0:Natural)
+++++++++++++++++++++++++++++++++

Maude> (auto .)

=================================
label-sel: comm@1.0
=================================
N*Natural = 0 + N*Natural
+++++++++++++++++++++++++++++++++
```

Using Lemmas (VI)

Proving now our first original subgoal becomes automatic (because of the lemma) but we are then faced with the second original subgoal:

```
Maude> (auto .)

=================================
label-sel: comm@2.0
=================================
A{V0#0:Natural}
(A{N:Natural} N:Natural + V0#0:Natural = V0#0:Natural + N:Natural)
==>
(A{N:Natural}
N:Natural + s(V0#0:Natural) = s(V0#0:Natural)+ N:Natural)
+++++++++++++++++++++++++++++++++
```

Using Lemmas (VII)

We can apply also the `auto` tactic to the remaining goal `comm@2.0`, but, again, we get an unproved equality that we can use as a suggestion for a new lemma.

```
Maude> (auto .)

=================================
label-sel: comm@2.0
=================================
s(V0#0*Natural + N*Natural) = s(V0#0*Natural)+ N*Natural
+++++++++++++++++++++++++++++++++
```

Using Lemmas (IX)

We can again enter and prove this lemma by induction on `N:Natural` and two applications of the `auto` tactic, which brings us back to our last unproved subgoal, which we can discharge with a last `auto` command.

```
Maude> (lem s-comm : A{N:Natural ; M:Natural}
  ((s(M) + N) = (s(M + N))) .)

=================================
label: comm@2.0
=================================
s(V0#0*Natural + N*Natural) = s(V0#0*Natural)+ N*Natural

=================================
label-sel: s-comm@0
=================================
A{N:Natural ; M:Natural} s(M:Natural)+ N:Natural = s(M:Natural + N:Natural)
```

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

Maude> (ind on N:Natural .)
rewrites: 1740 in 60ms cpu (88ms real) (29000 rewrites/second)

=================================
label: comm@2.0
=================================
s(V0#0*Natural + N*Natural) = s(V0#0*Natural)+ N*Natural

=================================
label-sel: s-comm@1.0
=================================
A{M:Natural} s(M:Natural)+ 0 = s(M:Natural + 0)

=================================
label: s-comm@2.0
=================================
A{V1#0:Natural}
(A{M:Natural} s(M:Natural)+ V1#0:Natural =
s(M:Natural + V1#0:Natural))
```

```
==>
(A{M:Natural} s(M:Natural)+ s(V1#0:Natural) =
s(M:Natural + s(V1#0:Natural)))
+++++++++++++++++++++++++++++++++

Maude> (auto .)

=================================
label-sel: s-comm@2.0
=================================
A{V1#0:Natural}
(A{M:Natural} s(M:Natural)+ V1#0:Natural = s(M:Natural +
V1#0:Natural))
==>
(A{M:Natural} s(M:Natural)+ s(V1#0:Natural) =
s(M:Natural + s(V1#0:Natural)))
+++++++++++++++++++++++++++++++++

Maude> (auto .)

=================================
```

```
label-sel: comm@2.0
==================================
s(V0#0*Natural + N*Natural) = s(V0#0*Natural)+ N*Natural
++++++++++++++++++++++++++++++++++

Maude> (auto .)

q.e.d


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

Caveats on the ITP Tool

The ITP tool is for the moment an experimental system, with limited support for error messages. Therefore, if you run into parsing troubles entering a goal or a command, besides consuting the ITP Manual to make sure you did things right, you may also use parentheses generously in all goals, lemmas, and other ITP commands to help the ITP parser.

Readings and Exercises

Study the description of ITP commands in the ITP documentation, which is included in the ITP software in the course web page.

Look at, and play with, some examples of ITP proofs, which are stored, together with the files for the ITP in the course web page.

Try to prove: (1) associativity and commutativity of natural number multiplication, and (2) the list equation `rev(rev(L)) = L`, for your favorite specifications of mutiplication, and of the `rev` function that reverses a list, using the ITP tool.