# Formal Methods: Lecture 12

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

## Structural Induction

We have already observed how the ITP supports inductive proofs in three cases: natural number induction, list induction, and tree induction. But what is the general form of induction supported by the ITP for a specification having a subsignature $\Omega$ of constructors? This general form is called structural induction. It reduces proving an inductive property of the form $(\forall x : s)\ P(x)$, to proving:

- **Base Case**. For any constant $a : nil \longrightarrow s'$ in $\Omega$ with $s' \leq s$, the subgoal $P(x/a)$

  **Notation**: Given a variable $x$, the substitution mapping $x$ to a term $t$ is denoted $(x/t)$, and its homomorphic extension $\overline{(x/t)}$ is also denoted $(x/t)$.

## Structural Induction (II)

- **Induction Step**. For each constructor $f : s_1 \ldots s_n \longrightarrow s'$ in $\Omega$ with $s' \leq s$, where the sorts $s_{i_1}, \ldots, s_{i_k}$ are those among the $s_1 \ldots s_n$ such that $s_{i_j} \leq s$, $1 \leq j \leq k$, the subgoal,

  $$(\forall x_1 : s_1, \ldots, x_n : s_n)\ P(x/x_{i_1}) \wedge \ldots \wedge P(x/x_{i_k}) \Rightarrow P(x/f(x_1, \ldots, x_n)).$$

  **Note**: It may happen that none of the sorts among the $s_1 \ldots s_n$ is $s$ or a subsort of $s$. In that case, the subgoal takes the form $(\forall x_1 : s_1, \ldots, x_n : s_n)\ P(x/f(x_1, \ldots, x_n))$.

- **Subsorts Without Constructors**. If $s' \leq s$ is a subsort having no constructor constants or operators in a sort $s'' \leq s'$, then we add the subgoal $(\forall y : s')\ P(x/y)$.

## Structural Induction (III)

**Note**: If the signature $\Omega$ of constructors has been fully specified, the base case and the induction step implicitly cover all subsorts, so that the third case should never arise, except perhaps for $s'$ an empty sort, with no terms whatsoever, for which the property $P$ then trivially holds.

Therefore, from now on we will systematically ignore the case of subsorts without constructors in the rest of our theoretical discussions. In practice, however, this case is actually quite useful, and this for two reasons:

## Structural Induction (IV)

- to deal with constants in predeclared modules, such as `QID`, which are built-in and are not defined as constructors (we encountered this phenomenon in our tree-reverse example); and

- to generalize these inductive proof methods to parameterized modules, such as `LIST(X)`, where the parametric sort of elements might be a subsort of the sort `List(X)`, but we have no a priori information about the constructors of such a parametric sort of elements.

## Structural Induction (V)

Ignoring the case of subsorts without constructors, this then becomes an inductive inference rule of the form,

$$\frac{\bigwedge_i P(x/a_i) \wedge \bigwedge_j (\forall \overline{x} : \overline{s})\ P(x/x_{i_1}) \wedge \ldots \wedge P(x/x_{i_k}) \Rightarrow P(x/f_j(x_1, \ldots, x_{n_j}))}{(\forall x : s)\ P(x)}$$

where the $a_i$ and the $f_j$ include all the constructor constants and operators meeting the properties specified above, and where $(\forall \overline{x} : \overline{s})$ abbreviates $(\forall x_1 : s_1, \ldots, x_{n_j} : s_{n_j})$.

Of course, in the ITP this rule is used backwards as the `ind` rule,

$$\frac{(\forall x : s)\ P(x)}{\bigwedge_i P(x/a_i) \wedge \bigwedge_j (\forall \overline{x} : \overline{s})\ P(x/x_{i_1}) \wedge \ldots \wedge P(x/x_{i_k}) \Rightarrow P(x/f_j(x_1, \ldots, x_{n_j}))}$$

## Justification of the `ind` Rule

Why is `ind` a sound inference rule? First consider:

**Lemma**: Given a confluent and terminating equational theory $(\Sigma, E)$ with subsignature of constructors $\Omega$, and given any $\Sigma$-equation $(\forall X)\, t = t'$, we have

$$T_{\Sigma/E} \models (\forall X)\, t = t' \quad \Leftrightarrow \quad (\forall \theta : X \longrightarrow T_\Omega)\, T_{\Sigma/E} \models (\forall \emptyset)\, \overline{\theta} t = \overline{\theta} t'.$$

**Proof**: Notice that, since $\Omega$ is the subsignature of constructors, any assignment $a : X \longrightarrow T_{\Sigma/E}$ factors as $a = \theta; eval_{T_{\Sigma/E}|\Omega}$, where if $a(x) = [t]$, we can define $\theta : X \longrightarrow T_\Omega$ by $\theta(x) = can_E(t)$. Therefore, by the usual factorization and initiality argument,

$$(T_{\Sigma/E}, a) \models t = t' \quad \Leftrightarrow \quad T_{\Sigma/E} \models (\forall \emptyset)\, \overline{\theta} t = \overline{\theta} t'. \quad \text{q.e.d.}$$

## Justification of the `ind` Rule (II)

Notice that the argument of the above lemma does not depend on our formula being actually an equation: it would similarly apply to conditional equations, and even to general, universally-quantified first-order formulas.

Therefore, we have reduced the problem of proving an inductive property, $(\forall x : s)\ P(x)$, to that of proving that for all $t \in T_{\Omega,s}$ the instantiated property $P(x/t)$ holds.

Here is where structural induction steps in as a method, namely, by analyzing more closely what it means to prove something for all $t \in T_{\Omega,s}$.

## Justification of the `ind` Rule (III)

By the very inductive definition of $T_{\Omega,s}$, a term $t$ is in $T_{\Omega,s}$ iff either:

1. $t = a$, for $a : nil \longrightarrow s'$ a constant in $\Omega$ with $s' \leq s$; or

2. there is a constructor $f : s_1 \ldots s_n \longrightarrow s'$ in $\Omega$ with $s' \leq s$ (where the sorts $s_{i_1}, \ldots, s_{i_k}$ are those among the $s_1 \ldots s_n$ such that $s_{i_j} \leq s$, $1 \leq j \leq k$) and terms $t_i \in T_{\Omega,s_i}$, $1 \leq i \leq n$ such that $t = f(t_1, \ldots, t_n)$

Therefore, given a property $P(x)$ with $x$ a variable of sort $s$, if we prove that:

## Justification of the `ind` Rule (IV)

1. for each constant $a$ as above, $P(x/a)$ holds; and

2. for each $f$ as above, assuming that $P(x/t_{i_1}), \ldots, P(x/t_{i_k})$, holds then we prove that $P(x/f(t_1, \ldots, t_n))$ holds,

   (where $t_{i_j} \in T_{\Omega, s_{i_j}}$, and $s_{i_j}$, $1 \leq j \leq k$ are those sorts among the $s_1 \ldots s_n$ for the arguments of $f$ such that $s_{i_j} \leq s$, $1 \leq j \leq k$)

then we have proved that $P(x/t)$ holds for all $t \in T_{\Omega,s}$ (indeed, 1–2 above amount to a proof by induction on the depth of $t \in T_{\Omega,s}$) which, by our previous lemma, shows,

$$T_{\Sigma/E} \models (\forall x : s)\ P(x),$$

and therefore justifies `ind` as a sound inference rule.

## Need to Check Sufficient Completeness

All this is fine, but there is a pending issue. How do we know that the declared subsignature of constructors is correct? We need to check that it is sufficiently complete, for example using the SCC tool.

For example, as discussed in the justification of the `split` rule, the user may have overlooked giving enough equations for the defined functions, and then it becomes impossible to simplify every ground term to a constructor term.

A quick glance at our proof of the lemma involved in justifying the soundness of the `ind` rule shows that the reduction of proving $P(x)$ to proving $P(x/t)$ for each $t \in T_{\Omega,s}$ works just the same under much weaker assumptions than $(\Sigma, E)$ confluent, sort-decreasing, and terminating with subsignature of constructors $\Omega$.

## Verification of Functional Modules

We are now ready to consider a general methodology for verifying declarative programs. We will present the ideas in the context of verifying Maude functional modules, which are based on equational logic. The first key observation is that there are three viewpoints involved:

- the customer's viewpoint, expressed in the form of requirements that the desired software should satisfy;
- the implementor's viewpoint, whose job is to write a program meeting the customer's requirements; and
- the verifier's viewpoint, whose responsibility is to verify that the implementation does indeed meet the customer's requirements.

## The Customer's Requirements and Specification

The customer's requirements may generally be informal. Furthermore, they may involve other concerns beyond correctness, such as user-friendliness, a good graphical user interface, performance requirements, requirements about the underlying hardware and systems software, interoperability requirements, and so on.

Program verification focuses primarily on correctness requirements, which are always important, but may be crucial for safety-critical applications, were incorrect software may cause loss of human lives and/or other important damages.

## The Customer's Requirements and Specification (II)

To make possible the high assurance of correctness afforded by mathematical verification, such correctness requirements must be formalized, typically in the form of a logical theory $T_{spec}$, stating precisely the customer's (correctness) specification.

This capture of the informal correctness requirements into a formal specification can be done by the customer himself, or by an expert aiding the customer in this task. It is of course very important to make sure that the formal specification captures faithfully the informal requirements.

## The Customer's Formal Specification

In the context of Maude functional modules, it is reasonable to assume that such a formal specification will take the form of a theory,

$$T_{spec} = (\Delta, E_0 \cup Q)$$

where:

- $(\Delta_0, E_0)$, with $\Delta_0 \subseteq \Delta$, is an equational theory, that could be called the framework theory, specifying things such as key data structures and functions, including auxiliary functions needed to state key properties, and
- $Q$ is a collection of sentences in first-order logic, specifying the actual correctness properties that the software must satisfy within the $(\Delta, E_0)$ framework.

## Customer Specification: A Sorting Example

We can illustrate these ideas with a simple example, namely a customer who wants a sorting program to sort lists of integers.

As already mentioned, the customer's requirements may involve other important considerations, such as reasonable efficiency; for example, that it returns answers in time at most quadratic on the size of the input list.

Informally, the correctness requirement seems both obvious and tautological, namely, the program should return the input list in sorted form.

## Customer Specification: A Sorting Example (II)

However, in order to prove that a given implementation satisfies such a requirement, we need to capture such an informal requirement in a formalized way as a theory $T_{spec}$.

We must specify two things:

1. the data, namely lists of integers, and some auxiliary functions, and

2. the sorting function and its properties.

## Customer Specification: A Sorting Example (III)

Specifying the properties of the sorting function is not entirely trivial:

- first of all, we need to make precise what we mean by a list being sorted;
- but it is not enough to just require that the result is sorted: a function returning always the empty list will satisfy such a requirement! The original list and the sorted list should have the same elements.

All this can be stated precisely in three equational theories:

## Customer Specification: A Sorting Example (IV)

First, the data, say lists of numbers, is specified in a module such as the following INT-LIST module

```
fmod INT-LIST is protecting INT .
  sorts List .
  op nil : -> List [ctor] .
  op _:_ : Int List -> List [ctor] .
endfm
```

## Customer Specification: A Sorting Example (V)

Then, a framework theory importing INT-LIST,

```
fmod FRAME-SORTING-REQUIREMENTS is protecting INT-LIST .
  sort Multiset .
  subsort Int < Multiset .
  op sorted : List -> Bool .
  op null : -> Multiset .
  op _ _ : Multiset Multiset -> Multiset [assoc comm id: null] .
  op mset : List -> Multiset .
  vars N M : Int .
  var L : List .
  eq sorted(nil) = true .
  eq sorted(N : nil) = true .
  ceq sorted(N : M : L) = sorted(M : L)  if (N <= M) = true .
  ceq sorted(N : M : L) = false  if N <= M = false .
  eq mset(nil) = null .
  eq mset(N : L) = N mset(L) .
endfm
```

## Customer Specification: A Sorting Example (VI)

Finally, a functional theory specifying two key requirements for the `sort` function:

```
(fth SORTING-REQUIREMENTS is
  protecting FRAME-SORTING-REQUIREMENTS .
  op sort : List -> List .
  var L : List .
  eq sorted(sort(L)) = true .
  eq mset(sort(L)) = mset(L) .
endfth)
```

## Customer Specification: A Sorting Example (VII)

This is an instance of our general methodology, where the correctness specification has the form, $T_{spec} = (\Delta, E_0 \cup Q)$. Here, the framework theory $(\Delta_0, E_0)$ is the module `FRAME-SORTING-REQUIREMENTS`, and the theory $T_{spec}$ itself is `SORTING-REQUIREMENTS`.

Note that the theory $T_{spec}$ is a Full Maude specification that has been introduced with the keywords `fth`$T_{spec}$`endth`. This means that it has a loose semantics; that is, we do not require its models to be initial. However, because of the keyword `protecting FRAME-SORTING-REQUIREMENTS`, the functional submodule `FRAME-SORTING-REQUIREMENTS` is imported with its initial semantics.

## Customer Specification: A Sorting Example (VIII)

Mathematically, what this means is that, for $A$ to be an acceptable model of $T_{spec}$, besides having to satisfy the axioms in $T_{spec}$, the data types of lists, multisets, integers, and of booleans, as well as all the functions defined on them by initial algebra semantics (with keywords `fmodFRAME-SORTING-REQUIREMENTSendfm`) must be respected. In short, we must have an isomorphism,

$$A|_{\Sigma_{\texttt{FRAME-SORTING-REQUIREMENTS}}} \cong T_{\texttt{FRAME-SORTING-REQUIREMENTS}}.$$

## The Implementation: Insert-Sort

One possible Maude implementation is insert-sort,

```
fmod INSERT-SORT is
  protecting INT-LIST .
  op ins : Int List -> List .
  op sort : List -> List .
  vars N M : Int .
  var L : List .
  eq ins(N, nil) = N : nil .
  ceq ins(N, M : L) = N : M : L if N <= M = true .
  ceq ins(N, M : L) = M : ins(N, L) if N <= M = false .
  eq sort(nil) = nil .
  eq sort(N : L) = ins(N, sort(L)) .
endfm
```

## The Implementation: Insert-Sort (II)

The module $T_{imp}$ in our general methodology, becomes in this case the Maude module `INSERT-SORT`.

Note that the auxiliary function `ins` defined in `INSERT-SORT` for implementation purposes is completely different from the auxiliary functions, `sorted`, `_ _`, and `mset` defined in `FRAME-SORTING-REQUIREMENTS` for specification purposes, to formally capture the customer's correctness requirements.

Therefore, the signatures of $T_{spec}$ and $T_{imp}$ do not necessarily coincide. However, for verification purposes, they are both included as subsignatures in $T_{ver}$.

## The Theory $T_{ver}$

For verification purposes we typically need to use the auxiliary functions defined in both the framework theory $(\Delta_0, E_0)$ and in $T_{imp} = (\Sigma, E)$. This means that the theory $T_{ver} = (\Sigma', E')$ will typically have theory inclusions,

- $(\flat)$ $(\Delta_0, E_0) \subseteq (\Sigma', E')$, and
- $(\dagger)$ $(\Sigma, E) \subseteq (\Sigma', E')$.

The main goal of the verification effort is then to establish,

$$T_{ver} \vdash_{ind} Q.$$

But for this inductive inference to be applicable to the implementation theory $T_{imp} = (\Sigma, E)$, we need to require that $(\dagger)$ is a protecting inclusion, so that we have an isomorphism, $T_{\Sigma'/E'}|_{\Sigma} \cong T_{\Sigma/E}$.

## The Theory $T_{ver}$ and its Verification

In this example, the theory $T_{ver}$ must contain both the framework theory and the implementation theory:

```
fmod INSERT-SORT-VERIFICATION is
protecting INSERT-SORT .
protecting FRAME-SORTING-REQUIREMENTS .
endfm
```

We can then give this theory to the ITP and prove in it as theorems the two equations in $T_{spec}$, namely,

```
eq sorted(sort(L)) = true .
eq mset(sort(L)) = mset(L) .
```