Program Verification: Lecture 15

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

Formal Verification of Equational Programs

We shall consider two main problems in the formal verification of equational programs:

1. Proofs of Program Equivalence, that is, of equivalences of the form: `fmod` $(\Sigma, E \cup B)$ `endfm` $\equiv_{sem}$ `fmod` $(\Sigma, E' \cup B')$ `endfm` for admissible and comparable programs.

2. Proofs of Program Properties, which in their most general form, for an admissible program `fmod` $(\Sigma, E \cup B)$ `endfm`, just means proofs of properties of the form $\mathbb{C}_{\Sigma/\vec{E},B} \models \varphi$ or, equivalently, $\mathbb{T}_{\Sigma/E \cup B} \models \varphi$, for φ a first-order logic (FOL) Σ-formula.

Formal Verification of Equational Programs (II)

Regarding proofs of program equivalence, we have three theorems, namely, the Program Equivalence Theorem, and the Lemma Internalization Theorems 2 and 3, which in essence reduce all such proofs to proofs of inductive consequences of the form $(\Sigma, E \cup B) \models_{ind} G$, for G a finite set of equations.

Regarding proofs of program properties, since equational logic is a sublogic of first-order logic, we can just generalize the $\models_{ind}$ relation to first-order logic Σ-formulas φ by stating that $(\Sigma, E \cup B) \models_{ind} \varphi$ holds by definition iff $\mathbb{T}_{\Sigma/E \cup B} \models \varphi$.

This requires explaining the syntax and semantics of first-order logic, including the satisfaction relation $\mathbb{A} \models \varphi$ between a Σ-algebra $\mathbb{A}$ and a first-order logic Σ-formula φ. The Appendix to this lecture explains these topics in sufficient detail for our present purposes.

The Need for an Inductive Logic

Therefore, the task of equational program verification, both in proving program equivalences and program properties, boils down to proving inductive consequences of the form $(\Sigma, E \cup B) \models_{ind} \varphi$ (in the case of a set of equations $G = \{u_1 = v_1, \ldots, u_n = v_n\}$, $\varphi = (u_1 = v_1 \wedge \ldots \wedge u_n = v_n)$). But, by definition, proving $(\Sigma, E \cup B) \models_{ind} \varphi$ exactly means proving that $\mathbb{T}_{\Sigma/E \cup B} \models \varphi$, which is a semantic relation between the initial algebra $\mathbb{T}_{\Sigma/E \cup B}$ and a FOL formula φ.

For this, we need correct reasoning principles unambiguously embodied in a formal system of inference rules which we can rightly call an inductive logic, denoted $\vdash_{ind}$, allowing us to prove the semantic property $(\Sigma, E \cup B) \models_{ind} \varphi$ by proving $(\Sigma, E \cup B) \vdash_{ind} \varphi$.

The Need for an Inductive Logic (II)

Of course, saying that the inductive logic $\vdash_{ind}$ provides "correct reasoning principles" for this task exactly means that $\vdash_{ind}$ is sound. That is, that for any $(\Sigma, E \cup B)$ and φ we have an implication:

$$(\Sigma, E \cup B) \vdash_{ind} \varphi \quad \Rightarrow \quad (\Sigma, E \cup B) \models_{ind} \varphi$$

Can $\vdash_{ind}$ be complete, so that the reverse implication holds?

The answer is no. To explain why not, we need to observe that the set $PThm_{\vdash_{ind}}(\Sigma, E \cup B)$ of theorems of a theory $(\Sigma, E \cup B)$ provable by an inference system $\vdash_{ind}$ defined by inference rules that syntactically manipulate formulas (where the theory's "axioms" $E \cup B$ are a finite or recursively enumerable set) must be a recursively enumerable set (r.e. set). This is so because we can implement $\vdash_{ind}$ by a computer program that generates the set $PThm_{\vdash_{ind}}(\Sigma, E \cup B)$, so that $PThm_{\vdash_{ind}}(\Sigma, E \cup B)$ must be r.e.

Göedel for Dummies

Let (Σ, E) be the equational theory of the Maude program:

```
fmod NAT+x is sort Nat .
op 0 : -> Nat [ctor] .  op s: Nat -> Nat [ctor] .
ops (_+_) (_*_) : Nat Nat -> Nat .   vars N  M : Nat .
eq N + 0 = N .              eq N * 0 = 0 .
eq N + s(M) = s(N + M) .    eq N * s(M) = N + (N * M) .
```

Theorem (Göedel's Incompleteness of Arithmetic). For the above theory (Σ, E), the set

$$Thm_{\models_{ind}}(\Sigma, E) =_{def} \{\varphi \in Form_{FOL}(\Sigma) \mid \mathbb{T}_{\Sigma/E} \models_{ind} \varphi\} = Thm_{FOL}(\mathbb{T}_{\Sigma/E})$$

is not r.e.

Therefore for any sound inductive logic $\vdash_{ind}$ in general we will have a strict containment $PThm_{\vdash_{ind}}(\Sigma, E \cup B) \subset Thm_{\models_{ind}}(\Sigma, E \cup B)$, making $\vdash_{ind}$ necessarily incomplete.

The Inference System $\vdash_{ind}$ of Maude's NuITP

To prove both equational program equivalences and equational program properties we shall use Maude's New Inductive Theorem Prover (NuITP), which mechanizes the inference rules of a sound inductive logic $\vdash_{ind}$.

The formulas that $\vdash_{ind}$, and therefore Maude's NuITP, proves are quantifier-free multiclauses, which, as the Appendix to this lecture on FOL explains, are formulas of the form:

$$(w_1 = w_1' \wedge \ldots \wedge w_k = w_k') \Rightarrow ((u_1^1 = v_1^1 \vee \ldots \vee u_{m_1}^1 = v_{m_1}^1) \wedge \ldots \wedge (u_1^k = v_1^k \vee \ldots \vee u_{m_k}^k = v_{m_k}^k).)$$

Proving Inductive Theorems with the NuITP

The NuITP is a next-generation inductive theorem proper for Maude. It uses advanced symbolic techniques to automate large parts of inductive proofs, thus saving proof time and effort.

In the NuITP, standard induction on the natural numbers is generalized to induction on constructors, using the so-called generator set induction (GSI) inference rule.

To better understand generator set induction we can see how, in the case of the natural numbers, it can directly express standard natural number induction.

Let us see how associativity of addition is proved, first by standard induction, and then by the NuITP using generator set induction.

Standard Proof of Associativity of Addition

We want to prove that the addition operation in the module:

```
fmod PEANO+R is
    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
```

where `PEANO+R` suggests that we recurse on the right (`R`) argument when defining $+$, satisfies the associativity property,

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

Standard Proof of Associativity of Addition (II)

We can prove this 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).

BaseCase: We need to show,

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

We can do this trivially, by simplification with the equation

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

Standard Proof of Associativity of Addition (III)

InductionStep: We think of $\overline{\mathtt{L}}$ as a generic constant (typically written n in textbooks) and assume that the associativity equation (induction hypothesis (IH))

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

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

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

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,

Standard Proof of Associativity of Addition (IV)

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

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

and

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

q.e.d

Machine-Assisted Inductive Proofs with Maude's NuITP

Maude's NuITP is an inductive theorem prover supporting proofs by induction of properties (expressed as first-order formulas) of Maude functional modules. The NuITP is a research collaboration involving Francisco Durán at the University of Málaga, Santiago Escobar and Julia Sapiña at the Technical University of Valencia, and José Meseguer at UIUC. It is a Maude program used as follows:

- one first loads in Maude the functional module or modules one wants to reason about
- one then loads the file `NuITP.maude` into Maude.
- one sets one of the modules previously loaded in Maude as the current module and sets a multiclause as the goal to be proved.
- one then gives commands, corresponding to inductive proof steps, or formula simplification steps, to prove the chosen goal.

Proof of + Associativity with Maude's NuITP (I)

To prove the associativity of addition, we first load into Maude `PEANO+R` annotated with an RPO termination order, just as for the MTA. To prevent Maude from also loading `BOOL` we first type:

```
set include BOOL off .

fmod PEANO+R is
    sort Nat .
    op 0 : -> Nat [ctor metadata "0"] .
    op s : Nat -> Nat [ctor metadata "4"] .
    op _+_ : Nat Nat -> Nat [metadata "8"] .
    vars N M L : Nat .
    eq N + 0 = N .
    eq N + s(M) = s(N + M) .
endfm
```

Then we load `NuITP.maude` into Maude and set `PEANO+R` as current module

Proof of + Associativity with Maude's NuITP (II)

```
==================================
          NuITP (alpha 22)
       Inductive Theorem Prover
    for Maude Equational Theories
==================================
        Copyright 2021-2023
Universitat Politècnica de València
```

```
NuITP> set module PEANO+R .

  Module PEANO+R is now active.

NuITP>
```

To perform proofs that exactly correspond to natural number induction we define the following generator set:

```
NuITP> genset SIND for Nat is 0 ;; s(n:Nat) .
```

```
  Generator set SIND for sort Nat added.

  SIND (default):
    0
    s(n:Nat)

NuITP>
```

This generator set specifies that the base case is 0 and the induction step will assume the property true for n and will prove it for $s(n)$. Since we can use different generator sets for proving different properties, we give each generator set a name (here `SIND` for "standard induction"). Since this is the first generator set that has been defined for sort `Nat`, the NuITP declares `SIND` as the default generator set for `Nat`. This means that we do not need to mention its name when later performing generator set induction. Let us explore the generator set concept in more detail.

Generator Sets

For `fmod` $(\Sigma, E \cup B)$ `endfm` an admissible equational program sufficiently complete w.r.t. constructors Ω, a generator set for sort s in Σ, is a finite set of constructor terms of sort s,

$$\{u_1, \ldots, u_n\} \subseteq T_\Omega(X)_s$$

such that any ground constructor term of sort s is a ground instance modulo B of some u_i, i.e., $\forall w \in T_{\Omega_s} \;\; \exists i, 1 \leq i \leq n$, $\exists \gamma \in [vars(u_i) \rightarrow T_\Omega]$, s.t. $w =_B u_i\gamma$.

$\{0, s(K)\}$ is a generator set of sort `Nat`; and $\{0, s(0), s(s(K))\}$ is also a generator set for `Nat`: many choices are possible.

For _;_ an associative operator of sort `List` with `Nat < List`, $\{nil, n, (L; L')\}$, $\{nil, n, (m; L)\}$ and $\{nil, n, (L; m)\}$ are all generator sets of sort `List` (with variables n, m : `Nat`, L, L' : `List`).

Checking Correctness of Generator Sets

Correctness of a generator set $\{u_1, \ldots, u_n\}$ for a sort s can be reduced to: (i) checking $\{u_1, \ldots, u_n\} \subseteq T_\Omega(X)_s$ and (ii) a sufficient completeness check for a module. For $\{nil, n, (m; L)\}$ the module:

```
fmod GEN-SET-SORT-PREDICATE-FOR-List is protecting TRUTH-VALUE .
sorts Nat List .                subsorts Nat < List .
op 0 : -> Nat [ctor]            op nil : -> List [ctor] .
op s : Nat -> Nat [ctor]        op _;_ : List List -> List [ctor assoc] .
op List : List -> Bool .
eq List(nil) = true .           eq List(n:Nat) = true .
eq List(m:Nat ; L:List) = true .
endfm
```

In the current alpha version of NuITP it is the user's responsibility to check the sufficient completeness of the module defining the sort predicate associated to a generator set using Maude's SCC.

Warning: the variables of a generator set should be fresh, not appearing in any goal. And the u_i should be linear terms.

Proof of + Associativity with Maude's NuITP (III)

To prove the associativity of + we first enter the associativity property as a goal for the NuITP to prove for `PEANO+R` as follows:

```
NuITP> set goal ((N:Nat + M:Nat) + L:Nat = N:Nat + (M:Nat + L:Nat)) .

  Initial goal set.

  Goal Id: 0
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    ($3:Nat +($2:Nat + $1:Nat)) =(($3:Nat + $2:Nat) + $1:Nat)

NuITP>
```

Note that NuITP has renamed the goal's variables. We can now give the `gsi` command to prove by induction this goal (goal 0) inducting on variable `$1:Nat` as follows:

```
NuITP> apply gsi to 0 on $1:Nat .

  Generator Set Induction (GSI) applied to goal 0.

  Goal Id: 0.1
  Generated By: GSI
  Skolem Ops:
    None
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    ($3:Nat +($2:Nat + 0)) =(($3:Nat + $2:Nat) + 0)

  Goal Id: 0.2
  Generated By: GSI
```

```
  Skolem Ops:
    $4.Nat
  Executable Hypotheses:
    (($3:Nat + $2:Nat) + $4) =>($3:Nat +($2:Nat + $4))
  Non-Executable Hypotheses:
    None
  Goal:
    ($3:Nat +($2:Nat + s($4))) =(($3:Nat + $2:Nat) + s($4))

NuITP>
```

These goals are exactly those generated by standard induction on the naturals. Note that the role of the generic constant $\overline{\mathtt{L}}$ is here played by the Skolem constant `$4`.

As in standard induction, all we have left to do is to simplify these goals using: (i) the module's equations; and (ii) the induction hypothesis. In the NuITP this is done with the equality predicate simplification (`eps`) command as follows:

Proof of + Associativity with Maude's NuITP (IV)

```
NuITP> apply eps to 0.1 .

  Equality Predicate Simplification (EPS) applied to goal 0.1.

  Goal 0.1.1 has been proved.

  Unproven goals:

  Goal Id: 0.2
  Generated By: GSI
  Skolem Ops:
    $4.Nat
  Executable Hypotheses:
    (($3:Nat + $2:Nat) + $4) =>($3:Nat +($2:Nat + $4))
  Non-Executable Hypotheses:
    None
  Goal:
    ($3:Nat +($2:Nat + s($4))) =(($3:Nat + $2:Nat) + s($4))
```

```
  Total unproven goals: 1

NuITP> apply eps to 0.2 .

  Equality Predicate Simplification (EPS) applied to goal 0.2.

  Goal 0.2.1 has been proved.

  qed

NuITP>
```

The `qed` acronym indicates that there are no pending goals and the inductive proof of associativity of + has been finished, exactly as with standard induction.

If we had instead used the generator set $\{0, s(0), s(s(n))\}$ a somewhat different proof with two "base cases" and one "induction step" would have been created. The user has the freedom to choose

a generator set that best matches the recursive equations in the module. In this example the generator set $\{0, s(n)\}$ was a good match; but in other examples, involving different recursive equations, other choices may be preferable.

A good example of a module where using the $\{0, s(0), s(s(n))\}$ generator set would be better than using the `SIND` one we have used so far is provided by the `PEANO+R-FAST` module later in this lecture.

The `gsi!` Command

For many NuITP commands like `gsi` that apply an inductive inference rule, the best strategy before applying another command is to simplify the subgoals just generated using the `eps` command.

This situation is so common, that the NuITP combines both commands into the `gsi!` command, that applies `eps` to each of the goals generated by `gsi`. This can greatly shorten proofs. Let us see the effect of proving associativity of + this way:

```
NuITP> set goal ((N:Nat + M:Nat) + L:Nat = N:Nat + (M:Nat + L:Nat)) .

  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
```

```
  Executable Hypotheses:
    None
  Non-Executable Hypotheses:
    None
  Goal:
    ($3:Nat +($2:Nat + $1:Nat)) =(($3:Nat + $2:Nat) + $1:Nat)

NuITP> apply gsi! to 0 on $1:Nat .

  Generator Set Induction with Equality Predicate Simplification (GSI!) applied
    to goal 0.

  Goals 0.1.1 and 0.2.1 have been proved.

  qed

NuITP>
```

Proving Program Equivalences in NuITP

Recall from the Program Equivalence Theorem in Lecture 14 that for comparable admissible modules `fmod` $(\Sigma, E \cup B)$ `endfm` $\equiv_{sem}$ `fmod` $(\Sigma, E' \cup B')$ `endfm` iff

$$(\Sigma, E \cup B) \models_{ind} (E'_0 \setminus E_0) \cup (B' \setminus B).$$

In particular, proving program equivalences can be useful for program optimization purposes.

Let us prove that our equational program `PEANO+R` is semantically equivalent to the following program `PEANO+R-FAST`, which runs, roughly, twice as fast.

Proving Program Equivalences in NuITP (II)

```
fmod PEANO+R-FAST is
    sort Nat .
    op 0 : -> Nat [ctor metadata "0"] .
    op s : Nat -> Nat [ctor metadata "4"] .
    op _+_ : Nat Nat -> Nat [metadata "8"] .
    vars N M : Nat .
    eq N + 0 = N .
    eq N + s(0) = s(N) .
    eq N + s(s(M)) = s(s(N + M)) .
endfm
```

Note that a good generator set for this program, matching its recursive equations, is: $\{0, s(0), s(s(n))\}$. Proofs for this module using this generator set will tend to be shorter than proofs using the "vanilla flavored" generator set $\{0, s(n)\}$.

Let us prove that `PEANO+R` and `PEANO+R-FAST` are semantically equivalent.

Proving Program Equivalences in NuITP (III)

if we choose PEANO+R as our $(\Sigma, E \cup B)$ and PEANO+R-FAST as our $(\Sigma, E' \cup B')$, and noticing that $B = B' = \emptyset$, $E = E_0$, $E' = E'_0$ and $E'_0 \setminus E_0 = \{N + s(0) = s(N), N + s(s(M)) = s(s(N + M))\}$, we will prove that PEANO+R and PEANO+R-FAST are semantically equivalent using the NuITP if in PEANO+R if we prove the inductive goal:

```
NuITP> set goal (N:Nat + s(0) = s(N:Nat)) /\
       (N:Nat + s(s(M:Nat)) = s(s(N:Nat + M:Nat))) .

  Initial goal set.

  Goal Id: 0
  Generated By: init
  Skolem Ops:
    None
  Executable Hypotheses:
    None
```

```
  Non-Executable Hypotheses:
    None
  Goal:
    (s($2:Nat) =($2:Nat + s(0))) /\ s(s($2:Nat + $1:Nat)) =($2:Nat + s(s(
    $1:Nat)))

NuITP>
```

This goal is so simple that we do not need to use induction: just applying the equality predicate simplification rule `eps` suffices:

```
NuITP> apply eps to 0 .

  Equality Predicate Simplification (EPS) applied to goal 0.

  Goal 0.1 has been proved.

  qed

NuITP>
```