Program Verification: Lecture 25

José Meseguer

University of Illinois
at Urbana-Champaign

Verification of Imperative Sequential Programs

We are now ready to consider the verification of sequential imperative programs. We will do so using a simple imperative language called IMP.

Of course, for the formal verification of some properties Q about a program P in a sequential imperative language $\mathcal{L}$ to be meaningful at all, our first and most crucial task is to make sure that the programming language $\mathcal{L}$ has a clear and precise mathematical semantics, since only then can we settle mathematically whether a program P satisfies some properties Q.

Verification of Imperative Sequential Programs (II)

The issue of giving a mathematical semantics to a programming language $\mathcal{L}$ is actually nontrivial, particularly for imperative languages; it is of course much easier for a declarative language, since we can rely on the underlying logic on which such a language is based.

For example, for a Maude functional module, its mathematical semantics is given by the initial algebra of its equational theory, whereas its operational semantics is based on equational simplification with its equations, which are assumed confluent and terminating.

Some imperative languages have never been given a precise semantics; their only precise documentation may be the different compilers, perhaps inconsistent with each other.

Verification of Imperative Sequential Programs (III)

In the end, giving mathematical semantics to a programming language $\mathcal{L}$ amounts to giving a mathematical model of the language. This is typically done using some mathematical formalism: either the language of set theory, which is a de-facto universal formalism for mathematics, or some other well-defined formalism.

For sequential imperative languages equational formalisms are quite well-suited to the task. In traditional denotational semantics, a higher-order equational logic, namely the lambda calculus, is used. However, it was pointed out by a number of authors, including Joseph Goguen, that first-order equational logic is perfectly adequate for the task, and has some specific advantages.

Algebraic Semantics of Sequential Languages

The choice of first-order equational logic leads to a form of algebraic semantics of sequential imperative languages in which:

- the semantics of a programming language $\mathcal{L}$ is axiomatized as an equational theory $\mathcal{E}_{\mathcal{L}}$;
- the mathematical semantics of the language is given by the initial algebra $\mathcal{T}_{\mathcal{E}_{\mathcal{L}}}$;
- if the equations in $\mathcal{E}_{\mathcal{L}}$ are ground confluent and sort-decreasing, this also gives an operational semantics to the language, expressed in terms of equational simplification.

Algebraic Semantics of Sequential Languages (II)

In this setting, the program correctness question can be formulated as follows: given a program P in a sequential imperative language $\mathcal{L}$, and given some properties Q about P (where Q typically involves the text of P) we say that P satisfies Q iff,

$$\mathcal{T}_{\mathcal{E}_{\mathcal{L}}} \models Q.$$

Proof-theoretically, we use an inductive inference system, to try to prove,

$$T_{\mathcal{L}} \vdash_{ind} Q.$$

Algebraic Semantics of Sequential Languages (III)

Given a language $\mathcal{L}$, we can interpret it by an equational theory,

$$\mathcal{E}_{\mathcal{L}} = (\Sigma_{\mathcal{L}}, E^t_{\mathcal{L}} \cup B \cup E^{nt}_{\mathcal{L}})$$

where:

- $(\Sigma_{\mathcal{L}}, E^t_{\mathcal{L}} \cup B)$ is a confluent and terminating equational subtheory that axiomatizes the terminating fragment of the language,
- and equations $E^{nt}_{\mathcal{L}}$ capture the non-terminating fragment.

Note if $\mathcal{L}$ is Turing Complete then we must have $E^{nt}_{\mathcal{L}} \neq \emptyset$.

Algebraic Semantics of IMP

```
fmod IMP-SYNTAX is
  sort Id Bool NzNat Nat .
  subsort NzNat < Nat .
  ops a b c d e f g i j k l m n
      o p q r s t u v w x y z : -> Id [ctor] .
  op  _, : Id -> Id [ctor] .
  ops true false : -> Bool [ctor] .
  op  0 : -> Nat [ctor] .
  op  1 : -> NzNat [ctor] .
  op  _+_ : Nat Nat -> Nat [ctor assoc comm id: 0] .
  op  _+_ : NzNat Nat -> NzNat [ctor ditto] .
```

```
sort BoolExp BoolRedex NatExp NatRedex .
subsort Id < NatRedex .
subsort Nat  NatRedex  < NatExp .
subsort Bool BoolRedex < BoolExp .
ops (_&&_) (_||_) : BoolExp BoolExp -> BoolRedex [ctor] .
op  ~_ : BoolExp -> BoolRedex [ctor] .
ops (_<_) (_<=_) (_=_) : NatExp NatExp ->
                         BoolRedex [ctor] .
op  _+_ : NatRedex NatExp -> NatRedex [ditto] .
op  _+_ : NatExp NatExp -> NatExp [ditto] .
op  _-_ : NatExp NatExp -> NatRedex [ctor] .
```

```
  sort BasicStmt Stmt .
  subsort BasicStmt < Stmt .
  op  _;_  : Stmt Stmt -> Stmt [ctor assoc prec 60] .
  op  skip : -> BasicStmt [ctor] .
  op  _:=_ : Id NatExp -> BasicStmt [ctor] .
  op  if_then_fi : BoolExp Stmt -> BasicStmt [ctor] .
  op  while_do_od : BoolExp Stmt -> BasicStmt [ctor] .
endfm
```

```
fmod IMP-REDUCE is pr IMP-SYNTAX .
  op  ~Bool_                    : Bool -> Bool .
  ops (_/\Bool_) (_\/Bool_) : Bool Bool -> Bool .
  op  _-Nat_                    : Nat Nat -> Nat .
  ops (_<Nat_)   (_<=Nat_)  : Nat Nat -> Bool .
  op  (_=Nat_)                  : Nat Nat -> Bool [comm] .
  var N M : Nat . var P : NzNat . var B : Bool .

  eq ~Bool true  = false .
  eq ~Bool false = true .
  eq true  /\Bool B = B .
  eq false /\Bool B = false .
  eq true  \/Bool B = true .
  eq false \/Bool B = B .
```

```
  eq  N       -Nat  (N + M) = 0 .
  eq (N + P) -Nat   N       = P .
  eq  N       <Nat   N + P  = true .
  eq  N + M  <Nat   N       = false .
  eq  N       <=Nat  N + M  = true .
  eq  N + P  <=Nat  N       = false .
  eq  N + P   =Nat  N       = false .
  eq  N        =Nat  N       = true .
endfm

fmod IMP-MEM is pr IMP-SYNTAX .
  sort Memory .
  op [_,_] : Id Nat -> Memory [ctor] .
  op none  : -> Memory [ctor] .
  op __    : Memory Memory ->
             Memory [ctor assoc comm id: none] .
endfm
```

```
fmod IMP-EVAL is pr IMP-MEM + IMP-REDUCE .
  op eval : Memory NatExp  -> Nat .
  op eval : Memory BoolExp -> Bool .
  var NE1 NE2 : NatExp   . var B : Bool . var P : NzNat  .
  var BE1 BE2 : BoolExp  . var N : Nat  . var M : Memory .
  var NR1 NR2 : NatRedex . var I : Id   .
  eq eval(M,NR1 + P  )  = eval(M,NR1) + P .
  eq eval(M,NR1 + NR2)  = eval(M,NR1) + eval(M,NR2) .
  eq eval(M,NE1 - NE2)  = eval(M,NE1) -Nat   eval(M,NE2) .
  eq eval(M,BE1 && BE2) = eval(M,BE1) /\Bool eval(M,BE2) .
  eq eval(M,BE1 || BE2) = eval(M,BE1) \/Bool eval(M,BE2) .
  eq eval(M,NE1 <  NE2) = eval(M,NE1) <Nat   eval(M,NE2) .
  eq eval(M,NE1 <= NE2) = eval(M,NE1) <=Nat  eval(M,NE2) .
  eq eval(M,NE1  = NE2) = eval(M,NE1)  =Nat  eval(M,NE2) .
  eq eval(M,~ BE1)      = ~Bool eval(M,BE1) .
  eq eval([I,N] M,I)    = N .
  eq eval(M,N)          = N .
  eq eval(M,B)          = B .
endfm
```

```
mod IMP is pr IMP-EVAL + IMP-SYNTAX .
  sort State .
  op _|_ : Stmt Memory -> State [ctor] .
  var I  : Id    . var NE : NatExp     . var S S' : Stmt .
  var N  : Nat   . var BR : BoolRedex . var M : Memory .
  var B  : Bool . var BE : BoolExp    .

  eq skip    ; S' | M       = S' | M .
  eq I := NE ; S' | [I,N] M = S' | [I,eval([I,N] M,NE)] M .

  eq if true  then S fi ; S' | M = S ; S' | M .
  eq if false then S fi ; S' | M = S' | M .

  eq if BR then S fi ; S' | M =
     if eval(M,BR) then S fi ; S' | M .
  eq while BE do S od ; S' | M =
     if BE then S ; while BE do S od fi ; S' | M .
endm
```

Algebraic Semantics of IMP (II)

Then we obtain the algebraic semantics for IMP:

$$\mathcal{E}_{\texttt{IMP}} = (\texttt{IMP-SYNTAX}, \texttt{IMP-EVAL} \cup \texttt{IMP})$$

where `IMP` is non-terminating.

Thus, while we do not have $C_{\texttt{IMP-SYNTAX}/\texttt{IMP-EVAL}\cup\texttt{IMP}}$ and cannot obtain an interpreter by naive equational simplification, we can still reason about $\mathcal{T}_{\texttt{IMP-SYNTAX}/\texttt{IMP-EVAL}\cup\texttt{IMP}}$ using an inductive theorem prover or $\mathcal{C}_{\texttt{IMP-SYNTAX}/\texttt{IMP-EVAL}}$ by equational simplification.

Algebraic Semantics of IMP (III)

For example, in $\mathcal{C}_{\texttt{IMP-SYNTAX/IMP-EVAL}}$ we can directly prove the commutativity of addition by simplification (_+_ is ACU):

$$
\begin{aligned}
\texttt{eval}([\mathrm{x}, X][\mathrm{y}, Y], \texttt{x + y}) &=_{\texttt{IMP-EVAL}} \\
\texttt{eval}([\mathrm{x}, X][\mathrm{y}, Y], x) \texttt{ + } \texttt{eval}([\mathrm{x}, X][\mathrm{y}, Y], y) &=_{\texttt{IMP-EVAL}} \\
X \texttt{ + } Y &=_{\texttt{IMP-EVAL}} \\
Y \texttt{ + } X &=_{\texttt{IMP-EVAL}} \\
\texttt{eval}([\mathrm{x}, X][\mathrm{y}, Y], y) \texttt{ + } \texttt{eval}([\mathrm{x}, X][\mathrm{y}, Y], x) &=_{\texttt{IMP-EVAL}} \\
\texttt{eval}([\mathrm{x}, X][\mathrm{y}, Y], \texttt{y + x})
\end{aligned}
$$

Q: Can we still obtain a mathematical, executable semantics (i.e. an interpreter) for all of IMP (incl. statements)?

Rewriting Semantics of Sequential Languages

Given algebraic semantics $\mathcal{E}_{\mathcal{L}} = (\Sigma_{\mathcal{L}}, E^t_{\mathcal{L}} \cup B \cup E^{nt}_{\mathcal{L}})$, by viewing $E^{nt}_{\mathcal{L}}$ as rewrite rules, we obtain a rewriting semantics:

$$\mathcal{R}_{\mathcal{L}} = (\Sigma_{\mathcal{L}}, E^t_{\mathcal{L}} \cup B, E^{nt}_{\mathcal{L}}).$$

Then we have initial reachability model $\mathcal{T}_{\mathcal{R}_{\mathcal{L}}}$; to prove property Q of program P in language $\mathcal{L}$, we just need to show:

$$\mathcal{T}_{R_{\mathcal{L}}} \models Q.$$

If $E^{nt}_{\mathcal{L}}$ is coherent with $E^t_{\mathcal{L}}$ modulo B, we also have canonical reachability model $\mathcal{C}_{\mathcal{R}_{\mathcal{L}}} \cong \mathcal{T}_{R_{\mathcal{L}}}$ and thus $\mathcal{L}$ has a mathematical, executable semantics (an interpreter) via rewriting.

Rewriting Semantics of IMP

Applying this idea to IMP, we obtain the rewrite theory:

$$\mathcal{R}_{\texttt{IMP}} = (\texttt{IMP-SYNTAX}, \texttt{IMP-EVAL}, \texttt{IMP}).$$

where all equations in IMP become rewrite rules. We also have the canonical rewrite theory $C_{\mathcal{R}_{\texttt{IMP}}}$. We can prove property Q about a program P by showing $C_{\mathcal{R}_{\texttt{IMP}}} \models Q$.

Q: How can mechanize checking $C_{\mathcal{R}_{\texttt{IMP}}} \models Q$ (or, more generally, how can we mechanize checking $\mathcal{C}_{R_{\mathcal{L}}} \models Q$)?

A: For some theories, we could possibly abstract/bound our systems and do model checking via search; in other cases, we can apply our Reachability Logic proof system.

An Example IMP Program

Consider the following IMP programs $swap(X, Y)$ and $skip(X, Y)$:

$$\texttt{while y < o do x := x - 1 ; y := y + 1 od |} \\ \texttt{[x,}X\texttt{] [y,}Y\texttt{] [o,}X\texttt{]}$$

$$\texttt{skip | [x,}X\texttt{] [y,}Y\texttt{] [o,}X\texttt{]}$$

Let $SWAP$ be the property that the numbers are swapped, i.e.

$$swap(X, Y) \mid Y \texttt{ <= } X = \texttt{true} \longrightarrow^{\circledast} skip(Y, X) \mid true$$

Interlude: Two Presentations of Hoare Logic

We saw previously Hoare Logic (HL) triples $\{A\}\ \mathcal{R}\ \{B\}$ are a special case of Reachability Logic (RL) formulas $A \longrightarrow^{\circledast} B$.

Since `skip` is a terminating state for IMP, we know $SWAP$ can be described as Hoare triple. We now show $SWAP$ in two different presentations of Hoare Logic:

- The presentation we saw previously, i.e. $\{swap(X, Y) \mid Y \texttt{ <= } X = \texttt{true}\}\ \texttt{IMP}\ \{skip(Y, X) \mid true\}$
- The classical presentation, i.e.

$$
\begin{gathered}
\{Y \texttt{ <= } X = \texttt{true} \wedge X = I \wedge Y = J\} \\
swap(X, Y) \\
\{X = J \wedge Y = I\}
\end{gathered}
$$

Interlude: Two Presentations of Hoare Logic (II)

$$\{swap(X,Y) \mid Y \texttt{ <= } X = \texttt{true}\}\ \texttt{IMP}\ \{skip(Y,X) \mid true\}$$

$$\{Y \texttt{ <= } X = \texttt{true} \wedge X = I \wedge Y = J\}\ swap(X,Y)\ \{X = J \wedge Y = I\}$$

Q: What important differences do these two presentations have?
A: In classical Hoare Logic,

- there is no underlying term structure/transition system; program syntax/semantics *directly* encoded in proof rules,
- thus, for each programming language, we need to *redefine* Hoare Logic for that language,
- in particular, logical and program variables coincide, which is arguably more confusing than helpful, and
- without term structure, reasoning about call stacks, heaps, exception stacks, class hierarchies, etc... *nontrivial*.

Verification by Model Checking

Using model checking via search, we can try to verify $SWAP$ *upto a given loop bound*. For example, using Maude search, by letting $X \geq Y \geq 0$, we can verify $SWAP$ upto X:

search swap(X, Y) =>! S such that S = skip(Y, X) .

where `S:State`. As an example, setting $X = 10$ and $Y = 3$, after performing exhaustive search, Maude replies:

```
Solution 1 (state 38)
states: 39  rewrites: 118
S --> skip | [o,10] [x,3] [y,10]

No more solutions.
states: 39  rewrites: 118
```

Verification by Reachability Logic

Since Reachability Logic can directly capture *inductive reasoning*, we can prove $SWAP$ for all values of X and Y as shown below:

```
(select SWAP .)
(def-term-set (skip | M:Memory) | true .)
(add-goal (swap | [x,X:Nat] [y,Y:Nat] [o,O:Nat]) |
   (Y:Nat <=Nat O:Nat) = (true) /\
   (O:Nat <=Nat X:Nat + Y:Nat) = (true) =>A
   (skip | [x,X':Nat] [y,Y':Nat] [o,O:Nat]) |
   (X':Nat + Y':Nat) = (X:Nat + Y:Nat) /\
   (Y':Nat) = (O:Nat) .)
(add-goal (swap | [x,X:Nat] [y,Y:Nat] [o,X:Nat]) |
   (Y:Nat <=Nat X:Nat) = (true) =>A
   (skip | [x,X':Nat] [y,Y':Nat] [o,X:Nat]) |
   (X':Nat) = (Y:Nat) /\ (Y':Nat) = (X:Nat) .)
(start-proof .)
(step* .)
```