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

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

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 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] .
   op true :                -> Bool [ctor] .
   op 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] .
   op _*_: Nat Nat        -> Nat  [ctor assoc comm] .

subsort Id < NatRedex.

subsort Nat NatRedex < NatExp.

subsort Bool BoolRedex < BoolExp.

op _&&_: BoolExp BoolExp -> BoolRedex [ctor].

op _||_: BoolExp BoolExp -> BoolRedex [ctor].

op ~_: BoolExp -> BoolRedex [ctor].

op _<_: NatExp NatExp -> BoolRedex [ctor].

op _<=_: NatExp NatExp -> BoolRedex [ctor].

op _=_: NatExp NatExp -> BoolRedex [ctor].

op _-_: NatExp NatExp -> NatRedex [ctor].

op _+_: NatRedex NatExp -> NatRedex [ditto].

op _+_: NatExp NatExp -> NatExp [ditto].

op _*_: NatExp NatExp -> NatRedex [ditto].
sort BasicStmt Stmt .
subsort BasicStmt < Stmt .
op _;_ : Stmt Stmt -> Stmt [ctor assoc id: skip 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-DATA is pr IMP-SYNTAX.

  op ~Bool_ : Bool --> Bool.
  op _/
      Bool_ : Bool Bool --> Bool.
  op _\/Bool_ : Bool Bool --> Bool.
  op _-Nat_ : Nat Nat --> Nat.
  op _<Nat_ : Nat Nat --> Bool.
  op _<=Nat_ : Nat Nat --> Bool.
  op _=Nat_ : Nat Nat --> Bool [comm].
  op _*Nat_ : Nat Nat --> Nat [assoc comm].

  var N M : Nat. var P Q R : 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 .
eq N *Nat 1 = N .
eq N *Nat 0 = 0 .
eq (P + Q) *Nat R = (R *Nat P) + (R *Nat Q) .
endfm
fmod IMP-MEM is pr IMP-SYNTAX .
    sort PreMemory Memory .
    op [_,_] : Id Nat -> PreMemory [ctor] .
    op none : -> PreMemory [ctor] .
    op __ : PreMemory PreMemory
       -> PreMemory [ctor assoc comm id: none] .
    op {_} : PreMemory -> Memory [ctor] .
    op err : -> Memory [ctor] .
    var I : Id . var N N’ : Nat . var M : PreMemory .
    eq {[I,N] [I,N’]} = err .
endfm
fmod IMP-EVAL is pr IMP-MEM + IMP-DATA.

var NE1 NE2 : NatExp . var B : Bool . var P : NzNat .
var BE1 BE2 : BoolExp . var N : Nat . var PM : PreMemory .
var NR1 NR2 : NatRedex . var I : Id . var M : Memory .
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,NE1 * NE2) = eval(M,NE1) *Nat eval(M,NE2) .
eq eval(M,~ BE1) = ~Bool eval(M,BE1) .
eq eval(M,N) = N .
eq eval(M,B) = B .
ceq eval([I,N] PM),I) = N if ([I,N] PM) =/= err .
endfm
mod IMP-NUMBERS is pr IMP-SYNTAX.
    ops 2 3 4 5 6 7 8 9 10 : -> NzNat.
    eq 2 = 1 + 1.
    eq 3 = 1 + 1 + 1.
    eq 4 = 1 + 1 + 1 + 1.
    eq 5 = 1 + 1 + 1 + 1 + 1.
    eq 6 = 1 + 1 + 1 + 1 + 1 + 1.
    eq 7 = 1 + 1 + 1 + 1 + 1 + 1 + 1.
    eq 8 = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1.
    eq 9 = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1.
    eq 10 = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1.
endm
mod IMP is pr IMP-EVAL + IMP-SYNTAX + IMP-NUMBERS .

sort ImpState .


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 . var PM : PreMemory .

eq I := NE ; S' | {[I,N] PM} =
    S' | {[I,eval({[I,N] PM},NE)] PM} .

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
In this way we obtain an algebraic semantics for IMP:

$$\mathcal{E}_{\text{IMP}} = (\Sigma_{\text{IMP}}, E_{\text{IMP-EVAL}} \cup E_{\text{IMP}} \cup B)$$

where $E_{\text{IMP}}$ is non-terminating.

Specifically, $\mathcal{E}_{\text{IMP}}$ gives three things:

- A parser for IMP
- An executable operational semantics which is also an interpreter for IMP
- A mathematical semantics for IMP, namely the initial algebra for $T_{\mathcal{E}_{\text{IMP}}}$. 
Parsing of IMP Programs

Programs in IMP are just terms in the module IMP-SYNTAX. Therefore our semantics $\mathcal{E}_{\text{IMP}}$ automatically gives us an IMP parser. Consider for example the IMP programs:

\begin{align*}
n & := 0 ; \\
\text{while} \ true \ \text{do} & \\
& \text{while} \ n < 6 \ \text{do} \\
& \quad n := n + 1 \\
& \quad \text{od} ; \\
& \text{od} \\
n & := 0 \\
& \text{od}
\end{align*}

\begin{align*}
s & := 1 ; \\
\text{while} \ 0 < n \ \text{do} & \\
& \quad s := s \times n ; \\
& \quad n := n - 1 \\
& \quad \text{od}
\end{align*}
We can parse programs in Maude by giving the parse command:

Maude> parse s := 1 ;
    while 0 < n do s := s * n ; n := (n - 1) od .
Stmt: s := 1 ; while 0 < n do s := s * n ; n := (n - 1) od

Maude> parse n := 0 ;
    while true do while n < 6 do n := n + 1 od ; n := 0 od .
Stmt: n := 0 ;
    while true do while n < 6 do n := n + 1 od ; n := 0 od
An Interpreter for \textbf{IMP} Programs

Since the theory $\mathcal{E}_{\text{IMP}} = (\Sigma_{\text{IMP}}, E_{\text{IMP-EVAL}} \cup E_{\text{IMP}} \cup B)$ is ground confluent modulo $B$, it gives us an executable operational semantics for IMP by term rewriting. In Maude this also provides us with an interpreter for IMP.
Now that we have an interpreter, let’s run it!

\[
\text{red } n := 3 \; ; \; m := 4 \; ; \\
\text{while } 0 < m \text{ do } m := m - 1 \; ; \; e := e * n \text{ od |} \\
\{[n,0] \; [m,0] \; [e,1]\} \; . \\
\text{rewrites: 111 in 0ms cpu (0ms real) (~ rewrites/second)} \\
\text{result State: skip | } [e,81] \; [m,0] \; [n,3]
\]

\[
\text{red } n := 5 \; ; \\
\text{while } 0 < n \text{ do } s := n * s \; ; \; n := (n - 1) \text{ od |} \\
\{[n,0] \; [s,1]\} \; . \\
\text{rewrites: 129 in 0ms cpu (0ms real) (~ rewrites/second)} \\
\text{result State: skip | } [n,0] \; [s,120]
\]
Given algebraic semantics $\mathcal{E}_L = (\Sigma_L, E^t_L \cup B \cup E^{nt}_L)$, by viewing $E^{nt}_L$ as rewrite rules $\vec{E}^{nt}_L$ we also obtain a rewriting logic semantics:

$$\mathcal{R}_L = (\Sigma_L, E^t_L \cup B, \vec{E}^{nt}_L).$$

Then we have initial reachability model $\mathcal{T}_{\mathcal{R}_L}$. This model can be quite useful to prove a property $Q$ of a program $P$ in language $L$ by model checking. We just need to show:

$$\mathcal{T}_{\mathcal{R}_L} \models Q.$$ 

Assuming $\vec{E}^{nt}_L$ is coherent with $E^t_L$ modulo $B$, we get a canonical reachability model $\mathcal{C}_{\mathcal{R}_L} \cong \mathcal{T}_{\mathcal{R}_L}$ and thus $L$ can be used to model check properties of programs in $L$. 

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

\[ R_{\text{IMP}} = (\text{IMP-SYNTAX}, E_{\text{IMP-EVAL}}, \vec{E}_{\text{IMP}}). \]

where all equations in IMP become rewrite rules. We also have the canonical rewrite theory \( C_{R_{\text{IMP}}} \). We can prove property \( Q \) about a program \( P \) by showing \( C_{R_{\text{IMP}}} \models Q \).

**Q:** How can mechanize checking \( C_{R_{\text{IMP}}} \models Q \) (or, more generally, how can we mechanize checking \( C_{R_{L}} \models Q \))?

**A:** For terminating or finite-state programs, or by using abstraction/bounding, we can do model checking via search or LTL model checking; in other cases, we can apply our Reachability Logic proof system.
Consider the following IMP programs $swap(X, Y)$ and $skip(X, Y)$:

$$swap(X, Y) = \text{while } y < o \text{ do } x := x - 1 ; y := y + 1 \text{ od}$$

$$skip(X, Y) = \text{skip} \mid [x, X] \mid [y, Y] \mid [o, X]$$

We would like to verify that whenever $X \geq Y \geq 0$ \(swap(X, Y)\) terminates as \(skip(X, Y)\). A proof for all such $X$ and $Y$ requires using Reachability Logic. However, we can use model checking to verify the property for concrete instances of $X$ and $Y$. 
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 $S = \text{skip}(Y,X)$.

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

Solution 1 (state 38)
states: 39  rewrites: 118
$S --> \text{skip} \mid [o,10] [x,3] [y,10]$

No more solutions.
states: 39  rewrites: 118
For model checking properties beyond invariants, *temporal logic* can be quite useful—particularly when analyzing *finite state, non-terminating* programs.

In order to model check temporal logic formulas, we typically use *LTL*. For this to work, we will need to define *predicates* over our *program state*.

Let us see an example.
One very common non-terminating and finite state program is a digital clock. Typically, such clocks include counters for hours and minutes (and possibly seconds). For simplicity, suppose our clock only counts seconds. See the clock program below:

```plaintext
while true do
  while n < 6 * 10 do
    n := n + 1 od;
  n := 0 od | {[n,0]}
```

Here the variable $n$ represents the current second counter.
There are several properties we might wish to verify for our clock. We give two examples below:

1. the clock counter is never greater than 60.
2. it is always the case that if the clock counter equals 0 seconds, it will later equal 60.

Let us see how to define these properties.
In Maude, we typically define LTL properties by extending SATISFACTION. We define properties is-eq and is-le below.

```
mod IMP-PREDs is
  pr IMP-SEMANTICS .
  inc SATISFACTION *
    (sort Bool to Bool*, op true to tt, op false to ff).
  subsort ImpState < State .
  op is-eq : Id Nat -> Prop .
  op is-le : Id Nat -> Prop .
  var S : Stmt . var M : PreMemory .
  var I : Id . var N N' : Nat .
  eq (S | {M [I,N]}) |= is-eq(I,N') = N =Nat N' == true .
  eq (S | {M [I,N]}) |= is-le(I,N') = N <=Nat N' == true .
endm
```
Here, we verify the first property, i.e. the clock counter never exceeds 60.

```plaintext
red modelCheck(
    while true do
        while n < 6 * 10 do
            n := n + 1 od ;
        n := 0 od | {[n,0]},
        [] is-le(n, eval({none}, 6 * 10))) .
rewrites: 2285 in 0ms cpu (3ms real) (~ rewrites/second)
result Bool: true
```
Here we verify the second property, i.e. it is always the case that if the clock counter equals 0, it will eventually equal 60.

\[
\text{red modelCheck(}
\begin{array}{l}
\text{while true do} \\
\text{while } n < 6 \times 10 \text{ do } \\
\quad n := n + 1 \text{ od ;} \\
\quad n := 0 \text{ od | } \{[n,0]\}, \\
\quad \text{is-eq}(n,0) \implies \text{is-eq}(n,\text{eval}(\{\text{none}\},6 \times 10)) \text{.)}
\end{array}
\]

rewrites: 2542 in 0ms cpu (3ms real) (~ rewrites/second)
result Bool: true