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 $L$ to be meaningful at all, our first and most crucial task is to make sure that the programming language $L$ 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.
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.
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 $E^{nt}_\mathcal{L} \neq \emptyset$. 
fmod IMP-SYNTAX is  protecting BOOL .
sorts Id NzNat Nat .
subsort NzNat < Nat .
op 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 0 :                  -> Nat    [ctor] .
op 1 :                  -> NzNat   [ctor] .
ops 2 3 4 5 6 7 8 9 :   -> NzNat .

eq 2 = 1 + 1 .  eq 3 = 1 + 2 .  eq 4 = 1 + 3 .
eq 5 = 1 + 4 .  eq 6 = 1 + 5 .  eq 7 = 1 + 6 .
eq 8 = 1 + 7 .  eq 9 = 1 + 8 .
sort BoolExp NatExp .
subsorts Nat Id < NatExp .
subsort Bool < BoolExp .
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 _-Nat_ : Nat Nat -> Nat . *** monus
op _<Nat_ : Nat Nat -> Bool .
op _<=Nat_ : Nat Nat -> Bool .
var N M : Nat . var P Q R : NzNat . var B : Bool .

  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 Cell PreMemory Memory .
  subsort Cell < PreMemory .
  op `{}` : PreMemory -> Memory [ctor] .
  op `[_,_]` : Id Nat -> Cell [ctor] .
  op none : -> PreMemory [ctor] .
  op `__` : PreMemory PreMemory
          -> PreMemory [ctor assoc comm id: none] .
  op err : -> Memory [ctor] .

  var I : Id . vars N N’ : Nat . var M : PreMemory .

  eq {[I,N] [I,N’] M} = err .
endfm
fmod IMP-EVAL is pr IMP-MEM + IMP-DATA.

op eval : PreMemory NatExp -> [Nat].
op eval : PreMemory BoolExp -> [Bool].

var NE1 NE2 : NatExp . var B : Bool . var P : NzNat .
var BE1 BE2 : BoolExp . var N K : Nat . var M : PreMemory .
var I : Id .

eq eval(M,N) = N .
eq eval(M,B) = B .
ceq eval(M,NE1 + NE2) = eval(M,NE1) + eval(M,NE2) if NE1 :: Nat /= true \ NE2 /= 0 .
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 && BE2) = eval(M,BE1) and eval(M,BE2) .
eq eval(M,BE1 || BE2) = eval(M,BE1) or eval(M,BE2) .
eq eval(M,~ BE1) = not eval(M,BE1) .
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) .
endfm
mod IMP is pr IMP-EVAL + IMP-SYNTAX.

sort State.

op _|_ : Stmt [PreMemory] -> State [ctor prec 61].

var I : Id. var NE : NatExp. var S S' : Stmt.
var N : Nat. var BR : BoolRedex. var M : PreMemory.
var B : Bool. var BE : BoolExp.

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
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 \( \mathcal{T}_{\mathcal{E}_{\text{IMP}}} \).
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:

```
n := 0 ;
while true
  do
    while n < 6 do
      n := n + 1
      od ;
  n := 0
  od

s := 1 ;
while 0 < n
  do
    s := s * n ; n := n - 1
  od
```
Parsing of IMP Programs

We can then parse these programs in Maude by giving the parse command:

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

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
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.
Rewriting Semantics of Sequential Languages

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

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

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

$$T_{R_\mathcal{L}} \models Q.$$

Assuming $\vec{E}^{nt}_\mathcal{L}$ is coherent with $E^t_\mathcal{L}$ modulo $B$, we get a canonical reachability model $C_{R_\mathcal{L}} \simeq T_{R_\mathcal{L}}$ and thus $\mathcal{L}$ can be used to model check properties of programs in $\mathcal{L}$. 
Rewriting Semantics of IMP

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

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

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

**Q:** How can mechanize checking \( \mathcal{C}_{\mathcal{R}_{\text{IMP}}} \models Q \) (or, more generally, how can we mechanize checking \( \mathcal{C}_{\mathcal{R}_L} \models Q \)?)

**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 $skip(X, Y)$:

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

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

We would like to verify that whenever $X \geq Y \geq 0$ $swap(X, Y)$ terminates as $skip(Y, X)$. 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 \( SWAP \) \textup{upto a given loop bound}. For example, using Maude search, by letting \( X \geq Y \geq 0 \), we can verify \( SWAP \) \textup{upto} \( X \):

\[
\text{search } \text{swap}(X,Y) \Rightarrow! S \text{ such that } S = \text{skip}(Y,X).
\]

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
\( S \rightarrow \text{skip} | [o,10] [x,3] [y,10] \)

No more solutions.
states: 39  rewrites: 118
Verification by Reachability Logic

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

\begin{verbatim}
(select SWAP .) --- SWAP imports IMP
(use tool varsat for validity on SWAP .)
(def-term-set (skip | M:Memory) | true .)

(add-goal swap-interm : (swap | [x,X:Nat] [y,Y:Nat] [o,O:Nat]) |
 (Y:Nat <=Nat O:Nat) = (true) /
 (O:Nat <=Nat X:Nat + Y:Nat) = (true)
=> (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-prop : (swap | [x,X:Nat] [y,Y:Nat] [o,X:Nat]) |
 (Y:Nat <=Nat X:Nat) = (true)
=> (skip | [x,X’:Nat] [y,Y’:Nat] [o,X:Nat]) |
 (X’:Nat) = (Y:Nat) /
 (Y’:Nat) = (X:Nat) .)

(start-proof .)
(on 2 use strat swap-interm .)
(auto* .)
\end{verbatim}