Proofs: Program Correctness

One of the great advantages of formal semantics is that we can actually prove that a program will behave correctly for *all* expected input values.

In order for this to work we need the notion of a *program specification*.

The program specification act as the *yard stick* for the expected program behavior for any set of input values.

Program Specifications

$\Rightarrow$ A program specification is a universally quantified sentence over states in first order logic.

Consider the following program specification for some program p and variables x and y:

$$\begin{aligned}\forall s, \exists Q, VX, VY \quad & [(p, s) \longrightarrow\!\!\gg Q \wedge \\ & \mathrm{lookup}(y, s, VY) \wedge \mathrm{lookup}(x, Q, VY) \wedge \\ & \mathrm{lookup}(x, s, VX) \wedge \mathrm{lookup}(y, Q, VX)]\end{aligned}$$

This specification states that running the program p in state s will give rise to some state Q. Furthermore, looking up the variable y in state s is the same as looking up the variable x in state Q and *vice versa*.

This is a program specification for a *swap* program that swaps the values of x and y.

Program Specifications

Now, consider the program p written in our simple language IMP defined in 'sem.pl':

$$p \equiv \textbf{assign}(t, x) \textbf{ seq } \textbf{assign}(x, y) \textbf{ seq } \textbf{assign}(y, t)$$

Without formal semantics and a program specification we would simply try "a bunch" of values, and if the results look good we would infer that the program works. But there will always be a doubt that it will work for all states since trying a bunch of values does not constitute a proof.

However, given our formal semantics we can prove that this program *satisfies* the specification and therefore we can prove that the program works for all possible states.

Program Specifications

```
% swap.pl
:-['sem.pl'].

:- >>> 'show that program P="assign(t,x) seq assign(x,y) seq assign(y,t))"'.
:- >>> 'satisfies the program specification:'.
:- >>> ' (P,s)-->>Q,lookup(y,s,VY),lookup(x,Q,VY),lookup(x,s,VX),lookup(y,Q,VX)'.

program(assign(t,x) seq assign(x,y) seq assign(y,t)).

:- asserta(lookup(x,s,vx)).
:- asserta(lookup(y,s,vy)).

:- program(P),
     (P,s) -->> Q,
     lookup(y,s,VY),
     lookup(x,Q,VY),
     lookup(x,s,VX),
     lookup(y,Q,VX).
```

Program Specifications

Now consider the program specification

$$\forall s, \exists Q, V1, V2 \quad \begin{array}{l}[(p, s) \longrightarrow\!\!\gg Q \wedge \\ \text{lookup}(z, s, V1) \wedge \text{lookup}(z, Q, V2) \wedge \\ V2 = 2 * V1]\end{array}$$

It is easy to see that the program $p \equiv \textbf{assign}(z, \textbf{mult}(2, z))$ satisfies the specification.

But so does this program $p \equiv \textbf{assign}(z, \textbf{add}(z, z))$.

$\Rightarrow$ Program specifications are *implementation independent*!

Program Specifications

```
% double.pl
:-['sem.pl'].

:- >>> 'show that program P="assign(z,add(z,z))"'.
:- >>> 'satisfies the program specification:'.
:- >>> ' (p,s) -->> Q,lookup(z,s,V1),lookup(z,Q,V2),V2 = 2*V1'.

program(assign(z,add(z,z))).

:- asserta(lookup(z,s,vz)).
:- asserta(2*I xis I+I). % property of integers

:- program(P),
      (P,s) -->> Q,
      lookup(z,s,V1),
      lookup(z,Q,V2),
      V2 = 2 * V1.
```