Review of proving programs correct that have loops and in-determinants.

Pre- and post-conditions for a program with a loop

$\{\text{pre}\}$
init **seq**
$\{\text{inv}\}$
while b **do**
    $\{\text{inv} \wedge b\}$
    c
    $\{\text{inv}\}$
$\{\text{inv} \wedge \neg b\}$
$\{\text{post}\}$

NOTE: *We now have pre- and postconditions for each statement in this iterative program.* These conditions will hold in all iterations of the loop.

This gives us a new proof rule for *partial correctness* of loops:

if

$\{\text{pre}\}\ \text{init}\ \{\text{inv}\}\ \wedge$
$\{\text{inv} \wedge b\}\ c\ \{\text{inv}\}\ \wedge$
$\text{inv} \wedge \neg b \Rightarrow \text{post}$

then

'init **seq while** *b* **do** *c* **end**' *is correct*

NOTE: We call this partial correctness because we make no assertions about termination. All we assert is that, if the computation terminates, then it will be correct.

Program Correctness & Iteration

Or written in our notation, partial correctness of loops:

if

$$(\mathrm{init}, S) \longrightarrow\!\!\!\gg Q \wedge [\mathrm{pre}(S) \Rightarrow \mathrm{inv}(Q)] \wedge$$
$$(c, S) \longrightarrow\!\!\!\gg Q \wedge (b, S) \longrightarrow\!\!\!\gg B \wedge [(\mathrm{inv}(S) \wedge B) \Rightarrow \mathrm{inv}(Q)] \wedge$$
$$(b, T) \longrightarrow\!\!\!\gg B \wedge [(inv(T) \wedge \neg B) \Rightarrow post(T)]$$

then

'init **seq while** *b* **do** *c* **end**' *is correct*

Program Correctness & Iteration

```
% pow-n-loop.pl

:- ['sem-func.pl'].

:- >>> 'prove that the program p:'.
:- >>> '        var(i) seq'.
:- >>> '        var(z) seq'.
:- >>> '        assign(i,1) seq'.
:- >>> '        assign(z,m) seq'.
:- >>> '        whiledo(not(eq(i,n)),'.
:- >>> '                assign(i,add(i,1)) seq'.
:- >>> '                assign(z,mult(z,m))'.
:- >>> '        )'.
:- >>> 'is correct for any value of m and n with n>0'.
:- >>> 'pre(R) = initialstate(env([bind(vm,m),bind(vn,n)],s))'.
:- >>> 'post(T) = lookup(z,T,vm^vn)'.
:- >>> 'inv(Q) = lookup(i,Q,vi) ^ lookup(z,Q,vm^vi)'.

:- >>> 'define the parts of our program'.
init((var(i) seq var(z) seq assign(i,1) seq assign(z,m))).
guard(not(eq(i,n))).
body((assign(i,add(i,1)) seq assign(z,mult(z,m)))).

:- >>> 'define a model for our power operation'.
:- dynamic pow/3.
pow(B,1,B).

pow(B,P,R) :-
    T1 is P-1,
    pow(B,T1,T2),
    R is B*T2.
```

Program Correctness & Iteration

```
:- >>> 'first proof obligation'.
:- >>> 'assume precondition'.
:- asserta(initialstate(env([bind(vm,m),bind(vn,n)],s))).
:- >>> 'prove the invariant'.
:- init(P),initialstate(IS),(P,IS) -->> Q,lookup(i,Q,VI),lookup(z,Q,VZ),pow(vm,VI,VZ).
:- retract(initialstate(env([bind(vm,m),bind(vn,n)],s))).

:- >>> 'second  proof obligation'.
:- >>> 'assume invariant on s'.
:- asserta(initialstate(env([bind(vi,i),bind(vz,z),bind(vm,m),bind(vn,n)],s))).
:- asserta(pow(vm,vi,vz)).
% implies
:- asserta(pow(vm,vi+1,vz*vm)).
:- >>> 'assume guard on s'.
:- asserta((not(eq(i,n)),s) -->> true).
:- >>> 'prove the invariant on Q'.
:- body(Bd),initialstate(IS),(Bd,IS) -->> Q,lookup(i,Q,VI),lookup(z,Q,VZ),pow(vm,VI,VZ).
:- retract(initialstate(env([bind(vi,i),bind(vz,z),bind(vm,m),bind(vn,n)],s))).
:- retract(pow(vm,vi,vz)).
:- retract(pow(vm,vi+1,vz*vm)).
:- retract((not(eq(i,n)),s) -->> true).
```

Program Correctness & Iteration

```
:- >>> 'third  proof obligation'.
:- >>> 'assume the invariant on s'.
:- asserta(initialstate(env([bind(vi,i),bind(vz,z),bind(vm,m),bind(vn,n)],s))).
:- asserta(pow(vm,vi,vz)).
:- >>> 'assume NOT guard on any s'.
:- asserta((not(eq(i,n)),s) -->> not(true)).
% implies
:- asserta((eq(i,n),s) -->> true).
% implies
:- asserta(pow(vm,vn,vz)).
:- >>> 'prove postcondition on s'.
:- initialstate(IS),lookup(z,IS,VZ),pow(vm,vn,VZ).
:- retract(initialstate(env([bind(vi,i),bind(vz,z),bind(vm,m),bind(vn,n)],s))).
:- retract(pow(vm,vi,vz)).
:- retract((not(eq(i,n)),s) -->> not(true)).
:- retract((eq(i,n),s) -->> true).
:- retract(pow(vm,vn,vz)).
```

Program Correctness & Recursive Function Calls

The key insights to proving programs with recursive function correct is that

- There exist at least one variable in argument list of the recursive function we call the *recursion variable* which measures the progress of the recursion.
- The body of the recursive function consists of at least two separate parts:
  - the part of the body that gets executed when recursion terminates (the base case)
  - the part of the body that gets executed during normal recursion (the recursive step)
- We can always identify an "accumulator variable" that contains the partially computed result at any particular step in the recursion.

Program Correctness & Recursive Function Calls

We introduce two new predicates for the correctness proof:

- callcond - describes the current condition within a called function
- finv - this is the *function invariant* and is similar to the loop invariant, it describes the condition at the "accumulator variable"

Program Correctness & Recursive Function Calls

This allows us to state the following proof schema:

```
function f (i)
   if (<term cond>)
      {callcond ^ term cond}
      return = <base case value>
      {finv}
   else
      {callcond ^ not term cond}
      return = <local comp> + f(i-1)
      {finv}

{pre ^ finv}
z = f(k)
{post}
```

⇒

```
function f (i)
   if (<term cond>)
      {callcond ^ term cond}
      body1
      {finv}
   else
      {callcond ^ not term cond}
      body2
      {finv}

{pre ^ finv}
final
{post}
```

Proof Obligations:

1. {callcond $\wedge$ termination condition} body1 {finv}
2. {callcond $\wedge$ $\neg$ termination condition} body2 {finv}
3. {pre $\wedge$ finv} final {post}

Program Correctness & Recursive Function Calls

```
% pow-n-func.pl
:- ['sem-func.pl'].
:- >>> 'prove that the program p:'.
:- >>> '     (fun(pow,'.
:- >>> '             [b,p],'.
:- >>> '             var(result) seq'.
:- >>> '             if(eq(p,1),'.
:- >>> '                assign(result,b),'.
:- >>> '                assign(result,mult(b,call(pow,[assign(b,b),assign(p,sub(p,1))]))),'.
:- >>> '             result) seq'.
:- >>> '         var(z) seq'.
:- >>> '         assign(z,call(pow,[assign(b,m),assign(p,n)]))'.
:- >>> 'is correct for any value of n and m and n>0'.
:- >>> 'pre(R) = initialstate(env([bind(vm,m),bind(vn,n)],s))'.
:- >>> 'post(T) = lookup(z,T,vm^vn)'.
:- >>> 'inv(Q) = lookup(b,Q,vb) ^ lookup(p,Q,vp) ^ lookup(result,Q,vb^vp)'.

:- >>> 'define the parts of our program'.
guard(eq(p,1)).
body1(var(result) seq assign(result,b)).
body2(var(result) seq assign(result,mult(b,call(pow,[assign(b,b),assign(p,sub(p,1))])))).
final(var(z) seq assign(z,result)).

:- >>> 'define a model for our power operation'.
:- dynamic pow/3.
pow(B,1,B).

pow(B,P,R) :-
    T1 is P-1,
    pow(B,T1,T2),
    R is B*T2.
```

Program Correctness & Recursive Function Calls

```
:- >>> 'Proof by case analysis on recursion variable p'.
:- >>> 'first proof obligation'.
:- >>> 'assume call condition -- guard is true'.
:- asserta(initialstate(env([bind(vb,b),bind(1,p)],s))).
:- >>> 'prove the invariant'.
:- body1(P),initialstate(IS),(P,IS)-->>Q,lookup(b,Q,VB),lookup(p,Q,VP),lookup(result,Q,VR),pow(VB,VP,VR).
:- retract(initialstate(env([bind(vb,b),bind(1,p)],s))).

:- >>> 'second proof obligation'.
:- >>> 'assume call condition -- guard is false: vp =/= 1'.
:- asserta(initialstate(env([bind(vb,b),bind(vp,p)],s))).
% assume that the recursive call returns a value k
:- asserta((call(pow,[assign(b,b),assign(p,sub(p,1))]),S) -->> (k,S)).
% where the value k is defined as
:- asserta(pow(vb,vp-1,k)).
% which implies
:- asserta(pow(vb,vp,vb*k)).
:- >>> 'prove the invariant'.
:- body2(P),initialstate(IS),(P,IS)-->>Q,lookup(b,Q,VB),lookup(p,Q,VP),lookup(result,Q,VR),pow(VB,VP,VR).
:- retract(initialstate(env([bind(vb,b),bind(vp,p)],s))).
:- retract((call(pow,[assign(b,b),assign(p,sub(p,1))]),S) -->> (k,S)).
:- retract(pow(vb,vp-1,k)).
:- retract(pow(vb,vp,vb*k)).
```

```
:- >>> 'third proof obligation'.
:- >>> 'assume precondition -- result = q = vm^vn'.
:- >>> '   this follows from the second proof obligation with vp = vn'.
:- asserta(initialstate(env([bind(vm,m),bind(vn,n),bind(q,result)],s))).
:- asserta(pow(vm,vn,q)).
:- >>> 'prove the post condition'.
:- final(P),initialstate(IS),(P,IS)-->>Q,lookup(z,Q,VZ),pow(vm,vn,VZ).
:- retract(initialstate(env([bind(vm,m),bind(vn,n),bind(q,result)],s))).
:- retract(pow(vm,vn,q)).
```