Software Verification
Exercise Solution: Software Model Checking

The routine we consider is:

always_positive (x: INTEGER): INTEGER
    if x > 0 **then**
        Result := x + x
    else
        if x = 0 **then**
            Result := 1
        else
            Result := x * x
        end
    end
ensure Result > 0 **end**

(a) We build the predicate abstraction of *always_positive* in an incremental fashion.

1. Normalize the conditions appearing in conditionals and loops. We get

   always_positive_1 (x: INTEGER): INTEGER
       if ? **then**
           assume x > 0
           Result := x + x
       else
           assume x <= 0
           if ? **then**
               assume x = 0
               Result := 1
           else
               assume x /= 0
               Result := x * x
           end
       end
   ensure Result > 0 **end**

2. We rewrite the assume statements, and apply common simplifications to the logical formulae as well as peephole optimizations.
   i. For **assume** x > 0:

¬ Pred(¬ x > 0) = ¬ Pred(x <= 0) = ¬ ¬ pos = pos
So we will add **assume** pos.

We must also take into account the effect of the assume statement on pos and Rpos:

For pos:
wp(**assume** x > 0, x > 0) = (x > 0 => x > 0) = **True**
Pred(**True**) = **True**
So we must include the update
if True then pos := **True else if** ... **else** ... **end**,
which simplifies to pos := **True**.

For Rpos:
wp(**assume** x > 0, **Result** > 0) = (x > 0 => **Result** > 0)
Pred(x > 0 => **Result** > 0) = (pos => Rpos).
Similarly, wp(**assume** x > 0, **Result** <= 0) = (x > 0 => **Result** <= 0)
Pred(x > 0 => **Result** <= 0) = (pos => ¬ Rpos)
So we get
if pos => Rpos **then**
  Rpos := **True**
else if pos => ¬ Rpos **then**
  Rpos := **False**
else Rpos := ?
end
Since we just assumed pos in the code, we can apply the peephole optimization and remove this update, since it will have no effect on the value of Rpos.

Hence **assume** x > 0 becomes **assume** pos; pos := **True** in the abstraction.

ii. For **assume** x <= 0:
¬ Pred(¬ x <= 0) = ¬ Pred(x > 0) = ¬ pos
So we will add **assume** ¬ pos in the abstraction.

The effect on pos:
wp(**assume** x <= 0, x > 0) = (x <= 0 => x > 0) = x > 0
Pred(x > 0) = pos.
Similarly, wp(**assume** x <= 0, x <= 0) = (x <= 0 => x <= 0) = **True**
Pred(**True**) = **True**.
So we have
if pos **then**
  pos := **True**
else if True then
  pos := **False**

else pos := ? **end**
Since we just assumed ¬ pos in the abstraction, we can simplify this update to pos := **False**.

The effect on Rpos:
wp(**assume** x <= 0, **Result** > 0) = (x <= 0 => **Result** > 0)
Pred(x <= 0 => **Result** > 0) = (¬ pos => Rpos) = Rpos because of a peephole optimization.
wp(**assume** x <= 0, **Result** <= 0) = (x <= 0 => **Result** <= 0)
Pred(x <= 0 => **Result** <= 0) = (¬ pos => ¬ Rpos) = ¬ Rpos because of the same peephole optimization.
So we have
if Rpos **then**
    Rpos := **True**
else if ¬ Rpos **then**
    Rpos := **False**
else Rpos := ? **end**
which can be eliminated.

Hence **assume** x <= 0 becomes **assume** ¬ pos; pos := **False**.

iii. For **assume** x = 0:
¬ Pred(¬ x = 0) = ¬ pos
So we will add **assume** ¬ pos in the abstraction.

The effect on pos:
wp(**assume** x = 0, x > 0) = (x = 0 => x > 0) = (x /= 0)
Pred(x /= 0) = pos
Similarly, wp(**assume** x = 0, x <= 0) = (x = 0 => x <= 0) = **True**
Pred(**True**) = **True**
So we have the update:
if pos **then**
    pos := **True**
else if True then
    pos := **False**
else pos := ? **end**
which becomes pos := **False** when we do a peephole simplification.

The effect on Rpos:
wp(**assume** x = 0, **Result** > 0) = (x = 0 => **Result** > 0)
Pred(x = 0 => **Result** > 0) = pos ∨ Rpos
Similarly, wp(**assume** x = 0, **Result** <= 0) = (x = 0 => **Result** <= 0)
Pred(x = 0 => **Result** <= 0) = pos ∨ ¬ Rpos
Applying peephole optimization, we see that the update will not have any effect and can be removed.

Hence **assume** x = 0 becomes **assume** ¬ pos; pos := **False**.

iv. For **assume** x /= 0:
¬ Pred(¬ x /= 0) = ¬ Pred(x = 0) = ¬ **False** = **True**
So we do not need to add an assume statement to the abstraction.

The effect on pos:
wp(**assume** x /= 0, x > 0) = (x /= 0 => x > 0) = (x >= 0)
Pred(x >= 0) = pos
wp(**assume** x /= 0, x <= 0) = (x /= 0 => x <= 0) = (x <= 0)
Pred(x <= 0) = ¬ pos
So the assume has no effect on the value of pos.

The effect on Rpos:
wp(**assume** x /= 0, **Result** > 0) = (x /= 0 => **Result** > 0)
Pred(x /= 0 => **Result** > 0) = Rpos
wp(**assume** x /= 0, **Result** <= 0) = (x /= 0 => **Result** <= 0)
Pred(x /= 0 => **Result** <= 0) = ¬ Rpos
So **assume** x /= 0 has no effect on the value of Rpos.

Hence **assume** x /= 0 becomes **skip**.

After transforming the assume statements, we also abstract the postcondition and get:

```
always_positive_2 (x: INTEGER): INTEGER
    if ? then
        assume pos;  pos := True
        Result := x + x
    else
        assume ¬ pos;  pos := False
        if ? then
            assume ¬ pos;  pos := False
            Result := 1
        else
            skip
            Result := x * x
        end
    end
ensure Rpos end
```

3. Lastly, we transform the assignment statements.
    i. The assignment **Result** := x + x.

The effect on pos:
wp(**Result** := x + x, x > 0) = (x > 0)
Pred(x > 0) = pos
wp(**Result** := x + x, x <= 0) = (x <= 0)
Pred(x <= 0) = ¬ pos
So the assignment has no effect on pos.

The effect on Rpos:
wp(**Result** := x + x, **Result** > 0) = (x + x > 0)
Pred(x + x > 0) = pos
wp(**Result** := x + x, **Result** <= 0) = (x + x <= 0)
Pred(x + x <= 0) = ¬ pos
So we have the update:
if pos **then**
   Rpos := **True**
else if ¬ pos **then**
   Rpos := **False**
else Rpos := ?
end
which can be simplified by a peephole optimization to Rpos := **True**.

Hence **Result** := x + x is transformed into Rpos := **True**.

ii. The assignment **Result** := 1.

The effect on pos:
wp(**Result** := 1, x > 0) = (x > 0)
Pred(x > 0) = pos
wp(**Result** := 1, x <= 0) = (x <= 0)
Pred(x <= 0) = ¬ pos
So **Result** := 1 has no effect on pos.

The effect on Rpos:
wp(**Result** := 1, **Result** > 0) = (1 > 0) = **True**
Pred(**True**) = **True**
So **Result** := 1 has the effect Rpos := **True**.

Hence **Result** := 1 is transformed into Rpos := **True**.

iii. The assignment **Result** := x * x.

The effect on pos:
wp(**Result** := x * x, x > 0) = (x > 0)
Pred(x > 0) = pos
wp(**Result** := x * x, x <= 0) = (x <= 0)

Pred(x <= 0) = ¬ pos
So **Result** := x * x has no effect on pos.

The effect on Rpos:
wp(**Result** := x * x, **Result** > 0) = (x * x > 0) = (x /= 0)
Pred(x /= 0) = pos
wp(**Result** := x * x, **Result** <= 0) = (x * x <= 0) = (x = 0)
Pred(x = 0) = **False**
The update:
if pos **then**
  Rpos := **True**
else if False then
  Rpos := **False**
else Rpos := ? **end**
can be simplified with a peephole optimization to become Rpos := ?.

Hence **Result** := x * x is transformed into Rpos := ?.

The resulting abstraction looks as follows:

always_positive_3 (x: INTEGER): INTEGER
  if ? **then**
    assume pos; pos := **True**
    Rpos := **True**
  else
    assume ¬ pos; pos := **False**
    if ? **then**
      assume ¬ pos; pos := **False**
      Rpos := **True**
    else
      skip
      Rpos := ?
    end
  end
ensure Rpos **end**

(b) No, we cannot verify the abstraction *always_positive_3*.

Here is a counterexample run:
{¬ pos, ¬ Rpos}
[¬ ?]
{¬ pos, ¬ Rpos}
assume ¬ pos; pos := **False**
{¬ pos, ¬ Rpos}
[¬ ?]

{¬ pos, ¬ Rpos}
Rpos := ?
{¬ pos, ¬ Rpos}

It corresponds to the following concrete run, for which we computed the weakest precondition with respect to x <= 0 /\ **Result** <= 0:

{x <= 0 /\ x^2 <= 0 /\ ¬ x = 0 /\ ¬ x > 0}   // Equivalent to **False**.
[¬ x > 0]
{x <= 0 /\ x^2 <= 0 /\ ¬ x = 0}   // Equivalent to **False**.
[¬ x = 0]
{x <= 0 /\ x^2 <= 0}   // Equivalent to x = 0
Result := x * x
{x <= 0 /\ **Result** <= 0}

Since two of the concrete assertions are not satisfiable (equivalent to **False**), we know that the conjunctions of these assertions will their abstract counterparts will not be satisfiable either. Hence the abstract run is spurious, and calls for abstraction refinement.

Cheat sheet for the translation of assume statements and assignments

Assume statements

A statement   assume ex end
gets translated into   assume ¬(Pred(¬ex)) end
plus a parallel assignment such that:
For each p(i),
  if ex => e(i),   add p(i) := True to the parallel assignment
  if ex => ¬e(i),   add p(i) := False to the parallel assignment
  else omit p(i) from the parallel assignment.
This parallel assignment is equivalent to the sequence of individual assignments, because all of them assign constants to different variables.

Assignments

If an assignment statement does not modify any of e(i)'s free variables, then the assignment will leave p(i) unchanged.