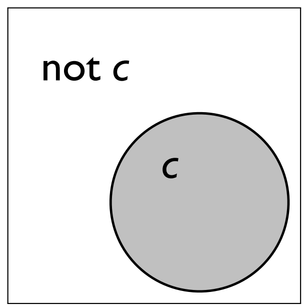
Problem Sheet 9: Software Model Checking
Sample Solutions

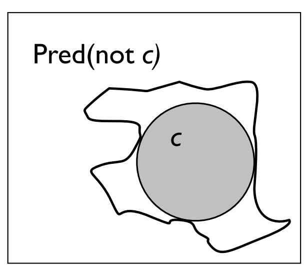
Chris Poskitt*
ETH Zürich

1 Predicate Abstraction

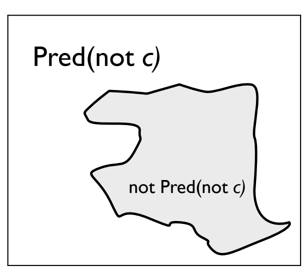
i. Let us first visualise c and `not` c in a Venn diagram:



$Pred$(`not` c) gives the weakest under-approximation of `not` c. In other words, $Pred$(`not` c) implies `not` c, but not (in general) the converse. A possible visualisation in a Venn diagram might then be:



In negating $Pred$(`not` c), we then get the strongest over-approximation, visualised as follows:



*Some exercises adapted from ones written by Stephan van Staden and Carlo A. Furia.

ii. We build a Boolean abstraction from C_1, one line at a time. First, we over-approximate `assume` $\texttt{x} > 0$ **`end`** with `assume` $\neg Pred(\neg \texttt{x} > 0)$ **`end`**, followed by a parallel conditional assignment updating the predicates with respect to the original `assume` statement.

$$\begin{aligned}\neg Pred(\neg \texttt{x} > 0) &= \neg(\neg p)\\ &= p\end{aligned}$$

Hence we add `assume` p **`end`** to A_1. This should be followed by a parallel conditional assignment (as described in the slides):

```
if Pred(+ex(i)) then
     p(i) := True
elseif Pred(-ex(i)) then
     p(i) := False
else
     p := ?
end
```

Using the axiom $\vdash \{c \Rightarrow post\}$ `assume` c **`end`** $\{post\}$ for the weakest precondition of assume statements, we compute every $+/-ex(i)$ for predicates i:

$$\begin{aligned}+ex(p) &= (x > 0 \Rightarrow x > 0)\\ -ex(p) &= (x > 0 \Rightarrow \neg x > 0)\\ +ex(q) &= (x > 0 \Rightarrow y > 0)\\ -ex(q) &= (x > 0 \Rightarrow \neg y > 0)\\ +ex(r) &= (x > 0 \Rightarrow z > 0)\\ -ex(r) &= (x > 0 \Rightarrow \neg z > 0)\end{aligned}$$

We apply the simplification step from the slides, and omit each $Pred(ex(i))$ that is not unconditionally valid. It so happens that only:

$$Pred(+ex(p)) = Pred(x > 0 \Rightarrow x > 0) = Pred(\text{true}) = \text{true}$$

is valid, hence the parallel conditional assignment reduces to simply:

```
if True then
     p := True
else
     p := ?
end
```

This reduces even further to `p := True`, which we add to A_1.

Next, we address the assignment `z := (x * y) + 1`. Recall that an assignment $x := f$ is over-approximated by a parallel conditional assignment:

```
if Pred(+f(i)) then
     p(i) := True
elseif Pred(-f(i)) then
     p(i) := False
else
     p := ?
end
```

Using the axiom $\vdash \{post[f/x]\}\ x := f\ \{post\}$ and the definition of $+/-f(i)$ for predicates i, we get:

$$
\begin{aligned}
Pred(+f(p)) &= Pred(x > 0) \\
&= p \\
Pred(-f(p)) &= Pred(\neg x > 0) \\
&= \neg p \\
Pred(+f(q)) &= Pred(y > 0) \\
&= q \\
Pred(-f(q)) &= Pred(\neg y > 0) \\
&= \neg q \\
Pred(+f(r)) &= Pred((x * y) + 1 > 0) \\
&= (p \wedge q) \vee (\neg p \wedge \neg q) \\
Pred(-f(r)) &= Pred(\neg (x * y) + 1 > 0) \\
&= Pred((x * y) + 1 \leq 0) \\
&= \text{false}
\end{aligned}
$$

The parallel conditional assignments for p, q have no effect, hence we add only the following to A_1:

```
if (p and q) or (not p and not q) then
     r := True
elseif False then
     r := False
else
     r := ?
end
```

Finally, we address the assertion `assert z >= 1 end`. The Boolean abstraction is simply `assert` $Pred(z \geq 1)$ `end`. We have:

$$Pred(z \geq 1) = r$$

and hence add `assert` r `end` to A_1.

Altogether, A_1 is the following program:

```
assume p end
p := True

if (p and q) or (not p and not q) then
     r := True
elseif False then
     r := False
else
     r := ?
end

assert r end
```

With a further simplification, we get:

```
assume p end
p := True

if (p and q) or (not p and not q) then
     r := True
else
     r := ?
end

assert r end
```

iii. (a) After normalising the program (following the details in the slides) we get:

```
if ? then
     assume x > 0 end
     y := x + x
else
     assume x <= 0 end
     if ? then
          assume x = 0 end
          y := 1
     else
          assume x /= 0 end
          y := x * x
     end
end
assert y > 0 end
```

(b) To build A_2 from the normalised code above, apply the transformations to each assignment, assume, and assert, analogously to how I did when constructing A_1 (except that this time you only have two predicates, p and q). The resulting abstraction (after some simplifications) should be equivalent to this:

```
if ? then
     assume p end
     p := True

     q := True
else
     assume not p end
     p := False
     if ? then
          assume not p end
          p := False

          q := True
     else
          assume True end -- can delete this assume

          q := ?
     end
end
assert q end
```

2 Error Traces

i. An abstract error trace is:

```
[p, not q, r]
     assume p end
[p, not q, r]
     p := True
[p, not q, r]
     r := ?
[p, not q, not r]
     assert r end
```

Observe that each concrete instruction corresponds to a (compound) abstract instruction. We can check whether or not this is a feasible concrete run by computing the weakest precondition of the concrete instructions with respect to $p \wedge \neg q \wedge \neg r$, interpreting conditions (assume, conditionals, or exit conditions) as asserts:

```
{x > 0 and y <= 0 and (x*y)+1 <= 0}
{(x > 0 and y <= 0 and (x*y)+1 <= 0) and x > 0}
     assert x > 0 end
{x > 0 and y <= 0 and (x*y)+1 <= 0}
     z := (x*y) + 1
{x > 0 and y <= 0 and z <= 0}
[p, not q, not r]
```

Executing the concrete program on a state s such that

$$s \models x > 0 \wedge y \leq 0 \wedge (x * y) + 1 \leq 0$$

will reveal the fault. One possible input state (of many) is $s = \{x \mapsto 3, y \mapsto -2, z \mapsto \_\}$.

ii. Here is an abstract counterexample trace:

```
[not p, not q]
     assume not p end
[not p, not q]
     p := False
[not p, not q]
     assume True end
[not p, not q]
     q := ?
[not p, not q]
     assert q end
```

As before, we check whether or not this abstract execution reflects a feasible, concrete counterexample, by computing the weakest precondition of the corresponding concrete instructions with respect to $\neg p \wedge \neg q$. Again, we interpret conditions (assume in this case) as asserts, and apply the corresponding Hoare logic axioms:

```
{x < 0 and x*x <= 0}
{x <= 0 and x /= 0 and x <= 0 and x*x <= 0}
     assert x <= 0
{x /= 0 and x <= 0 and x*x <= 0}
     assert x /= 0 end
{x <= 0 and x*x <= 0}
     y := x*x
{x <= 0 and y <= 0}
[not p, not q]
```

Observe that in this case, the weakest precondition we have constructed is equivalent to false. There is no assignment to `x` that will satisfy the assertion. Hence the abstract counterexample is infeasible (spurious) in the concrete program; abstraction refinement is needed.