Problem Sheet 9: Software Model Checking
Sample Solutions

Chris Poskitt*
ETH Zürich

1 Predicate Abstraction

i. Let us first visualise $c$ and $\neg c$ in a Venn diagram:

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

In negating $Pred(\neg c)$, 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
\begin{verbatim}
assume x > 0 end
\end{verbatim}
with
\begin{verbatim}
assume \neg Pred(\neg x > 0) end
\end{verbatim}
followed by a parallel conditional assignment updating the predicates with respect to the original assume statement.

\[-Pred(\neg x > 0) = \neg \neg \neg p
= \neg p
= p\]

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

\begin{verbatim}
if Pred(+ex(i)) then
  p(i) := True
elseif Pred(-ex(i)) then
  p(i) := False
else
  p := ?
end
\end{verbatim}

Using the rule $\vdash \{ ex \Rightarrow post \} assume \{ post \}$ for the weakest precondition of assume statements, we compute every $ex(i)$ (as defined in the slides):

\begin{align*}
+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{align*}

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

\begin{verbatim}
Pred(+ex(p)) = Pred(x > 0 \Rightarrow x > 0) = Pred(true) = true
\end{verbatim}

is valid, hence the parallel conditional assignment reduces to simply $p := True$, which we add to $A_1$.

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

\begin{verbatim}
if Pred(+f(i)) then
  p(i) := True
elseif Pred(-f(i)) then
  p(i) := False
else
  p := ?
end
\end{verbatim}
Using the rule \( \vdash \{ \text{post}[f/x] \} \; x := f \; \{ \text{post} \} \) and the definition of \( f(i) \) from the slides, we get:

\[
\begin{align*}
\text{Pred}(+f(p)) &= \text{Pred}(x > 0) \\
&= p \\
\text{Pred}(-f(p)) &= \text{Pred}(\neg x > 0) \\
&= \neg p \\
\text{Pred}(+f(q)) &= \text{Pred}(y > 0) \\
&= q \\
\text{Pred}(-f(q)) &= \text{Pred}(\neg y > 0) \\
&= \neg q \\
\text{Pred}(+f(r)) &= \text{Pred}((x \ast y) + 1 > 0) \\
&= (p \land q) \lor (\neg p \land \neg q) \\
\text{Pred}(-f(r)) &= \text{Pred}(\neg(x \ast y) + 1 > 0) \\
&= \text{Pred}((x \ast y) + 1 \leq 0) \\
&= \text{false}
\end{align*}
\]

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

\[
\begin{align*}
&\text{if} \; (p \text{ and } q) \; \text{or} \; (\text{not } p \text{ and not } q) \; \text{then} \\
&\quad \text{r} := \text{True} \\
&\quad \text{elseif False then} \\
&\quad \quad \text{r} := \text{False} \\
&\quad \text{else} \\
&\quad \quad \text{r} := ? \\
&\end{align*}
\]

Finally, we address the assertion \textbf{assert} \( z >= 1 \) \textbf{end}. This is analogous to the abstraction of assume statements, except that we add \textbf{assert} \( \neg \text{Pred}(\neg z >= 1) \) \textbf{end} followed by a parallel conditional assignment with each \textit{ex(i)} constructed using the rule \( \vdash \{ \text{exp} \land \text{post} \} \; \text{assert} \; \text{exp} \; \textbf{end} \; \{ \text{post} \} \). We have:

\[
\neg \text{Pred}(\neg z >= 1) = \neg \text{Pred}(z < 1) = \neg \neg r = r
\]

and hence add \textbf{assert} \( r \) \textbf{end} to \( A_1 \).
Given that \( r \) is asserted immediately before, the parallel conditional assignment will have no effect on the values of \( p, q, r \) and so we omit it from \( A_1 \). Altogether, \( A_1 \) is the following program:

```plaintext
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:

```plaintext
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) looks as follows:

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
assert q end

2 Error Traces

i. An abstract error trace is, for example:

[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 $p \land \neg q \land \neg r$, interpreting conditions (assume, conditionals, or exit conditions) as assert:

\[
\begin{align*}
\{ & x > 0 \text{ and } y \leq 0 \text{ and } (x*y)+1 \leq 0 \\
& x > 0 \text{ and } x > 0 \text{ and } y \leq 0 \text{ and } (x*y)+1 \leq 0 \\
& \text{assert } x > 0 \text{ end} \\
& x > 0 \text{ and } y \leq 0 \text{ and } (x*y)+1 \leq 0 \\
& z := (x*y) + 1 \\
& \{ x > 0 \text{ and } y \leq 0 \text{ and } z \leq 0 \\
& [p, \text{not } q, \text{not } r]
\end{align*}
\]

Some witnesses to the fault are $x = 3, y = -2$ which satisfy the constructed weakest precondition.

ii. Here is an abstract counterexample trace:

\[
\begin{align*}
& [\text{not } p, \text{not } q] \\
& \text{assume not } p \text{ end} \\
& [\text{not } p, \text{not } q] \\
& p := \text{False} \\
& [\text{not } p, \text{not } q] \\
& \text{assume True end} \\
& [\text{not } p, \text{not } q] \\
& q := ? \\
& [\text{not } p, \text{not } q] \\
& \text{assert } q \text{ end}
\end{align*}
\]

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 $\neg p \land \neg q$. Again, we interpret conditions (assume in this case) as assert, and apply the corresponding Hoare proof rule:

\[
\begin{align*}
& \{ x < 0 \text{ and } x*x \leq 0 \\
& \{ x \leq 0 \text{ and } x \neq 0 \text{ and } x < 0 \text{ and } x*x \leq 0 \\
& \text{assert } x \leq 0 \\
& \{ x \neq 0 \text{ and } x < 0 \text{ and } x*x \leq 0 \\
& \text{assert } x \neq 0 \text{ end} \\
& \{ x \leq 0 \text{ and } x*x \leq 0 \\
& y := x*x \\
& \{ x \leq 0 \text{ and } y \leq 0 \\
& [\text{not } p, \text{not } q]
\end{align*}
\]

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.