1 Predicate Abstraction

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

\[
\begin{array}{c}
\neg c \\
\end{array}
\]

\[
\begin{array}{c}
c \\
\end{array}
\]

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

\[
\begin{array}{c}
\text{Pred}(\neg c) \\
\end{array}
\]

\[
\begin{array}{c}
c \\
\end{array}
\]

In negating \( \text{Pred}(\neg c) \), we then get the strongest over-approximation, visualised as follows:

\[
\begin{array}{c}
\text{Pred}(\neg c) \\
\neg \text{Pred}(\neg c) \\
\end{array}
\]

*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 \textbf{assume} $x > 0$ \textbf{end} with \textbf{assume} $\neg \text{Pred}(\neg x > 0)$ \textbf{end}, followed by a parallel conditional assignment updating the predicates with respect to the original \textbf{assume} statement.

$$\neg \text{Pred}(\neg x > 0) = \neg(\neg p) = p$$

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

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

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

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

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

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

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

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

This reduces even further to $p := \text{True}$, 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:

\[
\begin{align*}
\text{if } \text{Pred}(+f(i)) \text{ then } & \quad p(i) := \text{True} \\
\text{elseif } \text{Pred}(-f(i)) \text{ then } & \quad p(i) := \text{False} \\
\text{else } & \quad p := ? \\
\end{align*}
\]

Using the axiom \( \vdash \{\text{post}[f/x]\} \ x := f \ \{\text{post}\} \) and the definition of \( +/ - f(i) \) for predicates \( i \), 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 \( A_1 \):

\[
\begin{align*}
\text{if } (p \land q) \text{ or } (\neg p \land \neg q) \text{ then } & \quad r := \text{True} \\
\text{elseif } \text{False} \text{ then } & \quad r := \text{False} \\
\text{else } & \quad r := ? \\
\end{align*}
\]

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

\[
\text{Pred}(z \geq 1) = r
\]

and hence add \textbf{assert } r \text{ 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
elsif 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:

\[
\begin{align*}
&\text{if } ? \text{ then} \\
&\quad \text{assume } x > 0 \text{ end} \\
&\quad y := x + x \\
&\text{else} \\
&\quad \text{assume } x \leq 0 \text{ end} \\
&\quad \text{if } ? \text{ then} \\
&\quad\quad \text{assume } x = 0 \text{ end} \\
&\quad\quad y := 1 \\
&\quad\text{else} \\
&\quad\quad \text{assume } x \neq 0 \text{ end} \\
&\quad\quad y := x \times x \\
&\text{end} \\
&\text{end} \\
&\text{assert } y > 0 \text{ end}
\end{align*}
\]

(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:

\[
\begin{align*}
&\text{if } ? \text{ then} \\
&\quad \text{assume } p \text{ end} \\
&\quad p := \text{True} \\
&\quad q := \text{True} \\
&\text{else} \\
&\quad \text{assume not } p \text{ end} \\
&\quad p := \text{False} \\
&\quad \text{if } ? \text{ then} \\
&\quad\quad \text{assume not } p \text{ end} \\
&\quad\quad p := \text{False} \\
&\quad\quad q := \text{True} \\
&\quad\text{else} \\
&\quad\quad \text{assume True end -- can delete this assume} \\
&\quad\quad q := ? \\
&\text{end} \\
&\text{end} \\
&\text{assert } q \text{ end}
\end{align*}
\]
2 Error Traces

i. An abstract error trace is:

\[
[p, \neg q, r] \\
\text{assume } p \text{ end} \\
[p, \neg q, r] \\
p := \text{True} \\
[p, \neg q, r] \\
r := ? \\
[p, \neg q, \neg r] \\
\text{assert } r \text{ end}
\]

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 asserts:

\[
\{x > 0 \text{ and } y \leq 0 \text{ and } (x*y)+1 \leq 0\}
\]

\[
\{(x > 0 \text{ and } y \leq 0 \text{ and } (x*y)+1 \leq 0) \text{ and } x > 0\}
\]

\[
\text{assert } x > 0 \text{ end}
\]

\[
\{x > 0 \text{ and } y \leq 0 \text{ and } z \leq 0\}
\]

\[
[p, \neg q, \neg r]
\]

Executing the concrete program on a state \(s\) such that

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

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:

\[
[\neg p, \neg q] \\
\text{assume } \neg p \text{ end} \\
[\neg p, \neg q] \\
p := \text{False} \\
[\neg p, \neg q] \\
\text{assume } \text{True} \text{ end} \\
[\neg p, \neg q] \\
q := ? \\
[\neg p, \neg q] \\
\text{assert } q \text{ end}
\]

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 \(\neg p \land \neg q\). Again, we interpret conditions (assume in this case) as asserts, and apply the corresponding Hoare logic axioms:
{\( x < 0 \) and \( x \times x \leq 0 \)}
{\( x \leq 0 \) and \( x \neq 0 \) and \( x \leq 0 \) and \( x \times x \leq 0 \)}
\quad assert \( x \leq 0 \)
{\( x \neq 0 \) and \( x \leq 0 \) and \( x \times x \leq 0 \)}
\quad assert \( x \neq 0 \) end
{\( x \leq 0 \) and \( x \times x \leq 0 \)}
\quad y := x \times x
{\( x \leq 0 \) and \( y \leq 0 \)}
\quad \text{[\text{not } p, \text{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.