Problem Sheet 1: Axiomatic Semantics
Sample Solutions

Chris Poskitt
ETH Zürich

Starred exercises (*) are more challenging than the others.

1 Partial and Total Correctness

i. Partial correctness: if program $P$ is executed on a state satisfying $pre$, then if its execution terminates, the state returned will satisfy $post$.

ii. Total correctness: partial correctness (as above), but termination of $P$ on states satisfying $pre$ is also guaranteed.

iii. $\models_i \models_{tot}$.

iv. Neither. The precondition does not express anything about $x$, and $\text{skip}$ certainly does not establish its value as 21. It could have any other value, failing the postcondition.

v. $\models_i \models_{tot}$.

vi. $\models$ only. If $x > 1$, then the program will terminate and the postcondition will be satisfied. But for $x \in \{0, 1\}$, $x$ will never grow larger than 0 or 1. Since $y > 1$, $x$ will never be larger than $y$, and the loop will never terminate.

vii. $\models$ only. Since the loop never terminates, it does not return any program states to check against the postcondition. Hence every program state returned satisfies the postcondition (or in other words, because no program states are returned, the postcondition is never violated).

2 A Hoare Logic for Partial Correctness

i. The proof rule [cons] allows us to strengthen the precondition and/or weaken the postcondition. It also allows us to replace them with syntactically distinct, but semantically equivalent assertions. Proving the validity of the logical implications proves that the strengthening/weakening holds in all program states, and is necessary for the soundness of the proof rule.

ii. The assignment axiom [ass] can be interpreted as follows. If an assertion $p$, with every occurrence of $x$ replaced by $e$, holds before execution, then surely, after the assignment is executed, the assertion $p$ without this replacement will still hold (since $x$ now has the value of $e$).

(The axiom, at first, may look a little strange or jarring. But it is much simpler to use than the alternative “forward” axiom – see the later exercise. Reasoning backwards is more efficient!)
iii. A possible proof tree is:

\[
\begin{align*}
\text{[ass]} \quad & x + 1 > 1 \\ 
\text{[cons]} \quad & x := x + 1 \quad \{x > 1\} \\ 
\text{[comp]} \quad & \{x > 0\} \quad x := x + 1; \quad \text{skip} \quad \{x > 1\}
\end{align*}
\]

The logical implication \(x > 0 \Rightarrow x + 1 > 1\) is clearly valid.

iv. A possible proof tree is given in Figure 1.

What remains to be shown is that

\[x = a \land y = b \Rightarrow x + y = a + b \land x = a\]

is valid. Using the antecedent we substitute \(a\) for \(x\) and \(b\) for \(y\) in the consequent, obtaining:

\[x = a \land y = b \Rightarrow a + b = a + b \land a = a.\]

Clearly this is valid.

v. The proof rule [while] allows us to reason about loop invariants, i.e. assertion \(p\). In proving that \(p\) is maintained after an execution of \(P\), we know that it will be maintained after any number of executions of \(P\). If the loop terminates, we know that the Boolean guard must no longer evaluate to true, so we get the additional conjunct \(\neg b\) in the postcondition of the conclusion.

vi. A possible proof tree is given in Figure 2.

We need to show that

\[in + m = 250 \Rightarrow (i - 1)n + m + n = 250\]

is valid. This follows from elementary mathematics:

\[250 = in + m = in + m + n - n = (i - 1)n + m + n\]

The other implications arising from [cons] are clearly valid.

vii. A possible inference rule is:

\[
\begin{align*}
\text{[repeat]} \quad & \vdash \{p\} \quad P \quad \{q\} \\ 
\text{[repeat]_2} \quad & \vdash \{q\} \quad P \quad \{q\} \\ 
\text{[repeat]_2} \quad & \vdash \{q\} \quad P \quad \text{until} \quad b \quad \{b \land q\}
\end{align*}
\]

A weaker, but also sound inference rule is:

\[
\begin{align*}
\text{[repeat]_2} \quad & \vdash \{q\} \quad P \quad \{q\} \\ 
\text{[repeat]_2} \quad & \vdash \{q\} \quad P \quad \text{until} \quad b \quad \{b \land q\}
\end{align*}
\]
viii. A sound axiom would be $\vdash \{p\} \text{surprise} \{\text{true}\}$. Because we do not know which variable will be changed by \text{surprise}, we cannot assert anything about program variables in the postcondition. We can however use \text{[cons]} to derive postconditions that are true in all program states, e.g. statements about arithmetic like:

\[
\vdash \{p\} \text{surprise} \{\forall x : \mathbb{N}. \exists y : \mathbb{N}. y > x\}.
\]

ix. The following is known to be equivalent to the well-known “backward” rule:

\[
\vdash \{p\} x := e \{\exists x^{old}. p[x^{old}/x] \land x = e[x^{old}/x]\}
\]

where $x^{old}$ is fresh (i.e. it does not occur free in $p$ or $e$) and is not the same variable as $x$. The variable $x^{old}$ can be understood as recording the value that $x$ used to have before the assignment. Because $x$ may have changed, the first conjunct replaces each occurrence of it in $p$ with the old value, $x^{old}$. The second conjunct expresses the value of $x$ after assignment, replacing occurrences of $x$ in the expression with the old value $x^{old}$.

If the soundness of this axiom is not clear at first, try applying it to an assignment like $x := x + 5$ with precondition $x > 0$.

While this axiom is sound and equivalent to the backward rule, it tends to be used less in practice, to avoid the accumulation of existential quantifiers (one per assignment!).
where Subtree $X$ is:

\[
\text{Subtree } X \quad \begin{array}{c}
\text{[ass]} \quad \vdash \{x + y = a + b \wedge x = a\} \\
\text{[cons]} \quad \vdash \{x = a \wedge y = b\}
\end{array}
\]

\[
\text{t} := \text{x}; \quad \text{x} := \text{x} + \text{y}; \quad \text{y} := \text{t} \{x = a + b \wedge y = a\}
\]

Figure 1: Proof tree for Exercise 2.1-iv

\[
\text{[ass]} \quad \vdash \{(i - 1)n + m + n = 250\} \\
\text{[cons]} \quad \vdash \{\text{INV}\} \\
\text{[comp]} \quad \vdash \{(i - 1)n + m = 250\} \quad \vdash \{i = 1\} \{\text{INV}\}
\]

\[
\text{while} (i > 0) \{\text{INV}\} \\
\text{[while]} \quad \vdash \{\text{INV}\} \text{ while } (i > 0) \text{ do } m := m + n; \ i := i - 1 \{\neg(i > 0) \wedge \text{INV}\}
\]

\[
\text{[cons]} \quad \vdash \{\text{INV}\} \text{ while } (i > 0) \text{ do } m := m + n; \ i := i - 1 \{\text{INV}\}
\]

Figure 2: Proof tree for Exercise 2.1-vi
Problem Sheet 3: Separation Logic
Sample Solutions

Chris Poskitt
ETH Zürich

Starred exercises (*) are more challenging than the others.

1 Separation Logic Assertions

i. (a) Does not hold. Expresses that the heap has exactly one location, obtained by evaluating $i$; but there are several other locations in the heap.

(b) Does not hold. Expresses that the heap can be split into two disjoint parts: one with the location obtained by evaluating $i$, another disjoint one by evaluating $j$, and the same contents at both. Not only are there additional locations in the heap, but $i$ and $j$ are evaluated by the store to the same location (i.e. they do not denote two disjoint locations in the heap).

(c) Holds. The heap can be divided into two disjoint portions: one portion with exactly the location given by evaluating $i$; the other portion being the rest of it. The former satisfies $i \rightarrow z$ and the latter satisfies true.

(d) Holds. The first conjunct is true for the reason in (c); the second is true because the store evaluates $i$ and $j$ to the same location. For the third conjunct, we can certainly split the heap into three disjoint parts: one satisfying $z \rightarrow 1$, one satisfying $z + 1 \rightarrow 2$, and finally one being the remainder of the heap which of course will satisfy true.

(e) Does not hold. It asserts that $i \rightarrow x$ and $j \rightarrow x'$ are disjoint parts of the heap, but $i$ and $j$ both evaluate to the same location in the heap.

(f) Holds. Every heap can be split into two disjoint parts: one, the original heap (which satisfies true), and the other, the empty heap (which satisfies emp).

ii. (a) $p$ implies $p \ast p$ is not valid. Counterexample: take $p$ to be $x \rightarrow a$. A state satisfying this assertion (i.e. that there is exactly one location) will not satisfy $x \rightarrow a \ast x \rightarrow a$.

(b) $p \ast q$ implies $[(p \land q) \ast true]$ is not valid. Counterexample: take $q$ to be $\neg p$. A heap can satisfy $p \ast \neg p$ because $p$ might hold in one disjoint part, and $\neg p$ in another. But there is no part of any heap that will satisfy both $p$ and $\neg p$, and so $[(p \land \neg p) \ast true]$ cannot be satisfied.

*Some exercises based on those previously set by Stephan van Staden.
2 Separation Logic Proofs

i. Proof outline using the small axioms, frame rule, consequence rule, and the following derived axiom:
\[ \vdash \{ e \rightarrow e' \} \ x := [e] \ {e \rightarrow e' \wedge x = e'} \]
(provided that \( x \) does not appear free in \( e, e' \)).

\[
\begin{align*}
\{ \text{emp} \} \\
\ l := \text{cons}(1) \\
\ {l \leftrightarrow 1} \\
\ {l \leftrightarrow 1 \ast \text{emp}} \\
\ r := \text{cons}(2, 3) \\
\ {l \leftrightarrow 1 \ast r \leftrightarrow 2, 3} \\
\ temp1 := [r + 1] \\
\ {l \leftrightarrow 1 \ast r \leftrightarrow 2, 3 \wedge temp1 = 3} \\
\ temp2 := [l] \\
\ {l \leftrightarrow 1 \ast r \leftrightarrow 2, 3 \wedge temp1 = 3 \wedge temp2 = 1} \\
\ [l] := temp1 \\
\ {l \leftrightarrow temp1 \ast r \leftrightarrow 2, 3 \wedge temp1 = 3 \wedge temp2 = 1} \\
\ [r] := temp2 \\
\ {l \leftrightarrow temp1 \ast r \leftrightarrow temp2,3 \wedge temp1 = 3 \wedge temp2 = 1} \\
\ {l \leftrightarrow 3 \ast r \leftrightarrow 1,3 \wedge temp1 = 3 \wedge temp2 = 1} \\
\ {l \leftrightarrow 3 \ast r \leftrightarrow 1,3} \\
\end{align*}
\]
A possible depiction of the post-state is given below:
ii. A possible proof outline:

\{ \text{list\(a \::\ as, i\)} \}
\{ \exists j. i \mapsto a, j * \text{list\(as, j\)} \}
\{ i \mapsto a, j * \text{list\(as, j\)} \}
\text{dispose\(i\)}
\{ i + 1 \mapsto j * \text{list\(as, j\)} \}
\quad k := |i + 1|
\{ i + 1 \mapsto j * \text{list\(as, j\)} \land k = j \}
\text{dispose\(i + 1\)}
\{ \text{list\(as, j\)} \land k = j \}
\{ \exists j. \text{list\(as, j\)} \land k = j \}
\{ \text{list\(as, k\)} \}
\quad i := k
\{ \text{list\(as, i\)} \}

iii. (*) Here is a possible procedure for CopyTree:

```pascal
procedure CopyTree(p, q)
  if isAtom(p) then
    q := p;
  else
    local p1, p2, q1, q2;
    p1 := [p];
    p2 := [p+1];
    CopyTree(p1, q1);
    CopyTree(p2, q2);
    q := cons(q1, q2);
  end
end
```

We focus on the most crucial part of the program, i.e. the code in the else block. The following proof outline uses the small axioms, frame rule, rule of consequence, as well as:

\[ \vdash \{ \text{tree\(t_1, \tau\)} \} \text{CopyTree\(t_1, t_2\)} \{ \text{tree\(t_1, \tau\)} \ast \text{tree\(t_2, \tau\)} \} \]

as an inductively assumed axiom (for recursive calls of the CopyTree procedure).
\{\text{tree}(p, \tau)\}
\{\exists x, y, \tau_1, \tau_2. \; p \mapsto x, y \ast \text{tree}(x, \tau_1) \ast \text{tree}(y, \tau_2) \land \tau = \langle \tau_1, \tau_2 \rangle\}
\ p_1 := [p]
\{\exists y, \tau_1, \tau_2. \; p \mapsto p_1, y \ast \text{tree}(p_1, \tau_1) \ast \text{tree}(y, \tau_2) \land \tau = \langle \tau_1, \tau_2 \rangle\}
\ p_2 := [p + 1]
\{\exists \tau_1, \tau_2. \; p \mapsto p_1, p_2 \ast \text{tree}(p_1, \tau_1) \ast \text{tree}(p_2, \tau_2) \land \tau = \langle \tau_1, \tau_2 \rangle\}
\text{CopyTree}(p_1, q_1)
\{\exists \tau_1, \tau_2. \; p \mapsto p_1, p_2 \ast \text{tree}(p_1, \tau_1) \ast \text{tree}(q_1, \tau_1) \ast \text{tree}(p_2, \tau_2) \land \tau = \langle \tau_1, \tau_2 \rangle\}
\text{CopyTree}(p_2, q_2)
\{\exists \tau_1, \tau_2. \; p \mapsto p_1, p_2 \ast \text{tree}(p_1, \tau_1) \ast \text{tree}(q_1, \tau_1) \ast \text{tree}(p_2, \tau_2) \ast \text{tree}(q_2, \tau_2) \land \tau = \langle \tau_1, \tau_2 \rangle\}
\ q := \text{cons}(q_1, q_2)
\{\exists \tau_1, \tau_2. \; p \mapsto p_1, p_2 \ast \text{tree}(p_1, \tau_1) \ast \text{tree}(q_1, \tau_1) \ast \text{tree}(p_2, \tau_2) \ast \text{tree}(q_2, \tau_2) \ast q \mapsto q_1, q_2 \land \tau = \langle \tau_1, \tau_2 \rangle\}
\{\exists \tau_1, \tau_2. \; p \mapsto p_1, p_2 \ast \text{tree}(p_1, \tau_1) \ast \text{tree}(p_2, \tau_2) \ast q \mapsto q_1, q_2 \ast \text{tree}(q_1, \tau_1) \ast \text{tree}(q_2, \tau_2) \land \tau = \langle \tau_1, \tau_2 \rangle\}
\{\text{tree}(p, \tau) \ast \text{tree}(q, \tau)\}
Problem Sheet 5: Program Proofs
Sample Solutions

Chris Poskitt
ETH Zürich

Starred exercises (*) are more challenging than the others.

1 Axiomatic Semantics Recap

i. I propose the axiom:

\[ \vdash \{ p \} \text{havoc}(x_0, \ldots, x_n) \{ \exists x_0^{\text{old}}, \ldots, x_n^{\text{old}}. p[x_0^{\text{old}}/x_0, \ldots, x_n^{\text{old}}/x_n] \} \]

Essentially it is the same as the forward assignment axiom (see Problem Sheet 1), but without conjuncts about the new values of each \( x_i \), since we do not know what they will be after the execution of \text{havoc}.

ii. Below is a possible program and proof outline:

\[
\begin{align*}
\{ x \geq 0 \} \\
\{ x! * 1 = x! \land x \geq 0 \} \\
y := 1; \\
\{ x! * y = x! \land x \geq 0 \} \\
z := x; \\
\{ z! * y = x! \land z \geq 0 \} \\
\text{while } z > 0 \text{ do} \\
\{ z > 0 \land z! * y = x! \land z \geq 0 \} \\
\{ (z - 1)! * (y \circ z) = x! \land (z - 1) \geq 0 \} \\
y := y \circ z; \\
\{ (z - 1)! * y = x! \land (z - 1) \geq 0 \} \\
z := z - 1; \\
\{ z! * y = x! \land z \geq 0 \} \\
\text{end} \\
\{ \neg(z > 0) \land z! * y = x! \land z \geq 0 \} \\
\{ y = x! \}
\end{align*}
\]

Observe that the loop invariant \( z! * y = x! \land z \geq 0 \) is key to completing the proof.
iii. A possible inference rule would be:

\[
\begin{align*}
&\text{from-until} & A\{\text{inv}\} &\implies C\{\text{inv}\} \\
&\text{from} & C\{\text{inv}\} &\implies \{p\} A\{\text{inv}\} \\
&\text{from} & \{p\} &\text{from } A \text{ until } b \text{ loop } C \text{ end } \{\text{inv } \land b\} \\
\end{align*}
\]

iv. A possible proof outline is the following:

\[
\begin{array}{l}
\{ n \geq 0 \} \\
\text{from} \\
\quad k := n \\
\quad found := \text{False} \\
\quad \{ 0 \leq k \leq n \land (found \implies 1 \leq k \leq n \land A[k] = v) \} \\
\text{until found or } k < 1 \text{ loop} \\
\quad \{ 1 \leq k \leq n \land \neg found \land (found \implies 1 \leq k \leq n \land A[k] = v) \} \\
\quad \text{if } A[k] = v \text{ then} \\
\quad \quad \{ A[k] = v \land 1 \leq k \leq n \land \neg found \} \\
\quad \quad \{ 0 \leq k \leq n \land 1 \leq k \leq n \land A[k] = v \} \\
\quad \quad found := \text{True} \\
\quad \quad \{ 0 \leq k \leq n \land (found \implies 1 \leq k \leq n \land A[k] = v) \} \\
\quad \text{else} \\
\quad \quad \{ A[k] /= v \land 1 \leq k \leq n \land \neg found \} \\
\quad \quad \{ 1 \leq k \leq n + 1 \land (found \implies 2 \leq k \leq n + 1 \land A[k - 1] = v) \} \\
\quad \quad k := k - 1 \\
\quad \quad \{ 0 \leq k \leq n \land (found \implies 1 \leq k \leq n \land A[k] = v) \} \\
\quad \text{end} \\
\quad \\{ 0 \leq k \leq n \land (found \implies 1 \leq k \leq n \land A[k] = v) \} \\
\text{end} \\
\{ (found \land 1 \leq k \leq n \land A[k] = v) \lor (\neg found \land k = 0) \} \\
\{ (found \implies 1 \leq k \leq n \land A[k] = v) \land (\neg found \implies k < 1) \} \\
\end{array}
\]

Again, note the importance of determining a strong enough loop invariant, i.e.

\[
0 \leq k \leq n \land (found \implies 1 \leq k \leq n \land A[k] = v)
\]

for the proof to be able to go through. Note also that we can apply backwards reasoning, as usual, when the assignment involves a Boolean value (in this case, \(found[\text{True } / \text{ found}] \equiv \text{True}\)).

v. Assume that \(\vdash \{\text{WP}[P, post]\} P \{post\}\) and \(\vdash \{p\} P \{q\}\). From the definition of \(\models\), executing \(P\) on a state satisfying \(p\) results in a state satisfying \(q\). By definition, \(\text{WP}[P, post]\) expresses the weakest requirements on the state for \(P\) to establish \(q\); hence \(p\) is either equivalent to or stronger than \(\text{WP}[P, post]\), and \(p \Rightarrow \text{WP}[P, post]\) is valid. Clearly, \(q \Rightarrow q\) is also valid, so we can apply the rule of consequence \([\text{cons}]\) and derive the result that \(\vdash \{p\} P \{q\}\).

**Note:** this property is called relative completeness, i.e. all valid triples can be proven in the Hoare logic, relative to the existence of an oracle for deciding the validity of implications (such as those in \([\text{cons}]\) ).
2 Separation Logic Recap

i. There are instances of $s, h$ and $p$ such that the state satisfies the first assertion. For example,

$$s, h \models x \mapsto x \not\mapsto x$$

if $s(x) = 5$, $h(5) = 5$, and $h$ is defined for no other values. However, $x = y \not\mapsto (x = y)$ is not satisfiable since $x, y$ denote values in the store, which is heap-independent.

ii. (a) Satisfies.

(b) Does not satisfy (the heap only contains two locations).

(c) Does not satisfy (the heap contains more than one location).

(d) Satisfies. The variables $x$ and $y$ are indeed evaluated to the same location by the store. The second conjunct expresses that there is a location in the heap determined by evaluating $y$ (clearly true).

(e) Satisfies.

iii. A proof outline is given below:

\[
\{\text{emp}\} \\
{x := \text{cons}(5, 9);} \\
\{x \mapsto 5, 9\} \\
{y := \text{cons}(6, 7);} \\
\{x \mapsto 5, 9 * y \mapsto 6, 7\} \\
\exists x_{\text{old}}. \ x \mapsto 5, 9 * y \mapsto 6, 7 \land x_{\text{old}} = x \\
\ x := [x]; \\
\exists x_{\text{old}}. \ x_{\text{old}} \mapsto 5, 9 * y \mapsto 6, 7 \land x = 5 \\
\ [y + 1] := 9; \\
\exists x_{\text{old}}. \ x_{\text{old}} \mapsto 5, 9 * y \mapsto 6, 9 \land x = 5 \\
\ \text{dispose}(y); \\
\exists x_{\text{old}}. \ x_{\text{old}} \mapsto 5, 9 * y + 1 \mapsto 9 \land x = 5 \\
\] and a depiction of the final state:

![Diagram of store and heap with values]
iv. A proof outline is given below:

\{ \text{tree } (1, t) \ i \} \\
\{ \exists l, r \cdot i \mapsto l, r \ast \text{tree } 1 l \ast \text{tree } t r \} \\
\quad \begin{array}{c}
\quad x := [i]; \\
\quad \exists r \cdot i \mapsto x, r \ast \text{tree } 1 x \ast \text{tree } t r \\
\quad \quad [i] := 2; \\
\quad \exists r \cdot i \mapsto 2, r \ast \text{tree } 1 x \ast \text{tree } t r \\
\quad \quad y := [i + 1]; \\
\quad \{ i \mapsto 2, y \ast \text{tree } 1 x \ast \text{tree } t y \} \\
\quad \quad \text{dispose } i; \\
\quad \{ (i + 1) \mapsto y \ast \text{tree } 1 x \ast \text{tree } t y \} \\
\quad \{ (i + 1) \mapsto y \ast x \mapsto 1 \ast \text{tree } t y \} \\
\quad \quad \text{dispose } x; \\
\quad \{ (i + 1) \mapsto y \ast \text{tree } t y \} \\
\quad \quad \text{dispose } (i + 1); \\
\{ \text{tree } t y \} \}

Problem Sheet 6: Data Flow Analysis
Sample Solutions

Chris Poskitt*
ETH Zürich

Starred exercises (*) are more challenging than the others.

1 Reaching Definitions Analysis

i-ii. The control flow graph and the results of the reaching definitions analysis are given in the diagram below:

iii. We give the use-definition information for \( x \) and \( y \) in the table below (you could also annotate the diagram above with additional arrows).

*These sample solutions were adapted from previous iterations of the course when Stephan van Staden was the teaching assistant.
2 Live Variables Analysis

i. Below we identify the blocks of the program:

\[
\begin{align*}
    & [x := y]^1 \\
    & [x := x-1]^2 \\
    & [x := 4]^3 \\
    & \text{while } [y < x]^4 \text{ do} \\
    & \quad [y := y+x]^5 \\
    & \text{end} \\
    & [y := 0]^6
\end{align*}
\]

ii. The system of equations for a live variable analysis are as follows:

\[
\begin{align*}
    LV_{entry}(1) &= (LV_{exit}(1) - \{x\}) \cup \{y\} \\
    LV_{entry}(2) &= (LV_{exit}(2) - \{x\}) \cup \{x\} \\
    LV_{entry}(3) &= LV_{exit}(3) - \{x\} \\
    LV_{entry}(4) &= LV_{exit}(4) \cup \{x, y\} \\
    LV_{entry}(5) &= (LV_{exit}(5) - \{y\}) \cup \{x, y\} \\
    LV_{entry}(6) &= LV_{exit}(6) - \{y\}
\end{align*}
\]

\[
\begin{align*}
    LV_{exit}(1) &= LV_{entry}(2) \\
    LV_{exit}(2) &= LV_{entry}(3) \\
    LV_{exit}(3) &= LV_{entry}(4) \\
    LV_{exit}(4) &= LV_{entry}(5) \cup LV_{entry}(6) \\
    LV_{exit}(5) &= LV_{entry}(4) \\
    LV_{exit}(6) &= \emptyset
\end{align*}
\]

iii. We begin the iteration by initialising every set to \(\emptyset\). Then, we iteratively update the sets by applying the equation system above. (For simplicity, the columns omit sets when a particular iteration does not update the previous value.)
iv. We eliminate blocks $b$ of the form $[x := \ldots]^b$ if $x$ is not an element of $LV_{exit}(b)$:

\[
\begin{align*}
[x := y]^1 \\
[x := 4]^3 \\
\text{while } [y < x]^4 \text{ do} \\
\quad [y := y+x]^5 \\
\text{end}
\end{align*}
\]

v. (⋆) The program is not yet free of dead variables: $x$ in block 1 is still dead. We strengthen the definition of $LV_{entry}$:

\[
LV_{entry}(b) = \begin{cases} 
(LV_{exit}(b) \cap kill_{LV}(b)) \cup gen_{LV}(b) & \text{if } kill_{LV}(b) \subseteq LV_{exit}(b) \\
LV_{exit}(b) & \text{otherwise}
\end{cases}
\]

The rationale is this: if a block assigns to a variable that is not live afterwards, then it must be eliminated, and should not influence the analysis by adding the variables it reads to the live variable set.

Performing a chaotic iteration with this new equation yields the following results:
1 Program Slicing

i. Here is the program dependence graph for the program fragment (blue arrows are from the use-definition analysis; red arrows indicate control dependencies):

ii. For slicing criterion print(x), i.e. block 12, we get:

```java
x := 0;
i := n;
while i > 0 do
    x := x + 1;
i := i - 1;
end
print(x);
```
For slicing criterion \texttt{print(y)}, i.e. block 13, we get:

\begin{verbatim}
y := 0;
i := n;
while i > 0 do
  i := i - 1;
j := i;
while j > 0 do
  y := y + 1;
j := j - 1;
end
end
\texttt{print(y);}\
\end{verbatim}

\section{Abstract Interpretation}

\begin{enumerate}
  \item We begin by mapping every variable to $\bot$ (except for $x, y$ in $A_1$, which are respectively mapped to $+\bot$ by assumption). Then, we iteratively update the (abstract) values of variables by applying the system of equations.

\begin{tabular}{|c|c|}
\hline
| Abstract States | Iterations | Final Values |
\hline
$A_1(x)$ & $+$ & $+$
\hline
$A_1(y)$ & $\top$ & $\top$
\hline
$A_2(x)$ & $\bot +$ & $\top$
\hline
$A_2(y)$ & $\bot +$ & $+\top$
\hline
$A_3(x)$ & $\bot +$ & $\top$
\hline
$A_3(y)$ & $\bot +$ & $+\top$
\hline
$A_4(x)$ & $\bot +$ & $\top$
\hline
$A_4(y)$ & $\bot +$ & $\top$
\hline
$A_5(x)$ & $\bot +$ & $0$
\hline
$A_5(y)$ & $\bot +$ & $0$
\hline
\end{tabular}

\item The analysis is not very precise: it cannot prove that $y$ is positive when the program fragment completes (i.e. at $A_5$).

\item (a) If we compute the factorial using a program that does not utilise the subtraction operator, then the result of the analysis becomes more precise:
\end{enumerate}
(b) Perhaps changing the program for the analysis to work more precisely is not the best approach—let’s try to improve the analysis! We’ll try a so-called relational analysis with domain $\mathcal{P}(\{-,0,+\} \times \{-,0,+\})$ to represent program states $(x,y)$. A relational analysis is more precise because the domain can express dependencies, or relationships, between $x$ and $y$.

We use the original version of the program fragment, but the new system of equations below:
A_1 = \{(+,-), (+,0), (+,+)\}
A_2 = \{(x,+) \mid (x,y) \in A_1 \} \cup \{(x,y') \mid (x',y') \in A_4 \text{ and } x \in x' \ominus + \}
A_3 = A_2 \cap \{(x,y) \mid x \in \{-,\} \text{ and } y \in \{-,0,\} \}
A_4 = \{(x',y) \mid (x',y') \in A_3 \text{ and } y \in x' \otimes y' \}
A_5 = A_2 \cap \{(0,y) \mid y \in \{-,0,\} \}

and obtain a more precise analysis allowing us to deduce that y will be positive after execution finishes:

<table>
<thead>
<tr>
<th>Iterations</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_1</td>
<td>{(+,-), (+,0), (+,+)}</td>
</tr>
<tr>
<td>A_2</td>
<td>{(+,)}</td>
</tr>
<tr>
<td>A_3</td>
<td>{(+,)}</td>
</tr>
<tr>
<td>A_4</td>
<td>{(+,)}</td>
</tr>
<tr>
<td>A_5</td>
<td>{(+,)}</td>
</tr>
</tbody>
</table>
Problem Sheet 8: Model Checking  
Sample Solutions

Chris Poskitt and Carlo A. Furia
ETH Zürich

1 Evaluating LTL Formulae on Automata

i. Yes: whenever start occurs, stop must occur eventually since it is the only means of getting to the accepting state.

ii. No: a counterexample is pull push.

iii. Yes: the formula asserts that from every position in a word (if there are any), eventually either turn off or push will occur. One of these events must occur to return to the accepting state.

iv. No: the empty word is a counterexample (◊ p demands the existence of a future position in the word for which p holds — the empty word cannot possibly satisfy it as it has no positions).

v. Yes: if the word is empty, then it will satisfy the first disjunct ("always false" holds simply because there are no positions in the empty word to check against); if the word is non-empty, the final position in the word must be turn off or push, and hence the second disjunct will be satisfied.

vi. No: a counterexample is the empty word; or turn on turn off.
2 Equivalence of LTL Formulae

i.

\[ w, i \models \text{true} \lor F \]
iff for some \( i \leq j \leq n \) we have \( w, j \models F \)
and for all \( i \leq k < j \) we have \( w, k \models \text{true} \) \hspace{1cm} [definition of until]
iff for some \( i \leq j \leq n \) we have \( w, j \models F \) \hspace{1cm} [semantics of true]

ii.

\[ w, i \models \neg \Diamond \neg F \]
iff \( w, i \not\models \Diamond \neg F \) \hspace{1cm} [definition of not]
iff it is not the case that for some \( i \leq j \leq n \) we have \( w, j \models \neg F \) \hspace{1cm} [semantics of eventually]
iff for all \( i \leq j \leq n \) it is not the case that \( w, j \models \neg F \) \hspace{1cm} [semantics of quantifiers]
iff for all \( i \leq j \leq n \) it is not the case that \( w, j \not\models F \) \hspace{1cm} [semantics of negation]
iff for all \( i \leq j \leq n \), \( w, j \models F \) \hspace{1cm} [simplify double negation]

iii.

\[ w, i \models \Diamond \Diamond p \]
iff for some \( i \leq j \leq n \) we have \( w, j \models \Diamond p \) \hspace{1cm} [semantics of eventually]
iff for some \( i \leq j \leq h \leq n \) we have \( w, h \models p \) \hspace{1cm} [sem. eventually; merging intervals]
iff for some \( i \leq h \leq n \) we have \( w, h \models p \) \hspace{1cm} [a fortiori]
iff \( w, i \models \Diamond p \) \hspace{1cm} [semantics of eventually]
3 Automata-Based Model Checking

i. The automaton we build from the temporal formula is the following.

![Automaton Diagram]

ii. The intersection automaton is the following:

![Intersection Automaton Diagram]

iii. Any accepting run is a counterexample to the LTL formula being a property of the microwave oven automaton. There are several, for example: pull push, pull push pull push, ...
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:

\[
\begin{array}{c}
\text{not } c \\
\text{c}
\end{array}
\]

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

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

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

\[
\begin{array}{c}
\text{Pred(}\neg c\text{)} \\
\text{not Pred(}\neg c\text{)}
\end{array}
\]

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

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

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

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

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

\[
\begin{align*}
+\texttt{ex}(p) &= (x > 0 \Rightarrow x > 0) \\
-\texttt{ex}(p) &= (x > 0 \Rightarrow \neg x > 0) \\
+\texttt{ex}(q) &= (x > 0 \Rightarrow y > 0) \\
-\texttt{ex}(q) &= (x > 0 \Rightarrow \neg y > 0) \\
+\texttt{ex}(r) &= (x > 0 \Rightarrow z > 0) \\
-\texttt{ex}(r) &= (x > 0 \Rightarrow \neg z > 0)
\end{align*}
\]

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

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

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

\[
\begin{align*}
\text{if } \text{True} \text{ then } \\
& \quad p := \text{True} \\
\text{else } \\
& \quad p := ?
\end{align*}
\]

This reduces even further to $p := \text{True}$, which we add to $A_1$. 

\[2\]
Next, we address the assignment \( z := (x \ast y) + 1 \). Recall that an assignment \( x := f \) is over-approximated by a parallel conditional assignment:

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

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

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

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

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

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

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

and hence add \textbf{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:

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

(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:

\[
\begin{align*}
\text{if } \mathcal{R} \text{ then} & \quad \text{assume } p \text{ end} \\
& \quad p := \text{True} \\
& \quad q := \text{True} \\
\text{else} & \quad \text{assume not } p \text{ end} \\
& \quad p := \text{False} \\
& \quad \text{if } \mathcal{R} \text{ then} \\
& \qquad \text{assume not } p \text{ end} \\
& \qquad p := \text{False} \\
& \qquad q := \text{True} \\
& \quad \text{else} \\
& \qquad \text{assume True end -- can delete this assume} \\
& \qquad q := ? \\
\text{end} \\
\text{end} \\
\text{assert } q \text{ end}
\end{align*}
\]
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 \( p \land \neg q \land \neg r \), 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

\[
s \models x > 0 \land y \leq 0 \land (x \times y) + 1 \leq 0
\]

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 \( \neg p \land \neg q \). Again, we interpret conditions (assume in this case) as asserts, and apply the corresponding Hoare logic axioms:
\begin{align*}
\{x < 0 \text{ and } x*x <= 0\} \\
\{x <= 0 \text{ and } x /= 0 \text{ and } x <= 0 \text{ and } x*x <= 0\} \\
\quad \text{assert } x <= 0 \\
\{x /= 0 \text{ and } x <= 0 \text{ and } x*x <= 0\} \\
\quad \text{assert } x /= 0 \text{ end} \\
\{x <= 0 \text{ and } x*x <= 0\} \\
\quad y := x*x \\
\{x <= 0 \text{ and } y <= 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.
Problem Sheet 10: Verification of Real-Time Systems
Sample Solutions

Chris Poskitt and Carlo A. Furia
ETH Zürich

Starred exercises (⋆) are more challenging than the others.

1 MTL Property Checking

i. Yes: it simply means that $a$ holds at every position (if any) of accepted timed words.

ii. No: this requires that relative to every position (if any) of accepted timed words, $a$ occurs 1 time unit in the future; but this cannot be the case for the last position of any (non-empty) timed word. (The only position that can be reasoned about relative to the end position is the end position, which is exactly 0 time units in the future.) A counterexample is the timed word $(a, 1.0)$ $(a, 2.0)$.

iii. Yes: the formula requires that if there is a future position 1 time unit in the future, then $a$ holds there.

iv. Yes: the clock $x$ is reset after reading $a$, then to reach an accepting state, $c$ must occur within the range $(0, 1)$ because of the clock constraint $0 < x < 1$.

v. Yes: as above, noting that $b$ must occur before $c$.

vi. Yes: as above. It expresses that after reading $a$, $c$ must occur within the range $(0, 1)$, and until then, only $a$ or $b$ may occur (only the latter does).

vii. No. A counterexample is the timed word $(a, 1.0)$ $(b, 1.2)$ $(c, 1.3)$.

2 Region Automaton Construction

i. First, draw the clock regions associated with the timed automaton. Since there is only one clock $x$, and because the maximum constant in the clock constraints is 1, our diagram is very simple:

![Diagram of clock regions]

The initial and accepting state of the region automaton will be $(S1, x = 0)$. To determine the outgoing edges from this state, we first determine the time successors of the region $x = 0$. This is the set of clock regions that can be reached from $x = 0$ by letting time pass, i.e.
\[ 0 < x < 1 \]
\[ x = 1 \]
\[ x > 1 \]

(We don’t break \( x > 1 \) into smaller clock regions because the largest constant in the clock constraints is 1.)

In the original timed automaton, we can reach state \( S_2 \) from \( S_1 \) when \( x = 1 \) and the next position of the timed word is \( a \). One might think that we should therefore add an edge from \((S_1, x = 0)\) to \((S_2, x = 1)\) on \( a \). However, the original automaton resets \( x \) to 0, so instead of adding an edge to \((S_2, x = 1)\), we add an edge to \((S_2, x = 0)\), i.e.

Through similar reasoning, for the other edge in the original timed automaton, we complete the region automaton (omitting the non-reachable states):
ii. Following the same process used for the previous question, we get the (somewhat larger!) region automaton below. You can construct it more efficiently by noting that at least one clock is reset on each transition (e.g. for the first transition, the corresponding regions will all have $x = 0$ but varying $y$).

![Region Automaton](image)

iii. (*

![Region Automaton](image)

3 Semantics of MTL Formulae

i. The empty word satisfies $\Box \Diamond > 0$ true, but the same is not true for non-empty words ($\Diamond > 0$ true does not hold relative to the final position, since there are no positions greater than 0 time units in the future).

ii. The formula $\Box \Diamond \geq 0$ true is satisfied by any word. For such a word $w$, the relation $w, i \models \Diamond \geq 0$ true must hold for all positions $i$ in $w$, i.e. there is some $j \geq i$ such that $w, j \models$ true. Clearly this is the case because we can always take $j = i$.

iii. The formulae are not in general equivalent. Let $w = (p, 4) (q, 8)$ and $a = 1, b = 2, c = 3, d = 6$. Then $w \models \Diamond [a + c, b + d] q = \Diamond [4, 8] q$, but $w \not\models \Diamond [a, b] \Diamond [c, d] q = \Diamond [1, 2] \Diamond [3, 6] q$. 

3
Problem Sheet 11: Testing
Sample Solutions

Chris Poskitt*
ETH Zürich

1 Branch and Path Coverage

i. (a) 4.
(b) 6.

ii. (a) Yes, e.g. function(4, 6) and function(6, 4).
(b) Yes, e.g. x := 1, x := 0, and x := −1.

iii. (a) 3.
(b) 10.

iv. (a) z := true → result := “b”.
(b) y := x + x [¬ y := y + 2]n for 0 ≤ n ≤ 6.

v. (a) For full path coverage we add the test function(1, 2).
(b) For full path coverage we add the tests: x := 2, x := 3, . . . x := 8.

2 Logic Coverage

i. (a) x < y and z && x + y == 10.
(b) x > 0, y < 15, and x = 0.

ii. Yes: we can use the same tests as we used for branch coverage.

iii. (a) x < y, z, and x + y == 10.
(b) x > 0, y < 15, and x = 0.

iv. Yes in both cases.

(a) For full clause coverage we can use the tests function(4, 6) and function(1, 2).
(b) For full clause coverage we can use the tests x := 1, x := 0, and x := −1.

v. Predicate coverage implies branch coverage (in fact, the definitions are equivalent). Clause coverage, however, does not imply branch coverage. Take for example the predicate:

\[ x > 0 \lor y > 0. \]

With the tests (x → 0, y → 1) and (x → 1, y → 0) we achieve clause coverage. However, these tests do not achieve predicate coverage (since the compound formula in both cases evaluates to true) and hence do not achieve branch coverage.

*Solution sheet adapted from an earlier version by Stephan van Staden.