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 the termination of P on states satisfying pre is also guaranteed.

iii. $\models, \models_{tot}$.

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

v. $\models, \models_{tot}$.

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:

$$\dfrac{[\text{cons}]\,\dfrac{[\text{ass}]\,\dfrac{}{\vdash \{x+1>1\}\ \texttt{x := x + 1}\ \{x>1\}}}{\vdash \{x>0\}\ \texttt{x := x + 1}\ \{x>1\}} \qquad [\text{skip}]\,\dfrac{}{\vdash \{x>1\}\ \texttt{skip}\ \{x>1\}}}{\vdash \{x>0\}\ \texttt{x := x + 1; skip}\ \{x>1\}}\,[\text{comp}]$$

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 the validity of:

$$x = a \wedge y = b \quad \Rightarrow \quad x + y = a + b \wedge x = a$$

Using the antecedent we substitute a for x and b for y in the consequent, obtaining:

$$x = a \wedge y = b \quad \Rightarrow \quad a + b = a + b \wedge a = a.$$

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

$$in + m = 250 \quad \Rightarrow \quad (i-1)n + m + n = 250$$

is valid. This follows from elementary mathematics:

$$\begin{aligned} 250 &= in + m \\ &= in + m + n - n \\ &= (i-1)n + m + n \end{aligned}$$

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

vii. A possible inference rule is:

$$[\text{repeat}]\ \frac{\vdash \{p\}\ P\ \{q\} \qquad \vdash \{\neg b \wedge q\}\ P\ \{q\}}{\vdash \{p\}\ \texttt{repeat}\ P\ \texttt{until}\ b\ \{b \wedge q\}}$$

A weaker, but also sound inference rule is:

$$[\text{repeat}]_2\ \frac{\vdash \{q\}\ P\ \{q\}}{\vdash \{q\}\ \texttt{repeat}\ P\ \texttt{until}\ b\ \{b \wedge q\}}$$

viii. A sound axiom would be $\vdash \{p\}$ `surprise` $\{\text{true}\}$. Because we do not know which variable will be changed by `surprise`, we cannot assert anything about program variables in the postcondition. We can however use [cons] to derive postconditions that are true in all program states, e.g. statements about arithmetic such as:

$$\vdash \{p\}\ \texttt{surprise}\ \{\forall x : \mathbb{N}.\ \exists y : \mathbb{N}.\ y > x\}.$$

ix. The following is equivalent to the well-known “backwards” assignment axiom:

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

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 such as $\texttt{x} := \texttt{x} + \texttt{5}$ with precondition $x > 0$.

While this axiom is sound and equivalent to the backwards axiom, it tends to be used less in practice, in order to avoid the accumulation of existential quantifiers (one per assignment!).

$$\text{[comp]}\ \dfrac{\textbf{Subtree } X \qquad \text{[ass]}\ \dfrac{}{\vdash \{x=a+b\wedge t=a\}\ \texttt{y := t}\ \{x=a+b\wedge y=a\}}}{\vdash \{x=a\wedge y=b\}\ \texttt{t := x; x := x + y; y := t}\ \{x=a+b\wedge y=a\}}$$

where **Subtree** X is:

$$\text{[comp]}\ \dfrac{\text{[cons]}\ \dfrac{\text{[ass]}\ \dfrac{}{\vdash \{x+y=a+b\wedge x=a\}\ \texttt{t := x}\ \{x+y=a+b\wedge t=a\}}}{\vdash \{x=a\wedge y=b\}\ \texttt{t := x}\ \{x+y=a+b\wedge t=a\}} \qquad \text{[ass]}\ \dfrac{}{\vdash \{x+y=a+b\wedge t=a\}\ \texttt{x := x + y}\ \{x=a+b\wedge t=a\}}}{\vdash \{x=a\wedge y=b\}\ \texttt{t := x; x := x + y}\ \{x=a+b\wedge t=a\}}$$

Figure 1: Proof tree for Exercise 2.1-iv

$$\text{[cons]}\ \dfrac{\text{[while]}\ \dfrac{\text{[cons]}\ \dfrac{\text{[comp]}\ \dfrac{\text{[cons]}\ \dfrac{\text{[ass]}\ \dfrac{}{\vdash \{(i-1)n+m+n=250\}\ \texttt{m := m + n}\ \{(i-1)n+m=250\}}}{\vdash \{INV\}\ \texttt{m := m + n}\ \{(i-1)n+m=250\}} \qquad \text{[ass]}\ \dfrac{}{\vdash \{(i-1)n+m=250\}\ \texttt{i := i - 1}\ \{INV\}}}{\vdash \{INV\}\ \texttt{m := m + n; i := i - 1}\ \{INV\}}}{\vdash \{i>0\wedge INV\}\ \texttt{m := m + n; i := i - 1}\ \{INV\}}}{\vdash \{INV\}\ \texttt{while (i > 0) do m := m + n; i := i - 1}\ \{\neg(i>0)\wedge INV\}}}{\vdash \{INV\}\ \texttt{while (i > 0) do m := m + n; i := i - 1}\ \{INV\}}$$

Figure 2: Proof tree for Exercise 2.1-vi

Problem Sheet 2: Auto-Active Verification
Sample Solutions

Chris Poskitt[*]
ETH Zürich

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

1 Boogie

Solutions to the Boogie exercises are available at the following URL:

`http://se.inf.ethz.ch/courses/2015b_fall/sv/exercises/problems2-sols-boogie`

2 AutoProof

Solutions to the AutoProof exercises are available at the following URL:

`http://cloudstudio.ethz.ch/e4pubs/#sv-solutions`

[*]With input from Nadia Polikarpova and Julian Tschannen.

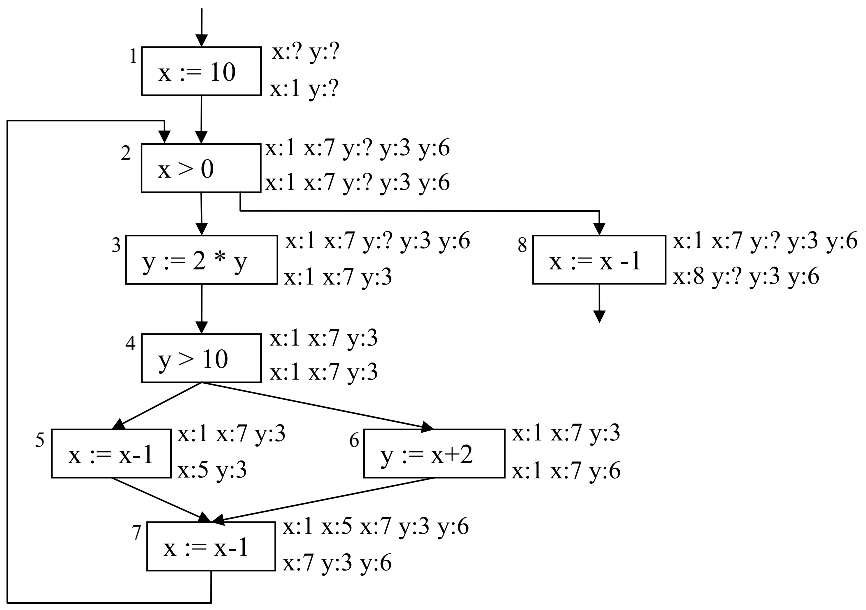
Problem Sheet 3: Data Flow Analysis
Sample Solutions

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).

[*]Solutions adapted from an earlier version of the course, when Stephan van Staden was the teaching assistant.

Program Block	x	y
1	$\emptyset$	$\emptyset$
2	$\{1,7\}$	$\emptyset$
3	$\emptyset$	$\{?,3,6\}$
4	$\emptyset$	$\{3\}$
5	$\{1,7\}$	$\emptyset$
6	$\{1,7\}$	$\emptyset$
7	$\{1,5,7\}$	$\emptyset$
8	$\{1,7\}$	$\emptyset$

2 Live Variables Analysis

i. Below we identify the blocks of the program:

```
[x := y]^1
[x := x-1]^2
[x := 4]^3
while [y < x]^4 do
    [y := y+x]^5
end
[y := 0]^6
```

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

$$\begin{aligned}
\mathrm{LV}_{\mathrm{entry}}(1) &= (\mathrm{LV}_{\mathrm{exit}}(1) - \{\mathtt{x}\}) \cup \{\mathtt{y}\} \\
\mathrm{LV}_{\mathrm{entry}}(2) &= (\mathrm{LV}_{\mathrm{exit}}(2) - \{\mathtt{x}\}) \cup \{\mathtt{x}\} \\
\mathrm{LV}_{\mathrm{entry}}(3) &= \mathrm{LV}_{\mathrm{exit}}(3) - \{\mathtt{x}\} \\
\mathrm{LV}_{\mathrm{entry}}(4) &= \mathrm{LV}_{\mathrm{exit}}(4) \cup \{\mathtt{x},\mathtt{y}\} \\
\mathrm{LV}_{\mathrm{entry}}(5) &= (\mathrm{LV}_{\mathrm{exit}}(5) - \{\mathtt{y}\}) \cup \{\mathtt{x},\mathtt{y}\} \\
\mathrm{LV}_{\mathrm{entry}}(6) &= \mathrm{LV}_{\mathrm{exit}}(6) - \{\mathtt{y}\}
\end{aligned}$$

$$\begin{aligned}
\mathrm{LV}_{\mathrm{exit}}(1) &= \mathrm{LV}_{\mathrm{entry}}(2) \\
\mathrm{LV}_{\mathrm{exit}}(2) &= \mathrm{LV}_{\mathrm{entry}}(3) \\
\mathrm{LV}_{\mathrm{exit}}(3) &= \mathrm{LV}_{\mathrm{entry}}(4) \\
\mathrm{LV}_{\mathrm{exit}}(4) &= \mathrm{LV}_{\mathrm{entry}}(5) \cup \mathrm{LV}_{\mathrm{entry}}(6) \\
\mathrm{LV}_{\mathrm{exit}}(5) &= \mathrm{LV}_{\mathrm{entry}}(4) \\
\mathrm{LV}_{\mathrm{exit}}(6) &= \emptyset
\end{aligned}$$

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.)

LV Sets	Iterations $\longrightarrow$				Final Values
$\mathrm{LV}_{\mathrm{entry}}(1)$	$\emptyset$	{y}			{y}
$\mathrm{LV}_{\mathrm{entry}}(2)$	$\emptyset$	{x}		{x, y}	{x, y}
$\mathrm{LV}_{\mathrm{entry}}(3)$	$\emptyset$		{y}		{y}
$\mathrm{LV}_{\mathrm{entry}}(4)$	$\emptyset$	{x, y}			{x, y}
$\mathrm{LV}_{\mathrm{entry}}(5)$	$\emptyset$	{x, y}			{x, y}
$\mathrm{LV}_{\mathrm{entry}}(6)$	$\emptyset$				$\emptyset$
$\mathrm{LV}_{\mathrm{exit}}(1)$	$\emptyset$	{x}		{x, y}	{x, y}
$\mathrm{LV}_{\mathrm{exit}}(2)$	$\emptyset$		{y}		{y}
$\mathrm{LV}_{\mathrm{exit}}(3)$	$\emptyset$	{x, y}			{x, y}
$\mathrm{LV}_{\mathrm{exit}}(4)$	$\emptyset$	{x, y}			{x, y}
$\mathrm{LV}_{\mathrm{exit}}(5)$	$\emptyset$	{x, y}			{x, y}
$\mathrm{LV}_{\mathrm{exit}}(6)$	$\emptyset$				$\emptyset$

iv. We eliminate blocks b of the form `[x := ...]`^{b} if `x` is not an element of $\mathrm{LV}_{\mathrm{exit}}(b)$:

```
[x := y]$^1$
[x := 4]$^3$
while [y < x]$^4$ do
    [y := y+x]$^5$
end
```

v. (∗) The program is not yet free of dead variables: `x` in block 1 is still dead. We strengthen the definition of $\mathrm{LV}_{\mathrm{entry}}$:

$$\mathrm{LV}_{\mathrm{entry}}(b) = \begin{cases} (\mathrm{LV}_{\mathrm{exit}}(b) - \mathrm{kill}_{\mathrm{LV}}(b)) \cup \mathrm{gen}_{\mathrm{LV}}(b) & \text{if } \mathrm{kill}_{\mathrm{LV}}(b) \subseteq \mathrm{LV}_{\mathrm{exit}}(b) \\ \mathrm{LV}_{\mathrm{exit}}(b) & \text{otherwise} \end{cases}$$

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:

LV Sets	Final Values
$\mathrm{LV}_{\mathrm{entry}}(1)$	{y}
$\mathrm{LV}_{\mathrm{entry}}(2)$	{y}
$\mathrm{LV}_{\mathrm{entry}}(3)$	{y}
$\mathrm{LV}_{\mathrm{entry}}(4)$	{x, y}
$\mathrm{LV}_{\mathrm{entry}}(5)$	{x, y}
$\mathrm{LV}_{\mathrm{entry}}(6)$	$\emptyset$
$\mathrm{LV}_{\mathrm{exit}}(1)$	{y}
$\mathrm{LV}_{\mathrm{exit}}(2)$	{y}
$\mathrm{LV}_{\mathrm{exit}}(3)$	{x, y}
$\mathrm{LV}_{\mathrm{exit}}(4)$	{x, y}
$\mathrm{LV}_{\mathrm{exit}}(5)$	{x, y}
$\mathrm{LV}_{\mathrm{exit}}(6)$	$\emptyset$

with which we can eliminate all of the dead code in the program:

```
[x := 4]$^3$
while [y < x]$^4$ do
    [y := y+x]$^5$
end
```

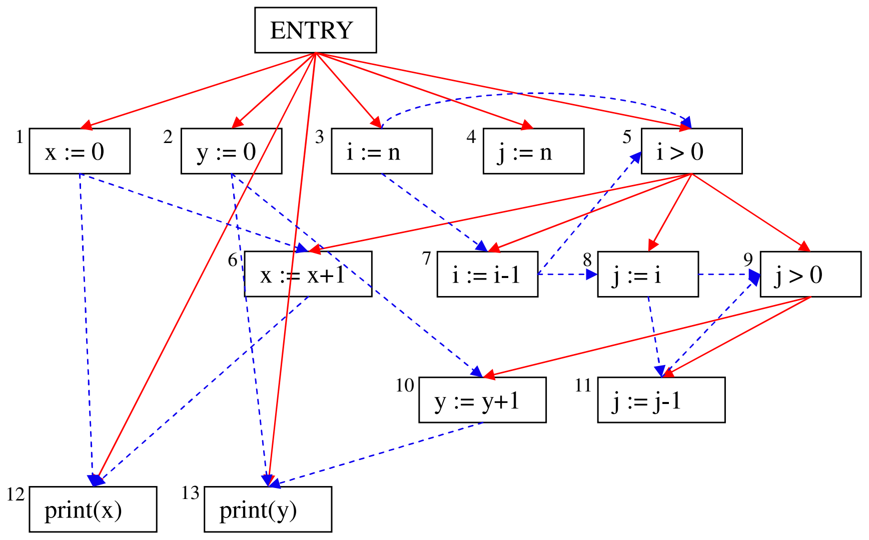
Problem Sheet 4: Program Slicing and Abstract Interpretation Sample Solutions

Chris Poskitt[*]
ETH Zürich

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

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:

```
x := 0;
i := n;
while i > 0 do
     x := x + 1;
     i := i - 1;
end
print(x);
```

[*]Solutions adapted from an earlier version of the course, when Stephan van Staden was the teaching assistant.

For slicing criterion `print(y)`, i.e. block 13, we get:

```
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
print(y);
```

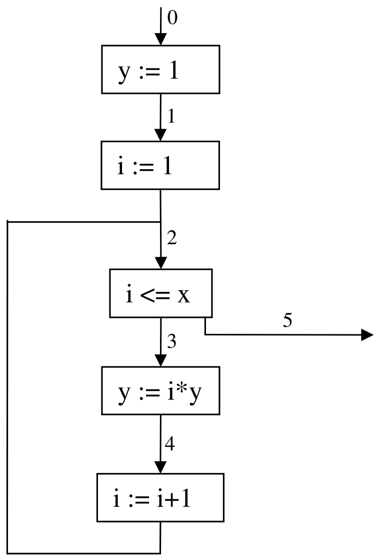
2 Abstract Interpretation

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

Abstract States	Iterations $\longrightarrow$													Final Values
$A_1(\texttt{x})$	+													+
$A_1(\texttt{y})$	$\top$													$\top$
$A_2(\texttt{x})$	$\bot$	+				$\top$				$\top$				$\top$
$A_2(\texttt{y})$	$\bot$	+				+				$\top$				$\top$
$A_3(\texttt{x})$	$\bot$		+				$\top$				$\top$			$\top$
$A_3(\texttt{y})$	$\bot$		+				+				$\top$			$\top$
$A_4(\texttt{x})$	$\bot$			+				$\top$				$\top$		$\top$
$A_4(\texttt{y})$	$\bot$			+				$\top$				$\top$		$\top$
$A_5(\texttt{x})$	$\bot$				$\bot$				0				0	0
$A_5(\texttt{y})$	$\bot$				+				+				$\top$	$\top$

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

iii. (a) If we compute the factorial using a program that does not utilise the subtraction operator, then the result of the analysis is more precise:



$$\begin{aligned}
A_0 &= [x \mapsto +,\ y \mapsto \top,\ i \mapsto \top]\\
A_1 &= A_0[y \mapsto +]\\
A_2 &= A_1[i \mapsto +] \sqcup A_4[i \mapsto A_4(i) \oplus +]\\
A_3 &= A_2\\
A_4 &= A_3[y \mapsto A_3(i) \otimes A_3(y)]\\
A_5 &= A_2
\end{aligned}$$

Abstract States	Final Values
$A_0(\mathtt{x})$	+
$A_0(\mathtt{y})$	$\top$
$A_0(\mathtt{i})$	$\top$
$A_1(\mathtt{x})$	+
$A_1(\mathtt{y})$	+
$A_1(\mathtt{i})$	$\top$
$A_2(\mathtt{x})$	+
$A_2(\mathtt{y})$	+
$A_2(\mathtt{i})$	+
$A_3(\mathtt{x})$	+
$A_3(\mathtt{y})$	+
$A_3(\mathtt{i})$	+
$A_4(\mathtt{x})$	+
$A_4(\mathtt{y})$	+
$A_4(\mathtt{i})$	+
$A_5(\mathtt{x})$	+
$A_5(\mathtt{y})$	+
$A_5(\mathtt{i})$	+

(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 $\mathfrak{P}(\{\texttt{-},\texttt{0},\texttt{+}\} \times \{\texttt{-},\texttt{0},\texttt{+}\})$ to represent program states $(\mathtt{x},\mathtt{y})$. 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:

$$
\begin{aligned}
A_1 &= \{(+,-), (+,0), (+,+)\} \\
A_2 &= \{(x,+) \mid (x,y) \in A_1\} \cup \{(x,y') \mid (x',y') \in A_4 \text{ and } x \in x' \ominus +\} \\
A_3 &= A_2 \cap \{(x,y) \mid x \in \{-,+\} \text{ and } y \in \{-,0,+\}\} \\
A_4 &= \{(x',y) \mid (x',y') \in A_3 \text{ and } y \in x' \otimes y'\} \\
A_5 &= A_2 \cap \{(0,y) \mid y \in \{-,0,+\}\}
\end{aligned}
$$

and obtain a more precise analysis allowing us to deduce that y will be positive at the end of the execution:

	Iterations							Answer
A_1	{(+,-), (+,0), (+,+)}						...	{(+,-),(+,0), (+,+)}
A_2	Ø	{(+,+)}			{(+,+),(0,+), (-,+)}		...	{(+,+),(-,+), (0,+),(-,-)}
A_3	Ø		{(+,+)}			{(+,+), (-,+)}	...	{(+,+),(-,+), (-,-)}
A_4	Ø			{(+,+)}			...	{(+,+),(-,-), (-,+)}
A_5	Ø						...	{(0,+)}

Problem Sheet 5: 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 ($\Diamond\, p$ 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.

$w, i \models \text{true} \; \mathsf{U} \; F$

iff for some $i \leq j \leq n$ we have $w, j \models F$
and for all $i \leq k < j$ we have $w, k \models \text{true}$ **[definition of until]**

iff for some $i \leq j \leq n$ we have $w, j \models F$ **[semantics of true]**

ii.

$w, i \models \neg\Diamond\neg F$

iff $w, i \not\models \Diamond\neg F$ **[definition of not]**

iff it is *not* the case that for some $i \leq j \leq n$ we have $w, j \models \neg F$ **[semantics of eventually]**

iff for all $i \leq j \leq n$ it is not the case that $w, j \models \neg F$ **[semantics of quantifiers]**

iff for all $i \leq j \leq n$ it is not the case that $w, j \not\models F$ **[semantics of negation]**

iff for all $i \leq j \leq n$, $w, j \models F$ **[simplify double negation]**

iii.

$w, i \models \Diamond\Diamond p$

iff for some $i \leq j \leq n$ we have $w, j \models \Diamond p$ **[semantics of eventually]**

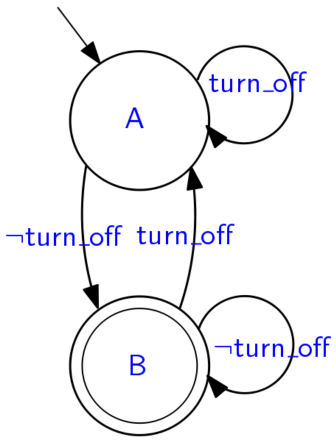
iff for some $i \leq j \leq h \leq n$ we have $w, h \models p$ **[sem. eventually; merging intervals]**

iff for some $i \leq h \leq n$ we have $w, h \models p$ **[a fortiori]**

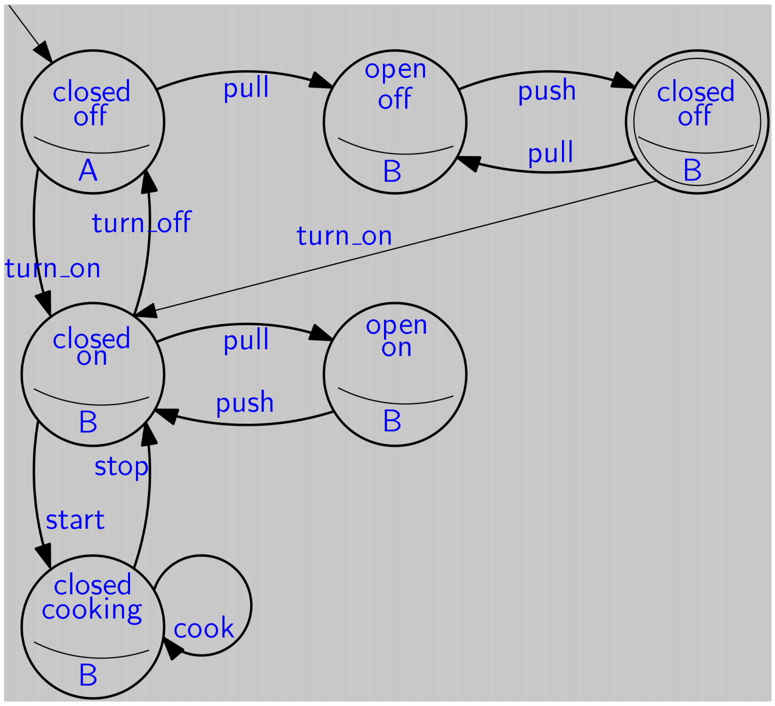
iff $w, i \models \Diamond \, p$ **[semantics of eventually]**

3 Automata-Based Model Checking

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



ii. The intersection automaton is the following:



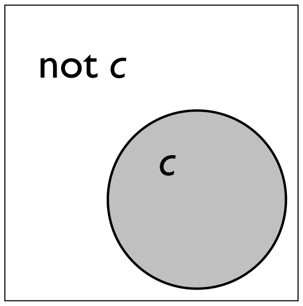
iii. Any accepting run is a counterexample to the LTL formula being a property of the microwave oven automaton. There are several, for example: `pull push`, `pull push pull push`, . . .

Problem Sheet 6: Software Model Checking
Sample Solutions

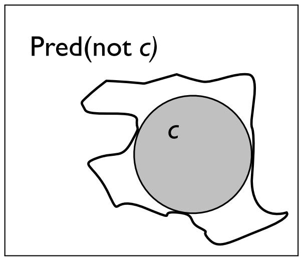
Chris Poskitt[*]
ETH Zürich

1 Predicate Abstraction

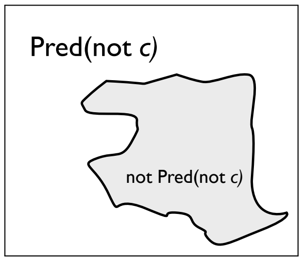
i. Let us first visualise c and `not` c in a Venn diagram:



Pred(`not` c) gives the weakest under-approximation of `not` c. Note that *Pred*(`not` c) implies `not` c, but `not` c does not (in general) imply *Pred*(`not` c). A possible visualisation in a Venn diagram might then be:



By negating *Pred*(`not` c), we get the strongest over-approximation, visualised as follows:



[*]Some exercises were adapted from earlier 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 x > 0 end` with `assume` $\neg Pred(\neg \texttt{x} > \texttt{0})$ `end`, followed by a parallel conditional assignment updating the predicates with respect to the original `assume` statement.

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

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(i) := ?
end
```

Using the axiom $\vdash \{c \Rightarrow post\}$ `assume` c `end` $\{post\}$ for the weakest precondition of assume statements, which instantiates to $\vdash \{x > 0 \Rightarrow post\}$ `assume` $x > 0$ `end` $\{post\}$, 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 consider only the branches that correspond to a $+/-\ ex(i)$ that is valid. It so happens that only $+ex(p)$ is valid, so we compute:

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

resulting in the parallel conditional assignment:

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

This simplifies 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(i) := ?
end
```

Using the axiom $\vdash \{post[f/x]\}\ x := f\ \{post\}$, which instantiates to $\vdash \{post[(\mathtt{x} * \mathtt{y}) + \mathtt{1}/\mathtt{z}]\}\ \mathtt{z} := (\mathtt{x} * \mathtt{y}) + \mathtt{1}\ \{post\}$, 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. Recall that the weakest preconditions of assert statements can be computed using $\vdash \{c \wedge \mathit{post}\}$ `assert` c `end` $\{\mathit{post}\}$.

```
{x > 0 and y <= 0 and (x*y)+1 <= 0}
{x > 0 and (x > 0 and y <= 0 and (x*y)+1 <= 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 (assumes 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.

Problem Sheet 7: Verification of Real-Time Systems Sample Solutions

Carlo A. Furia and Chris Poskitt
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 $0 < x < 1$.

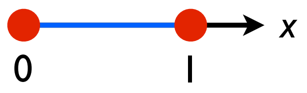
v. Yes: as above, noting that b must occur before c.

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:

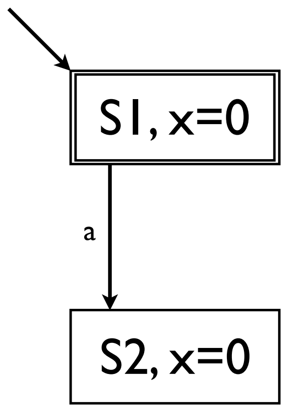


The initial and accepting state of the region automaton will be $(S1, x = 0)$. To determine the outgoing edges from this state, we first determine the *time successors* of the region $x = 0$. This is the set of clock regions that can be reached from $x = 0$ by letting time pass, i.e.

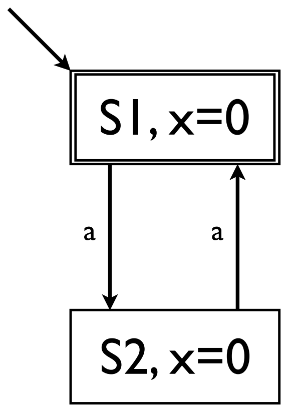
$$0 < x < 1$$
$$x = 1$$
$$x > 1$$

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

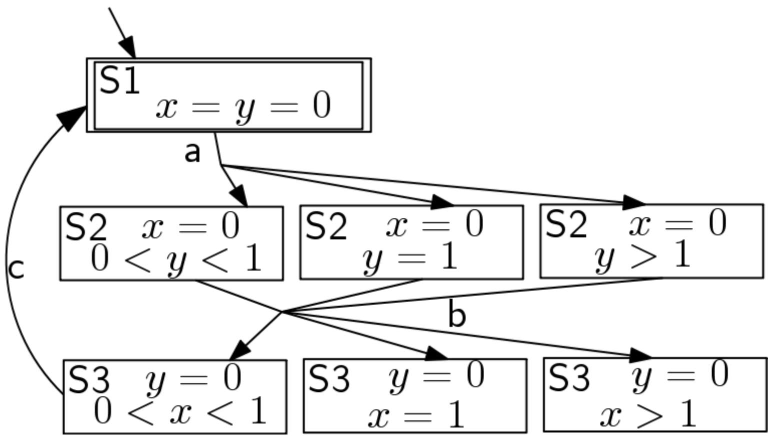
In the original timed automaton, we can reach state $S2$ from $S1$ when $x = 1$ and the next position of the timed word is a. One might think that we should therefore add an edge from $(S1, x = 0)$ to $(S2, x = 1)$ on a. However, the original automaton resets x to 0, so instead of adding an edge to $(S2, x = 1)$, we add an edge to $(S2, x = 0)$, i.e.



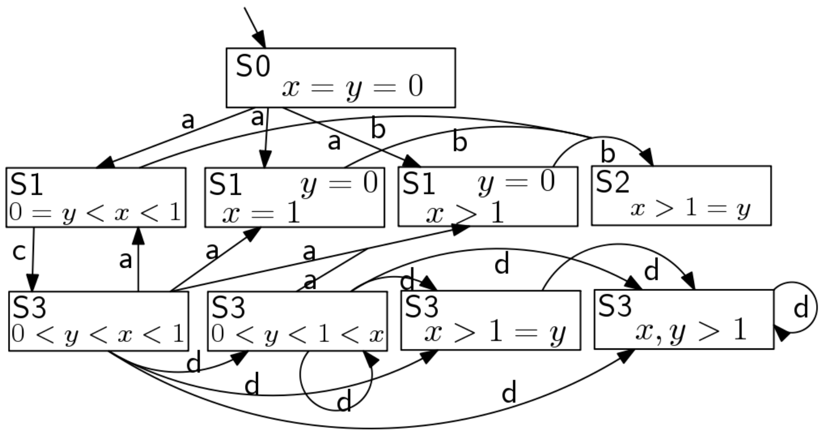
Through similar reasoning, for the other edge in the original timed automaton, we complete the region automaton (omitting the non-reachable states):



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).



iii. (∗)



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 $w \not\models \Diamond[a, b]\ \Diamond[c, d]\ q = \Diamond[1, 2]\ \Diamond[3, 6]\ q$.

Problem Sheet 8: 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, and another with the location obtained by evaluating j, with the same contents at both. Not only are there additional locations in the heap, but i and j are evaluated w.r.t. 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 $i \mapsto z$ 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 $z \mapsto 1$, one satisfying $z+1 \mapsto 2$, and then the remainder of the heap which of course will satisfy true.

   (e) Does not hold. It asserts that $i \mapsto x$ and $j \mapsto x'$ 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 $p * p$ is not valid. Counterexample: take p to be $x \mapsto a$. A state satisfying this assertion (i.e. that there is exactly one location) will not satisfy $x \mapsto a * x \mapsto a$.

   (b) $p * q$ implies $[(p \wedge q) * \text{true}]$ is not valid. Counterexample: take q to be $\neg p$. A heap can satisfy $p * \neg p$ 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 $[(p \wedge \neg p) * \text{true}]$ cannot be satisfied.

2 Separation Logic Proofs

i. Proof outline using the small axioms, frame rule, consequence rule, and the following derived axiom:

$\vdash \{e \mapsto e'\}\ x := [e]\ \{e \mapsto e' \wedge x = e'\}$

(provided that x does not appear free in e, e').

$\{\text{emp}\}$

$\quad l := \text{cons}(1)$

$\{l \mapsto 1\}$

$\{l \mapsto 1 * \text{emp}\}$

$\quad r := \text{cons}(2, 3)$

$\{l \mapsto 1 * r \mapsto 2, 3\}$

$\quad \mathit{temp}1 := [r + 1]$

$\{l \mapsto 1 * r \mapsto 2, 3 \wedge \mathit{temp}1 = 3\}$

$\quad \mathit{temp}2 := [l]$

$\{l \mapsto 1 * r \mapsto 2, 3 \wedge \mathit{temp}1 = 3 \wedge \mathit{temp}2 = 1\}$

$\quad [l] := \mathit{temp}1$

$\{l \mapsto \mathit{temp}1 * r \mapsto 2, 3 \wedge \mathit{temp}1 = 3 \wedge \mathit{temp}2 = 1\}$

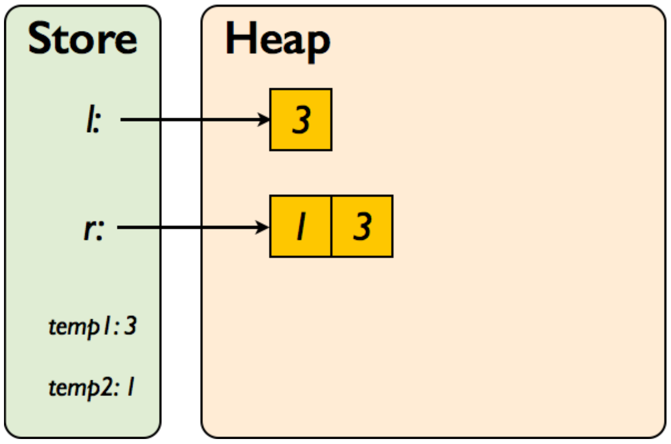
$\quad [r] := \mathit{temp}2$

$\{l \mapsto \mathit{temp}1 * r \mapsto \mathit{temp}2, 3 \wedge \mathit{temp}1 = 3 \wedge \mathit{temp}2 = 1\}$

$\{l \mapsto 3 * r \mapsto 1, 3 \wedge \mathit{temp}1 = 3 \wedge \mathit{temp}2 = 1\}$

$\{l \mapsto 3 * r \mapsto 1, 3\}$

A possible depiction of the post-state is given below:



ii. A possible proof outline:

```
$\{\text{list}(a :: as, i)\}$
$\{\exists j.\ i \mapsto a, j * \text{list}(as, j)\}$
    $\{i \mapsto a, j * \text{list}(as, j)\}$
    dispose($i$)
    $\{i + 1 \mapsto j * \text{list}(as, j)\}$
    $k := [i + 1]$
    $\{i + 1 \mapsto j * \text{list}(as, j) \wedge k = j\}$
    dispose($i + 1$)
    $\{\text{list}(as, j) \wedge k = j\}$
$\{\exists j.\ \text{list}(as, j) \wedge k = j\}$
$\{\text{list}(as, k)\}$
    $i := k$
$\{\text{list}(as, i)\}$
```

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

```
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:

$$\vdash \{\text{tree}(t_1, \tau)\}\ \text{CopyTree}(t_1, t_2)\ \{\text{tree}(t_1, \tau) * \text{tree}(t_2, \tau)\}$$

as an inductively assumed axiom (for recursive calls of the CopyTree procedure).

```
$\{\mathrm{tree}(p, \tau)\}$
$\{\exists x, y, \tau_1, \tau_2.\ p \mapsto x, y * \mathrm{tree}(x, \tau_1) * \mathrm{tree}(y, \tau_2) \land \tau = \langle \tau_1, \tau_2 \rangle\}$
    $p1 := [p]$
$\{\exists y, \tau_1, \tau_2.\ p \mapsto p1, y * \mathrm{tree}(p1, \tau_1) * \mathrm{tree}(y, \tau_2) \land \tau = \langle \tau_1, \tau_2 \rangle\}$
    $p2 := [p + 1]$
$\{\exists \tau_1, \tau_2.\ p \mapsto p1, p2 * \mathrm{tree}(p1, \tau_1) * \mathrm{tree}(p2, \tau_2) \land \tau = \langle \tau_1, \tau_2 \rangle\}$
    $\mathrm{CopyTree}(p1, q1)$
$\{\exists \tau_1, \tau_2.\ p \mapsto p1, p2 * \mathrm{tree}(p1, \tau_1) * \mathrm{tree}(q1, \tau_1) * \mathrm{tree}(p2, \tau_2) \land \tau = \langle \tau_1, \tau_2 \rangle\}$
    $\mathrm{CopyTree}(p2, q2)$
$\{\exists \tau_1, \tau_2.\ p \mapsto p1, p2 * \mathrm{tree}(p1, \tau_1) * \mathrm{tree}(q1, \tau_1) * \mathrm{tree}(p2, \tau_2) * \mathrm{tree}(q2, \tau_2) \land \tau = \langle \tau_1, \tau_2 \rangle\}$
    $q := \mathrm{cons}(q1, q2)$
$\{\exists \tau_1, \tau_2.\ p \mapsto p1, p2 * \mathrm{tree}(p1, \tau_1) * \mathrm{tree}(q1, \tau_1) * \mathrm{tree}(p2, \tau_2) * \mathrm{tree}(q2, \tau_2) * q \mapsto q1, q2 \land \tau = \langle \tau_1, \tau_2 \rangle\}$
$\{\exists \tau_1, \tau_2.\ p \mapsto p1, p2 * \mathrm{tree}(p1, \tau_1) * \mathrm{tree}(p2, \tau_2) * q \mapsto q1, q2 * \mathrm{tree}(q1, \tau_1) * \mathrm{tree}(q2, \tau_2) \land \tau = \langle \tau_1, \tau_2 \rangle\}$
$\{\mathrm{tree}(p, \tau) * \mathrm{tree}(q, \tau)\}$
```

Problem Sheet 9: Program Proofs
Sample Solutions

Chris Poskitt
ETH Zürich

1 Axiomatic Semantics

i. I propose the axiom:

$$\vdash \{p\}\ \texttt{havoc}(\texttt{x}_0, \ldots, \texttt{x}_n)\ \{\exists x_0^{\text{old}}, \ldots, x_n^{\text{old}}.\ p[x_0^{\text{old}}/x_0, \ldots, x_n^{\text{old}}/x_n]\}$$

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 `havoc`.

ii. Below is a possible program and proof outline:

```
{$x \geq 0$}
{$x! * 1 = x! \land x \geq 0$}
    $y := 1$;
{$x! * y = x! \land x \geq 0$}
    $z := x$;
{$z! * y = x! \land z \geq 0$}
    while $z > 0$ do
    {$z > 0 \land z! * y = x! \land z \geq 0$}
    {$(z-1)! * (y * z) = x! \land (z-1) \geq 0$}
        $y := y * z$;
    {$(z-1)! * y = x! \land (z-1) \geq 0$}
        $z := z - 1$;
    {$z! * y = x! \land z \geq 0$}
    end
{$\neg(z > 0) \land z! * y = x! \land z \geq 0$}
{$y = x!$}
```

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:

$$[\text{from-until}]\ \frac{\vdash \{p\}\ A\ \{inv\} \qquad \vdash \{inv \land \neg b\}\ C\ \{inv\}}{\vdash \{p\}\ \texttt{from}\ A\ \texttt{until}\ b\ \texttt{loop}\ C\ \texttt{end}\ \{inv \land b\}}$$

iv. A possible proof outline is the following:

```
{ $n \geq 0$ }
  from
    $k := n$
    $found := False$
  { $0 \leq k \leq n \ \land\ (found \Longrightarrow 1 \leq k \leq n \land A[k] = v)$ }
  until $found$ or $k < 1$ loop
    { $1 \leq k \leq n \ \land\ \neg found \ \land\ (found \Longrightarrow 1 \leq k \leq n \land A[k] = v)$ }
    if $A[k] = v$ then
      { $A[k] = v \ \land\ 1 \leq k \leq n \ \land\ \neg found$ }
      { $0 \leq k \leq n \ \land\ 1 \leq k \leq n \ \land\ A[k] = v$ }
      $found := True$
      { $0 \leq k \leq n \ \land\ (found \Longrightarrow 1 \leq k \leq n \land A[k] = v)$ }
    else
      { $A[k] \neq v \ \land\ 1 \leq k \leq n \ \land\ \neg found$ }
      { $1 \leq k \leq n+1 \ \land\ (found \Longrightarrow 2 \leq k \leq n+1 \land A[k-1] = v)$ }
      $k := k - 1$
      { $0 \leq k \leq n \ \land\ (found \Longrightarrow 1 \leq k \leq n \land A[k] = v)$ }
    end
    { $0 \leq k \leq n \ \land\ (found \Longrightarrow 1 \leq k \leq n \land A[k] = v)$ }
  end
  { $(found \land 1 \leq k \leq n \land A[k] = v) \ \lor\ (\neg found \land k = 0)$ }
{$(found \Longrightarrow 1 \leq k \leq n \land A[k] = v) \land (\neg found \Longrightarrow k < 1)$}
```

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

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

for the proof to be able to go through. Note that we can still apply backwards reasoning when the assignment involves a Boolean value (in this case, $found[True/found] = True$).

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

2 Separation Logic

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

$$s, h \models x \mapsto x * \neg x \mapsto x$$

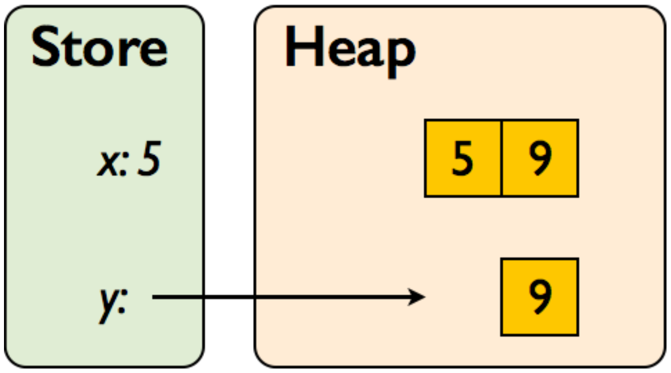
if $s(x) = 5$, $h(5) = 5$, and h is defined for no other values. However, $x = y * \neg(x = y)$ is not satisfiable since x, y denote values in the store, which is heap-independent.

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:

```
{emp}
    $x := \text{cons}(5, 9);$
$\{x \mapsto 5, 9\}$
    $y := \text{cons}(6, 7);$
$\{x \mapsto 5, 9 * y \mapsto 6, 7\}$
$\{\exists x^{\text{old}}.\ x \mapsto 5, 9 * y \mapsto 6, 7 \wedge x^{\text{old}} = x\}$
    $x := [x];$
$\{\exists x^{\text{old}}.\ x^{\text{old}} \mapsto 5, 9 * y \mapsto 6, 7 \wedge x = 5\}$
    $[y + 1] := 9;$
$\{\exists x^{\text{old}}.\ x^{\text{old}} \mapsto 5, 9 * y \mapsto 6, 9 \wedge x = 5\}$
    dispose($y$);
$\{\exists x^{\text{old}}.\ x^{\text{old}} \mapsto 5, 9 * y + 1 \mapsto 9 \wedge x = 5\}$
```

and a depiction of the final state:



iv. A proof outline is given below:

$\{tree((1,t),\, i)\}$

$\{\exists l, r.\ i \mapsto l, r * tree(1,\, l) * tree(t,\, r)\}$

$\quad x := [i];$

$\{\exists r.\ i \mapsto x, r * tree(1,\, x) * tree(t,\, r)\}$

$\quad [i] := 2;$

$\{\exists r.\ i \mapsto 2, r * tree(1,\, x) * tree(t,\, r)\}$

$\quad y := [i+1];$

$\{i \mapsto 2, y * tree(1,\, x) * tree(t,\, y)\}$

$\quad$ **dispose** i;

$\{(i+1) \mapsto y * tree(1,\, x) * tree(t,\, y)\}$

$\{(i+1) \mapsto y * x \mapsto 1 * tree(t,\, y)\}$

$\quad$ **dispose** x;

$\{(i+1) \mapsto y * tree(t,\, y)\}$

$\quad$ **dispose** $(i+1)$;

$\{tree(t,\, y)\}$

Problem Sheet 10: 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. $\texttt{x} := 1$, $\texttt{x} := 0$, and $\texttt{x} := -1$.

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

iv. (a) $\texttt{z} := \texttt{true} \rightsquigarrow \texttt{result} := \text{"b"}$.
   (b) $\texttt{y} := \texttt{x} + \texttt{x}\ [\rightsquigarrow\ \texttt{y} := \texttt{y} + 2]^n$ for $0 \leq n \leq 6$.

v. (a) For full path coverage we add the test `function`$(1, 2)$.
   (b) For full path coverage we add the tests: $\texttt{x} := 2$, $\texttt{x} := 3$, $\ldots \texttt{x} := 8$.

2 Logic Coverage

i. (a) $\texttt{x} < \texttt{y}$ and $\texttt{z}\ \&\&\ \texttt{x} + \texttt{y} == 10$.
   (b) $\texttt{x} > 0$, $\texttt{y} < 15$, and $\texttt{x} = 0$.

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

iii. (a) $\texttt{x} < \texttt{y}$, $\texttt{z}$, and $\texttt{x} + \texttt{y} == 10$.
   (b) $\texttt{x} > 0$, $\texttt{y} < 15$, and $\texttt{x} = 0$.

iv. Yes in both cases.

   (a) For full clause coverage we can use the tests `function`$(6, 4)$ and `function`$(1, 2)$.
   (b) For full clause coverage we can use the tests $\texttt{x} := 1$, $\texttt{x} := 0$, and $\texttt{x} := -1$.

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:

$$x > 0 \vee y > 0.$$

With the tests $(x \mapsto 0, y \mapsto 1)$ and $(x \mapsto 1, y \mapsto 0)$ 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.