Problem Sheet 1: Axiomatic Semantics

Chris Poskitt
ETH Zürich

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

1 Partial and Total Correctness

Recall the Hoare triple from lectures,

\{ pre \} P \{ post \}

where P denotes some program instructions, and pre, post are respectively pre- and postconditions (assertions about the program state). The triple denotes a specification of the program, with respect to the given pre- and postconditions. Such a specification, in this course, can be interpreted in one of two different ways. If the specification holds in the sense of partial correctness, we will write:

\models \{ pre \} P \{ post \}

and if it holds in the sense of total correctness, we will write:

\models_{tot} \{ pre \} P \{ post \}

1.1 Exercises

Define, in English, the meaning of:

i. \models \{ pre \} P \{ post \}

ii. \models_{tot} \{ pre \} P \{ post \}

Which senses of correctness (\models, \models_{tot}, or neither) do the following triples hold under? Explain your judgements, giving counterexamples when they do not hold. (Formal proofs are not yet required. Assume a standard semantics for the program constructs. Assume that integer overflows cannot occur.)

iii. \{ x = 21 \land y = 5 \} \text{skip} \{ y = 5 \}

iv. \{ y = 5 \} \text{skip} \{ x = 21 \land y = 5 \}

v. \{ x > 10 \} x := 2 * x \{ x > 21 \}

vi. \{ x \geq 0 \land y > 1 \} \text{while} x < y \text{do} x := x * x \{ x \geq y \}

vii. \{ x = 5 \} \text{while} x > 0 \text{do} x := x + 1 \{ x < 0 \}
2 A Hoare Logic for Partial Correctness

In the previous section we reasoned about the correctness of triples in an ad hoc way. In this section we will be more rigorous, instead proving such triples (i.e. such program specifications) using a formal system of proof rules called a Hoare logic.

In particular, we will consider a Hoare logic for partial correctness. If a Hoare triple \{pre\} P \{post\} can be proven using the rules of this Hoare logic, we will write:

\[ \vdash \{pre\} P \{post\} \]

### Provability and Validity

Note carefully the distinction between \( \vdash \), provability, and \( \models \), validity. An important property, called soundness, is that every triple provable in a Hoare logic indeed holds in the desired sense of correctness. For our partial correctness Hoare logic, you can safely assume that it is sound, i.e.:

\[ \vdash \{pre\} P \{post\} \implies \models \{pre\} P \{post\} \]

Figure 1 contains the proof rules of a Hoare logic for partial correctness that should be used in the exercises overleaf. Let \( x \) denote a program variable, and \( e \) denote a side-effect free expression. Let the lower-case symbols \( p, p', q, q', r \) denote assertions, \( b \) a Boolean expression, and the upper-case symbols \( P, Q \) arbitrary programs.

A triple can be proven in this Hoare logic if it can be instantiated from an axiom ([ass] or [skip]), or if it can be derived as the conclusion of an inference rule ([comp], [if], [while], or [cons]). (A reminder of how to apply axioms and inference rules is given in the appendix of this problem sheet.)

\[
\begin{align*}
\text{[ass]} & \quad \vdash \{p[e/x]\} x := e \{p\} \\
\text{[skip]} & \quad \vdash \{p\} \text{skip} \{p\} \\
\text{[comp]} & \quad \vdash \{p\} P \{r\} \quad \vdash \{r\} Q \{q\} \quad \implies \quad \vdash \{p\} P; Q \{q\} \\
\text{[if]} & \quad \vdash \{b \land p\} P \{q\} \quad \vdash \{\neg b \land p\} Q \{q\} \\
& \quad \implies \quad \vdash \{p\} \text{if } b \text{ then } P \text{ else } Q \{q\} \\
\text{[while]} & \quad \vdash \{b \land p\} P \{p\} \\
& \quad \implies \quad \vdash \{p\} \text{while } b \text{ do } P \{\neg b \land p\} \\
\text{[cons]} & \quad p \Rightarrow p' \quad \vdash \{p'\} P \{q'\} \quad q' \Rightarrow q \\
& \quad \implies \quad \vdash \{p\} P \{q\}
\end{align*}
\]

Figure 1: A Hoare logic for partial correctness
2.1 Exercises

In the exercises that require a proof (i.e. \(\vdash\)), you should use the proof rules of the Hoare logic in Figure 1.

**Hint:** the appendix contains a recap on the different types of proof rules—axioms and inference rules—and how to apply them in order to prove a Hoare triple.

i. What does the rule of consequence, [cons], allow us to do in proofs? Why must we show that \(p \Rightarrow p'\) and \(q' \Rightarrow q\) are valid implications?

ii. Explain the intuition behind the axiom of assignment, [ass].

iii. Show that \(\vdash \{x > 0\} \ x := x + 1 \ \text{skip} \ \{x > 1\}\).

iv. Show that \(\vdash \{x = a \land y = b\} \ t := x; x := x + y; y := t \ \{x = a + b \land y = a\}\).

v. Explain the intuition behind the proof rule [while].

vi. Let \(INV\) denote \(in + m = 250\). Show that:

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

vii. Define a proof rule for the Pascal construct:

\[\text{repeat } P \text{ until } b\]

without reference to the [while] rule.

**Hint:** the loop body is always executed at least once.

viii. A new command has been added to the language: \texttt{surprise}. It has a very unusual semantics. Upon execution, \texttt{surprise} randomly chooses a variable in the program state, and randomly changes its assignment. Propose and justify an axiom for this command.

ix. (⋆) Propose an alternative, “forward” axiom of assignment in which a substitution is not applied to the precondition, i.e. replace the question marks in:

\[\vdash \{p\} \ x := e \ \{??\}\].

**Hint:** the postcondition contains an existential quantifier.
Appendix: Applying Proof Rules

We review in this appendix how to prove triples in Hoare logics through the application of axioms and inference rules.

The proof rules \([\text{skip}]\) and \([\text{ass}]\) of Figure 1 are examples of \textit{axioms} 1. Axioms describe “self-evident” facts about programs, in the sense that they can be applied without additional reasoning. A triple \(\{\text{pre}\} \ P \ \{\text{post}\}\) is \textit{proven in a Hoare logic}, denoted \(\vdash \{\text{pre}\} \ P \ \{\text{post}\}\), if it can be instantiated from such an axiom. For example, from the \([\text{skip}]\) axiom, we can immediately prove any triple for \texttt{skip} that has the same pre- and postcondition, for example:

\[
\vdash \{x = 5\} \ \text{skip} \ \{x = 5\}
\]
\[
\vdash \{\text{true}\} \ \text{skip} \ \{\text{true}\}
\]

Most triples, of course, cannot simply be instantiated from axioms. For these, we must apply \textit{inference rules}, such as \([\text{comp}]\), \([\text{if}]\), \([\text{while}]\), and \([\text{cons}]\) from Figure 1. Inference rules consist of at least one premise (given above the horizontal line), and exactly one conclusion (given underneath). If we can prove the premises, then we can deduce the triple in the conclusion. For example, to prove the triple:

\[
\{x > 0\} \ x := x + 1; \ \text{skip} \ \{x > 1\}
\]

simply instantiate \([\text{comp}]\) with this triple as the conclusion, and then proceed to prove the premises:

\[
\vdash \{x > 0\} \ x := x + 1 \ \{r\}
\]

and \(\vdash \{r\} \ \text{skip} \ \{x > 1\}\).

Here \(r\) is some assertion, to be determined, that holds at the midpoint of the two subprograms.

Visualising proofs is a matter of personal taste. For smaller proofs, I usually find it helpful to visualise them as \textit{proof trees}, with axiom instantiations as the leaves, applications of inference rules as the body, and the triple we want to prove as the root. For example, the above could be visualised as follows (note that we have taken \(r\) to be \(x < 1\)):

\[
\begin{array}{c}
\vdash \{x > 0\} \ x := x + 1 \ \{x > 1\} \\
\text{[comp]} \\
\vdash \{x > 0\} \ x := x + 1; \ \text{skip} \ \{x > 1\}
\end{array}
\]

\[
\begin{array}{c}
\vdash \{x > 1\} \ \text{skip} \ \{x > 1\} \\
\text{[skip]} \\
\vdash \{x > 1\} \ \text{skip} \ \{x > 1\}
\end{array}
\]

(The left-hand side of this proof tree is left incomplete because it forms part of an exercise!)

An inference rule we should give some special attention to is \([\text{cons}]\), the rule of consequence. Two of its premises require showing—outside of the Hoare logic—that some logical implications are valid. We will discuss this further in the exercises.

\footnote{More precisely, \textit{axiom schemata}, but most authors drop the latter part in their terminology.}
Problem Sheet 2: AutoProof

Chris Poskitt and Julian Tschannen
ETH Zürich

"Beware of bugs in the above code; I have only proved it correct, not tried it."
– Donald E. Knuth

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

1 Background

This exercise class is concerned with the AutoProof tool [2, 3], a static verifier for programs written in (a subset of) the object-oriented language Eiffel. The tool takes an Eiffel program—annotated with contracts (i.e. executable pre-/postconditions, class invariants, intermediate assertions)—and automatically attempts to verify the correctness of the program with respect to its contracts.

The tool is built on top of Boogie [1], an automatic verification framework developed by Microsoft Research. AutoProof translates Eiffel programs and their contracts (i.e. their proof obligations) into the front-end language of Boogie—an intermediate verification language encoding the semantics of the source program in terms of primitive constructs, and prescribing what it means for the source program to be correct. The Boogie tool then translates this intermediate program into a set of verification conditions: logical formulae which if valid, indicate the correctness of the source program. The validity of these verification conditions is checked automatically by an SMT solver (currently Z3).

This workflow is summarised in Figure 1. We will only be interacting with AutoProof itself in this exercise class, but it is helpful to be roughly aware of how it works and what translations it is performing (in a later class, we will look at the Boogie framework directly).
2 Exercises

The Eiffel programs for the following exercises are all available online in a web-based interface to AutoProof. Simply follow the given links for each exercise, edit the code and contracts in your browser, then hit the “Verify” button to run the tool.

Important: the web interface to AutoProof does not permanently save your changes, so please make sure to save local copies of your solutions.

i. Consider the class WRAPPING_COUNTER in:

http://cloudstudio.ethz.ch/e4pubs/#sv-task1

The method increment increases its integer input by one, except if the input is 59, in which case it wraps it round to 0. Verify the class in AutoProof without changing the implementation, i.e. adding only the necessary preconditions. Strengthen the postcondition further as suggested in the comments, and check that the proof still goes through.

ii. In the axiomatic semantics problem sheet, we encountered several simple program specifications expressed as Hoare triples. Using the class AXIOMATIC_SEMANTICS in:

http://cloudstudio.ethz.ch/e4pubs/#sv-task2

write some simple contract-equipped methods and show the following in AutoProof:

(A) \{ x = 21 \land y = 5 \} \mathsf{skip} \{ y = 5 \}
(B) \{ x > 10 \} x := 2 \times x \{ x > 21 \}
(C) \{ x \geq 0 \land y > 1 \} \mathsf{while} x < y \mathsf{do} x := x \times x \{ x \geq y \}
(D) \{ x = 5 \} \mathsf{while} x > 0 \mathsf{do} x := x + 1 \{ x < 0 \}
(E) \{ x = a \land y = b \} t := x; x := x + y; y := t \{ x = a + b \land y = a \}
(F) \{ in + m = 250 \} \mathsf{while} (i > 0) \mathsf{do} m := m + n; i := i - 1 \{ in + m = 250 \}

Hint: Eiffel does not offer a while construct. Try experimenting with from-until-loop instead, as well as if-then-else with recursion (note that recursive calls should be surrounded by wrap and unwrap so that the verifier checks the class invariant—see the code comments).

iii. Consider the class MAX_IN_ARRAY in:

http://cloudstudio.ethz.ch/e4pubs/#sv-task3

What does the max_in_array method do? Prove the class correct in AutoProof by determining a suitable precondition and loop invariant.

Hint: you might find Eiffel’s across-as-all loop construct\(^1\) helpful for expressing loop invariants.

---

\(^1\)See: http://bertrandmeyer.com/2010/01/26/more-expressive-loops-for-eiffel/
iv. (*) Consider the class `SUM_AND_MAX` in:
   \url{http://cloudstudio.ethz.ch/e4pubs/#sv-task4}
   What does the method `sum_and_max` do? What can you prove about it using AutoProof?

v. (**) Consider the class `LCP` in:
   \url{http://cloudstudio.ethz.ch/e4pubs/#sv-task5}
   The method `lcp` implements a Longest Common Prefix (LCP) algorithm\(^2\) with input and output as follows:

   \begin{center}
   \textit{Input:} an integer array \(a\), and two indices \(x\) and \(y\) into this array.
   
   \textit{Output:} length of the longest common prefix of the subarrays of \(a\) starting at \(x\) and \(y\) respectively.
   \end{center}

   What can you prove about the class in AutoProof?

References


\(^2\)From the FM 2012 verification challenge.
Problem Sheet 3: Separation Logic

Chris Poskitt
ETH Zürich

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

1 Separation Logic Assertions

These exercises are about assertions in separation logic, in particular, the separating conjunction ⇤. For a reminder of the semantics of assertions (as well as numerous examples), please refer to the first set of separation logic lecture slides:

http://se.inf.ethz.ch/courses/2014b_fall/sv/slides/05-SeparationLogic-1.pdf

i. Consider the following state:

Which of the following assertions hold in this state? For the ones that do not: why not?

(a) 3z. i ! z
(b) 3z. i ! z \& j ! z
(c) 3z. i ! z \& true
(d) 3z. (i ! z \& true) \& i = j \& (z ! 1, 2 \& true)
(e) 3x, x', y, z. i ! x \& j ! x' \& x ! 1, 2 \& j + 1 ! y \& y ! z, 5 \& z ! 3, 4
(f) true \& emp
ii. Do the following implications hold for all states and predicates \( p, q \)? If not, why not?

\[
p \implies p * p \\
p * q \implies [(p \land q) * true]
\]

2 Separation Logic Proofs

These exercises involve proving specifications of heap-manipulating programs using separation logic. For some Hoare-style proof rules, please refer to the first problem sheet. For the small axioms and the frame rule of separation logic, please refer to the first set of separation logic lecture slides:

http://se.inf.ethz.ch/courses/2014b_fall/sv/slides/05-SeparationLogic-1.pdf

The second set of lecture slides demonstrates their use on two simple heap-manipulating programs:

http://se.inf.ethz.ch/courses/2014b_fall/sv/slides/05-SeparationLogic-2.pdf

i. Consider the following program:

\[
\begin{align*}
l &:= \text{cons}(1); \\
r &:= \text{cons}(2,3); \\
temp1 &:= \text{[r+1]}; \\
temp2 &:= \text{[l]}; \\
\text{[l]} &:= temp1; \\
\text{[r]} &:= temp2;
\end{align*}
\]

Starting from precondition \{\text{emp}\}, apply the axioms and inference rules of separation logic to derive a postcondition expressing exactly the contents of the store and heap at termination. Then, depict this state using the store and heap diagrams presented in the lectures.

ii. We can assert that a heap portion contains a linked list by using the following inductively defined predicate:

\[
\begin{align*}
\text{list}(\text{[]},i) &\iff \text{emp} \land i = \text{nil} \\
\text{list}(a::as,i) &\iff \exists j. i \mapsto a, j * \text{list}(as,j)
\end{align*}
\]

where \text{nil} is a constant used to terminate the list.

Using the list predicate, verify the following program that deletes the first item in a non-empty linked list (assume that \( i \) points to the first node).

\[
\begin{align*}
dispose(i); \\
k &:= [i+1]; \\
dispose(i+1); \\
i &:= k;
\end{align*}
\]
iii. (⋆) Consider the following inductively defined predicate:

\[
\text{tree}(e, \tau) \iff \begin{cases}
\text{if } (\text{isAtom}(e) \land e = \tau) \text{ then emp} \\
\text{else } \exists x, y, \tau_1, \tau_2. \tau = (\tau_1, \tau_2) \\
\land e \mapsto x, y \ast \text{tree}(x, \tau_1) \ast \text{tree}(y, \tau_2)
\end{cases}
\]

where \(\text{tree}(p, \tau)\) holds if \(p\) points to a data structure in memory representing the “mathematical” tree \(\tau\) (i.e. the tree as an abstract mathematical object; not a representation in computer memory). Consider the following specification:

\[
\{\text{tree}(p, \tau)\} \text{ CopyTree}(p, q) \{\text{tree}(p, \tau) \ast \text{tree}(q, \tau)\}
\]

which expresses that the procedure CopyTree stores—in a separate portion of memory pointed to by \(q\)—a mathematically equivalent tree to the one pointed to by \(p\).

Define the procedure CopyTree and verify the specification.

**Hint:** the following derived axiom for heap lookups may simplify the proof:

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

provided that \(x\) does not appear free in \(e\) or \(e'\).
Problem Sheet 4: Boogie and Boogaloo

Chris Poskitt and Nadia Polikarpova
ETH Zürich

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

1 Introduction

In the problem class two weeks ago we worked with the AutoProof tool, a static verifier for Eiffel programs. Recall, as illustrated in Figure 1, that the tool performs this verification by translating Eiffel programs—and their contracts (i.e. their specifications)—into an intermediate verification language called Boogie. The Boogie verifier then performs a further translation into verification conditions; logical formulae which, if valid, imply the correctness of your original Eiffel program with respect to its contracts. An SMT solver is then used to try and automatically determine whether or not these formulae are valid, and hence whether or not the original program is correct.

While hopefully you were able to verify a number of Eiffel classes, you may have hit some problems, e.g. a lack of expressiveness in the specification language, or a lack of control over what was happening “under the hood”. In this problem class, we return to the issue of automatic program verification, but from a different angle; one that helps alleviate such concerns.

2 Boogie and Boogaloo

Rather than interacting with Boogie via AutoProof, now we will work with the Boogie verifier directly, writing programs and specifications directly in its front-end language [1]. Boogie provides a much richer specification and implementation language, including among its features: non-determinism, unbounded quantification, and infinite structures (e.g. sets). For an introduction to the language and verifier, consult the following slides and manual:

- Nadia Polikarpova’s Boogie/Boogaloo slides:
  http://se.inf.ethz.ch/courses/2014b_fall/sv/slides/boogie.pdf
- Boogie manual:

Boogie, unsurprisingly, suffers from two of the same problems as AutoProof: an absence of counterexamples when implementations do not agree with specifications, and little means of validation when the implementation is actually correct but the specification insufficient for the proof to go through automatically. Hence we will also be using the Boogaloo tool, which provides a means to automatically generate executions of Boogie programs, in order to better understand why a verification attempt is failing.

Boogaloo [2] is an interpreter and run-time assertion checker for Boogie. Based on a technique called symbolic execution, it enumerates all the possible paths of a Boogie program (there may be several due to the nondeterminism), but does so without having to enumerate all the possible inputs too: instead, it uses symbolic values—essentially “placeholders” for concrete
program values. Upon arriving at the end of a path, Boogaloo will have a constraint built up of such symbols to solve; each solution representing concrete values that variables could possibly have in this execution. Such solutions are reported to the user, providing either (1) concrete counterexamples for understanding why a verification attempt failed; or (2) passing test cases, providing some validation of the implementation. This workflow is summarised in Figure 2.

For an introduction to Boogaloo, and an informative, concise manual, please consult:

- Nadia Polikarpova's Boogie/Boogaloo slides:
  http://se.inf.ethz.ch/courses/2014b_fall/sv/slides/boogie.pdf

- Boogaloo wiki and manual:
  https://bitbucket.org/nadiapolikarpova/boogaloo/wiki/Home

3 Setting Up

Both Boogie and Boogaloo are available to try in your browser, through rise4fun and Comcom respectively:

http://rise4fun.com/boogie

http://cloudstudio.ethz.ch/comcom/#Boogaloo

Should you prefer, both tools can also be installed on your laptop. Please refer to their respective websites for details:


https://bitbucket.org/nadiapolikarpova/boogaloo/wiki/Home
4 Exercises

i. The Boogie program Fibonacci is buggy:
http://rise4fun.com/Boogie/v8IN

Use Boogaloo to debug it, verify it in Boogie, and then return to Boogaloo to generate some valid executions.

**Hint:** in the command box for Boogaloo on Comcom, type `test` for test mode, `~p PROC` to specify a procedure under test, and `~o N` to generate $N$ valid executions.

ii. Prove the following specification in Boogie:

$$\{\text{true}\} \ t := x; \ x := x + y; \ y := t \ \{x = x' + y' \land y = x'\}$$

where $x', y'$ respectively denote the values of $x$ and $y$ in the pre-state.

iii. Consider the Boogie program ArraySum which is supposed to recursively compute the sum of array elements:
http://rise4fun.com/Boogie/2kR

Fix the program and then verify it in Boogie.

**Hint:** don’t forget about loop invariants! Without an invariant, any loop in Boogie is treated as equivalent to assigning arbitrary values to program variables.

iv. Implement, test, and verify the algorithm FindZero (its signature is given in http://rise4fun.com/Boogie/SciP), that linearly searches an array for the element 0:

| Input: an integer array $a$, and its length $N$. |
| Output: an index $k \in \{0, \ldots, N-1\}$ into the array $a$ such that $a[k] = 0$; otherwise $k = -1$. |

The specification should guarantee that if there exists an array element $a[i] = 0$ with $0 \leq i < N$, then FindZero will always return a $k$ such that $k \geq 0$ and $a[k] = 0$.

v. (*) Copy the procedure FindZero you wrote in part (iv), rename it to FindZeroPro, and add the following two preconditions:

- `requires (forall i: int :: 0 <= i && i < N ==> 0 <= a[i]);`
- `requires (forall i: int :: 0 <= i-1 && i < N ==> a[i-1]-1 <= a[i]);`

These additional preconditions require that along the array, values never decrease by more than one. Adapt your linear search algorithm such that after an iteration of its loop, instead of incrementing the current index $k$ by 1, it now increments it by $a[k]$. Verify that the procedure still establishes the same postconditions as in (iv).

**Hint:** you will need to prove that all array values between $a[k]$ and $a[k+a[k]]$ are non-zero (i.e. that 0-values are not skipped over by the search) and use this property in the loop. For this you will need to write more than simply a loop invariant, e.g. some “ghost” (or “proof”) code.
vi. (*) Take a look at the Boogie program \texttt{BinarySearch} which is supposed to perform a binary search\textsuperscript{1}:

\url{http://rise4fun.com/Boogie/Ilf}

Debug the implementation and add the missing loop invariants.

References


\textsuperscript{1}See: \url{http://en.wikipedia.org/wiki/Binary_search_algorithm}
Problem Sheet 5: Program Proofs

Chris Poskitt
ETH Zürich

1 Axiomatic Semantics Recap

This section provides some additional questions on Hoare logic. The proof rules are given again in Figure 1.

i. Devise an axiom for the command $\text{havoc}(x_0, \ldots, x_n)$, which assigns arbitrary values to the variables $x_0, \ldots, x_n$.

ii. Write a program that computes the factorial of a natural number stored in variable $x$ and assigns the result to variable $y$. Prove that the program is correct using our Hoare logic.

iii. Define a proof rule for the $\text{from-until-loop}$ construct.

iv. Consider the following annotated program, where $A$ is an array indexed from 1 of element type $G$, $n$ is an integer variable storing $A$’s size, $k$ is another integer variable, $v$ is a variable of type $G$ initialised to some fixed value, and $\text{found}$ is a Boolean variable.

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

(a) What does the program do? In particular, what does the value of $k$ represent on exit?

(b) Prove the triple using the axioms and inference rules of Hoare logic.

v. Sarah Proofgood has successfully shown that given an arbitrary program $P$ and postcondition $\text{post}$, the triple:

\[ \{ \text{WP}[P, \text{post}] \} \ P \{ \text{post} \} \]

can be proven in our Hoare logic, i.e. $\vdash \{ \text{WP}[P, \text{post}] \} \ P \{ \text{post} \}$. Here, $\text{WP}[P, \text{post}]$ is an assertion expressing the weakest (liberal) precondition relative to $P$ and $\text{post}$; that is, the weakest condition that must be satisfied for $P$ to establish $\text{post}$ (without guaranteeing termination).

Using Sarah’s result, show that any valid triple $\models \{ p \} \ P \{ q \}$ is provable in our Hoare logic, i.e. $\vdash \{ p \} \ P \{ q \}$.
2 Separation Logic Recap

This section provides some additional practice on using separation logic. The small axioms and frame rule of separation logic are given in Figure 2.

i. Are the following assertions satisfiable? Justify your answers.

\[ p \land \neg p \]
\[ x = y \land \neg(x = y) \]

ii. Consider the following program state:

Which of the following assertions does this state satisfy? For the assertions it does not satisfy: why not?

(a) \( \exists v. x \mapsto v \land v \mapsto v \)
(b) \( \exists v. x \mapsto v \land v \mapsto v \land y \mapsto v \)
(c) \( y \mapsto _- \)
(d) \( (x = y) \land (y \mapsto _- \land true) \)
(e) \( (x = y) \land true \)

iii. Starting from precondition \{\text{emp}\}, apply the axioms and inference rules of separation logic to derive a postcondition expressing exactly the contents of the store and heap at termination (assume that \( x \) and \( y \) are the only variables). Then, depict this state using the store and heap diagrams presented in the lectures.

\[
\begin{align*}
x &:= \text{cons}(5,9); \\
y &:= \text{cons}(6,7); \\
x &:= [x]; \\
[y+1] &:= 9; \\
\text{dispose}(y); \\
\end{align*}
\]
iv. A well-formed binary tree $t$ is defined by the grammar:

\[ t \triangleq n \mid (t_1, t_2) \]

i.e. $t$ can be either a leaf, which is a single number $n$, or an internal node with a left subtree $t_1$ and a right subtree $t_2$. Consider the following definition of the inductive predicate $tree(t, i)$ which asserts that $i$ is a pointer to a well-formed binary tree $t$:

\[
\begin{align*}
tree(n, i) & \triangleq i \mapsto n \\
tree((t_1, t_2), i) & \triangleq \exists l, r. \ i \mapsto l, r \ast tree(t_1, l) \ast tree(t_2, r)
\end{align*}
\]

Using these definitions, give a proof outline of the following triple. There must be at least one assertion between every two commands.

\[
\begin{align*}
\{ & tree((1, t), i) \} \\
& x := \llbracket i \rrbracket; \ \\
& \llbracket i \rrbracket := 2; \ \\
& y := \llbracket i+1 \rrbracket; \ \\
& dispose(i); \ \\
& dispose(x); \ \\
& dispose(i+1); \ \\
\{ & tree(t, y) \}
\end{align*}
\]
Appendix: Proof Rules

\[
\begin{align*}
[\text{ass}] & \vdash \{p[e/x]\} \ x := e \ \{p\} \\
[\text{skip}] & \vdash \{p\} \ \text{skip} \ \{p\} \\
[\text{comp}] & \vdash \{p\} \ P \ \{r\} \quad \vdash \{r\} \ Q \ \{q\} \quad \vdash \{p\} \ P; \ Q \ \{q\} \\
[\text{if}] & \vdash \{b \land p\} \ P \ \{q\} \quad \vdash \{\neg b \land p\} \ Q \ \{q\} \\
[\text{while}] & \vdash \{b \land p\} \ P \ \{p\} \\
[\text{cons}] & \vdash \{p \Rightarrow p'\} \ P \ \{q'\} \quad q' \Rightarrow q \\
& \vdash \{p\} \ P \ \{q\}
\end{align*}
\]

Figure 1: A Hoare logic for partial correctness

\[
\begin{align*}
\vdash \{e \mapsto \_\} \ e := f \ \{e \mapsto f\} \\
\vdash \{e \mapsto \_\} \ \text{dispose}(e) \ \{\text{emp}\} \\
\vdash \{X = x \land e \mapsto Y\} \ x := \{e[X/x] \mapsto Y \land Y = x\} \\
\vdash \{\text{emp}\} \ x := \text{cons}(e_0, \ldots, e_n) \ \{x \mapsto e_0, \ldots, e_n\} \\
& \vdash \{p\} \ P \ \{q\} \\
& \vdash \{p \ast r\} \ P \ \{q \ast r\}
\end{align*}
\]

side condition: no variable modified by \(P\) appears free in \(r\)

Figure 2: The small axioms and frame rule of separation logic
Problem Sheet 6: Data Flow Analysis

Chris Poskitt*
ETH Zürich

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

1 Reaching Definitions Analysis

These exercises are based on the material from the “Reaching Definitions Analysis” section of this lecture:

http://se.inf.ethz.ch/courses/2014b_fall/sv/slides/08-ProgramAnalysis.pdf

Consider the following program fragment:

```plaintext
x := 10;
while x > 0 do
    y := 2 * y;
    if y > 10 do
        x := x - 1;
    else
        y := x + 2;
    end
end
x := x - 1;
```

i. Draw the control flow graph of the program fragment.

ii. Annotate the control flow graph with the results of a reaching definitions analysis.

iii. Provide (or draw) the use-definition information for program variables x and y.

*These exercises are from previous iterations of the course when Stephan van Staden was the teaching assistant.
2 Live Variables Analysis

These exercises are based on the material from the “Live Variables Analysis” and “Equation Solving” sections of this lecture:

http://se.inf.ethz.ch/courses/2014b_fall/sv/slides/08-ProgramAnalysis.pdf

Consider the following program fragment:

\[
x := y; \\
x := x - 1; \\
x := 4; \\
\text{while } y < x \text{ do} \\
\quad y := y + x; \\
end \\
y := 0;
\]

i. Identify the elementary blocks of the program and label them.

ii. Write down the equations for a live variables analysis of the program.

iii. Solve the data flow equations using chaotic iteration.

iv. Using the result obtained in (iii), perform dead code elimination on the program fragment.

v. (⇤) Is the program resulting in step (iv) free of dead variables? If not, explain why and modify the live variables analysis so that it can be used to produce a program free of dead variables.
Problem Sheet 7: Program Slicing
and Abstract Interpretation

Chris Poskitt*
ETH Zürich

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

1 Program Slicing

These exercises are based on the material from the “Program Slicing” section of this lecture:

http://se.inf.ethz.ch/courses/2014b_fall/sv/slides/09-ProgramAnalysis-Slicing.pdf

Consider the following program fragment:

\[
\begin{align*}
x &:= 0; \\
y &:= 0; \\
i &:= n; \\
j &:= n; \\
\text{while } i > 0 \text{ do} \\
\quad x &:= x + 1; \\
\quad i &:= i - 1; \\
\quad j &:= i; \\
\quad \text{while } j > 0 \text{ do} \\
\quad\quad y &:= y + 1; \\
\quad\quad j &:= j - 1; \\
\text{end} \\
\text{end} \\
\text{print}(x); \\
\text{print}(y);
\end{align*}
\]

i. Draw the program dependence graph for this fragment.

ii. Compute the backward slice of the program fragment for the slicing criteria \texttt{print(x)} and \texttt{print(y)}.
2 Abstract Interpretation

These exercises are based on the material from the “Abstract Interpretation” lecture:

http://se.inf.ethz.ch/courses/2014b_fall/sv/slides/10-AbstractInterpretation.pdf

Consider again the factorial algorithm from the lecture with sign analysis equations:

\[
\begin{align*}
A_1 &= [x \mapsto +, y \mapsto T] \\
A_2 &= A_1[y \mapsto +] \cup A_4[x \mapsto A_4(x) \oplus +] \\
A_3 &= A_2 \\
A_4 &= A_3[y \mapsto A_3(x) \otimes A_3(y)] \\
A_5 &= A_2 \sqcap [x \mapsto 0, y \mapsto T]
\end{align*}
\]

i. Compute the analysis result by chaotic iteration.

ii. Is the analysis precise? What is it unable to prove about the program?

iii. Improve the precision by:

(a) Changing the program but not the analysis (i.e. compute the factorial in a way that is more “friendly” for the analysis).

(b) (*) Changing the analysis but not the program.
Problem Sheet 8: Model Checking

Chris Poskitt and Carlo A. Furia
ETH Zürich

The exercises in this problem sheet are all based on the first set of model checking slides:

http://se.inf.ethz.ch/courses/2014b_fall/sv/slides/11-ModelChecking.pdf

1 Evaluating LTL Formulae on Automata

Consider the the following automaton:

Do the following properties hold? Justify your judgements.

i. □ (start → ◇ stop)
ii. □ ◇ turn_off
iii. □ ◇ (turn_off ∨ push)
iv. ◇ (turn_off ∨ push)
 v. (□ false) ∨ ◇ (turn_off ∨ push)
vi. (turn_on U start) ∨ (pull U push)
2 Equivalence of LTL Formulae

These exercises are about proving the equivalence of LTL formulae. In the first two of these exercises, you will be proving that the operators ◊ and □ are derived, i.e. they can be defined entirely in terms of the core LTL operators.

i. Recall that \( w, i \models ◊F \) if there exists some \( j \) such that \( i \leq j \leq n \) and \( w, j \models F \).
   Prove that this is also the case for \( w, i \models \text{true} U F \) (i.e. that this is an equivalent way of expressing ◊F as a derived operator).

ii. Recall that \( w, i \models □F \) if for all \( j \) such that \( i \leq j \leq n \), \( w, j \models F \).
   Prove that this is also the case for \( w, i \models \neg ◊\neg F \) (i.e. that this is an equivalent way of expressing □F in terms of other LTL operators).

iii. Prove that ◊ is idempotent, i.e. that ◊p is equivalent to ◊◊p.

3 Automata-Based Model Checking

Let us prove by model checking that □ ◊ turn_off is not a property of the microwave oven automaton in Section 1.

i. Build an automaton with the same language as ¬(□ ◊ turn_off).
   **Hint:** start with the non-negated formula and then invert the accepting and non-accepting states of its automaton.

ii. Compute the intersection of the automaton you built in part (i) and the microwave oven automaton.

iii. Check the intersection automaton for accepting runs, using them to prove that □ ◊ turn_off is not a property of the microwave oven automaton.
Problem Sheet 9: Software Model Checking

Chris Poskitt*
ETH Zürich

The exercises in this problem sheet are based on the software model checking slides:

http://se.inf.ethz.ch/courses/2014b_fall/sv/slides/12-SoftwareModelChecking.pdf

1 Predicate Abstraction

Recall that:

- $\text{Pred}(f)$ denotes the weakest under-approximation of the expression $f$ expressible as a Boolean combination of the given predicates.
- The Boolean abstraction of an $\text{assume } c \text{ end}$ statement is $\text{assume not Pred(not } c) \text{ end}$ followed by a parallel conditional assignment updating the predicates with respect to the original $\text{assume}$ statement.
- The Boolean abstraction of an $\text{assert } c \text{ end}$ statement is $\text{assert Pred}(c) \text{ end}$.

Exercises

i. Justify, using Venn diagrams, the use of double negation in the Boolean abstraction of $\text{assume}$ statements.

ii. Consider the following code snippet $C_1$, where $x, y, z$ are integer variables:

$\text{assume } x > 0 \text{ end}$
$z := (x*y) + 1$
$\text{assert } z \geq 1 \text{ end}$

Build the Boolean abstraction $A_1$ of the code snippet $C_1$ with respect to the following set of predicates:

$p \triangleq x > 0$
$q \triangleq y > 0$
$r \triangleq z > 0$

*Some exercises adapted from ones written by Stephan van Staden and Carlo A. Furia.
iii. Consider the following code snippet $C_2$, where $x, y$ are integer variables:

```plaintext
if x > 0 then
  y := x + x
else
  if x = 0 then
    y := 1
  else
    y := x * x
  end
end
assert y > 0 end
```

(a) Normalise the guards of conditionals using nondeterminism and assume statements.

(b) Build the Boolean abstraction $A_2$ of the normalised code snippet $C_2$ with respect to the following set of predicates:

$$p \triangleq x > 0$$
$$q \triangleq y > 0$$

2 Error Traces

i. Provide an annotated trace for the Boolean abstraction $A_1$, and a corresponding annotated trace for the concrete program $C_1$ that is feasible such that `assert z >= 1` end evaluates to false when reached.

ii. Can you verify the Boolean abstraction $A_2$? If not, give a counterexample path and prove whether or not it is spurious.
Problem Sheet 10: Verification of Real-Time Systems

Chris Poskitt and Carlo A. Furia

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

The following exercises are based on the lectures about verifying real-time systems:


Assume that the time domain consists of exactly the non-negative real numbers.

1 MTL Property Checking

Consider first the following timed automaton:

Do the following properties hold?

i. □ a

ii. □ (◊ = 1 a)

iii. □ (□ = 1 a)
Consider now the following timed automaton:

![Timed Automaton Diagram]

Do the following properties hold?

iv. $\square (a \rightarrow \diamond (0, 1) c)$

v. $\square (a \rightarrow \diamond (0, 1) b)$

vi. $\square (a \rightarrow (a \lor b) U(0, 1) c)$

vii. $\square (a \rightarrow (a \lor b) U(1, 2) c)$

2 Region Automaton Construction

i. Construct the region automaton for the first timed automaton in Section 1.

ii. Construct the region automaton for the second timed automaton in Section 1.

iii. ($\ast$) Construct the region automaton for the following timed automaton (exercise taken from Alur & Dill, 1994):

![Region Automaton Diagram]
3 Semantics of MTL Formulae

i. Is the formula $\square \Diamond > 0$ true satisfied by any timed word?

ii. Is the formula $\square \Diamond \geq 0$ true satisfied by any timed word?

iii. Is $\Diamond [a, b] \Diamond [c, d] q$ equivalent or non-equivalent to $\Diamond [a + c, b + d] q$ for all $0 \leq a \leq b \leq c \leq d$?
Problem Sheet 11: Testing

Chris Poskitt*
ETH Zürich

For both of the algorithms, answer the coverage-related questions that follow:

Algorithm 1:
String function(int x, int y)
{
    boolean z;
    if (x < y)
        z := true
    else
        z := false
    if (z && x+y == 10)
        result := "a"
    else
        result := "b"
}

Algorithm 2:
if x > 0 then
    y := x + x
    while y < 15 do
        y := y + 2
    end
else
    if x = 0 then
        y := 1
    else
        y := x * x
    end
end

1 Branch and Path Coverage
i. How many branches are present?
ii. Is it possible to test every branch? Provide a set of tests to exercise as many branches as possible.
iii. How many paths are present?
iv. Which path(s) remains untested by your tests in part (ii)?
v. Add tests, if needed, to achieve path coverage.

2 Logic Coverage
i. Write down the predicates that occur in the code.
ii. Is it possible to obtain full predicate coverage? Provide a set of tests that will obtain the highest predicate coverage.
iii. Write down the clauses appearing in the code.
iv. Can we exercise full clause coverage? Write tests for maximal clause coverage.
v. In general, does predicate and/or clause coverage imply branch coverage?

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