Program Specifications

Specifying Kotlin Programs

Challenge: The execution of some Kotlin code can modify the program state in a way that depends on conditions *before* the execution.

We specify this in the following way:

```
// PRE: P
somecode
// POST: Q
```

The above specification expresses that if the state satisfies P, then after the execution of `somecode`, the state will satisfy Q.

- We call P the *pre-condition* of `somecode`
- We call Q the *post-condition* of `somecode`

Specification of Kotlin Programs

We can generalise this behaviour over arbitrary integers u, and v.

```
// PRE: a = u ∧ x = v
val y = a + x
// POST: a = u ∧ x = v ∧ y = u + v
```

The specification above should be read as follows:

> For any integers u, and v, if we start with a program state where `a` has the value u and `x` has the value v, and we execute the given code, then we will reach a program state where `a` still has the value u, `x` still has the value v and `y` now has the value $u + v$.

Important: There is an implicit universal quantification of u and v over the whole specification. We also assume that there was no `int` overflow.

Proof Obligations - Hoare Triples

To prove the specification:

```
// PRE: P
code
// POST: Q
```

we have to show that:

"If property P holds before the execution of `code`*, then after the execution of* `code` *property Q will hold".*

We can formally express this as a *Hoare Triple*:

$$\{P\} \quad \texttt{code} \quad \{Q\}$$

For example:

$$\{\ true\ \} \quad \texttt{x = 5} \quad \{\ \mathrm{x} > 0\ \}$$

$$\{\ 0 \leq \mathrm{x} < 10\ \} \quad \texttt{x++} \quad \{\ 0 \leq \mathrm{x} \leq 10\ \}$$

Proving the correctness of a Hoare triple:

$$\{P\} \quad \texttt{code} \quad \{Q\}$$

is essentially a proof that P modified by the **effects** of `code` implies Q.

For example, to prove:

$$\{\ true\ \} \quad \texttt{x = 5} \quad \{\ \mathrm{x} > 0\ \}$$

we would need to show that:

$$true \ \wedge\ \mathrm{x} = 5 \quad \longrightarrow \quad \mathrm{x} > 0$$

Important: For all of our proofs, we assume that the code compiles correctly (i.e. it is syntactically and semantically valid). We do not attempt to prove the correctness of invalid code (fix it first!).

Notational Conventions - Variables in Proofs

Some proof obligations present a problem when it comes to the meaning of program variables. For example, in

$$\{\ 0 \leq \mathrm{x} < 10\ \} \quad \texttt{x++} \quad \{\ 0 \leq \mathrm{x} \leq 10\ \}$$

the variable $\mathtt{x}$ appears in the pre-condition, the post-condition *and* it is modified by the code.

Naively, we might try to show that:

$$0 \leq \mathrm{x} < 10 \ \wedge\ \mathrm{x} = \mathrm{x} + 1 \quad \longrightarrow \quad 0 \leq \mathrm{x} \leq 10$$

To make things clear, we use the following notational convention:

- x will always refer to the most recent value of $\mathtt{x}$ (i.e. *after* the code has been executed).
- x_{old} refers to the value of $\mathtt{x}$ *before* the code is executed.

So now to prove:

$$\{\ 0 \leq \mathrm{x} < 10\ \} \quad \texttt{x++} \quad \{\ 0 \leq \mathrm{x} \leq 10\ \}$$

we would need to show that:

$$0 \leq \mathrm{x}_{old} < 10 \ \wedge\ \mathrm{x} = \mathrm{x}_{old} + 1 \quad \longrightarrow \quad 0 \leq \mathrm{x} \leq 10$$

To simply the presentation, we will only use the $_{old}$ annotation when a variable is **actually** modified by the code.

For example, to prove:

$$\{\ \mathrm{x} = 1\ \wedge\ \mathrm{y} = 2\ \} \quad \texttt{x = x + y} \quad \{\ \mathrm{x} = 3\ \wedge\ \mathrm{y} = 2\ \}$$

we would need to show that:

$$\mathrm{x}_{old} = 1\ \wedge\ \mathrm{y} = 2\ \wedge\ \mathrm{x} = \mathrm{x}_{old} + \mathrm{y} \quad \longrightarrow \quad \mathrm{x} = 3\ \wedge\ \mathrm{y} = 2$$

We also apply this convention to array contents.

For example, to prove:

$$\{\ \texttt{a[k]} = 0\ \}\quad \texttt{a[k] = a[k] + 5}\quad \{\ \texttt{a[k]} = 5\ \}$$

we would need to show that:

$$\texttt{a[k]}_{old} = 0\ \wedge\ \texttt{a[k]} = \texttt{a[k]}_{old} + 5 \quad\longrightarrow\quad \texttt{a[k]} = 5$$

Important: $\texttt{a[k]}_{old}$ is not the same as $\texttt{a[k}_{old}\texttt{]}$

- $\texttt{a[k]}_{old}$ refers to the value stored in array `a`, at index `k`, *before* the code is executed.
- $\texttt{a[k}_{old}\texttt{]}$ refers to the current value stored in array `a`, at the index that is stored in `k` *before* the code is executed.

To help us track when use of the $_{old}$ annotation is needed, we formally define the set of variables/object-attributes that are modified by a piece of code via the Mod function ($Mod : \mathbb{C} \to \wp ID$):

$$\begin{array}{rcl}
Mod(\,) & = & \{\} \\
Mod(\texttt{var x = } E)^1 & = & \{\} \\
Mod(\texttt{x = } E) & = & \{\texttt{x}\} \\
Mod(\texttt{i++}) & = & \{\texttt{i}\} \\
Mod(\texttt{i--}) & = & \{\texttt{i}\} \\
Mod(\texttt{a[k] = } E) & = & \{\texttt{a[k]}\} \\
Mod(\mathbb{C}_1\ \texttt{;}\ \mathbb{C}_2) & = & Mod(\mathbb{C}_1) \cup Mod(\mathbb{C}_2) \\
Mod(\texttt{if}(E)\{\mathbb{C}_1\}\texttt{else}\{\mathbb{C}_2\}) & = & Mod(\mathbb{C}_1) \cup Mod(\mathbb{C}_2) \\
Mod(\texttt{while}(E)\{\mathbb{C}\}) & = & Mod(\mathbb{C})
\end{array}$$

where ID is the set of variables/identifiers, E is an arbitrary side-effect free expression and $\mathbb{C}$, $\mathbb{C}_1$ and $\mathbb{C}_2$ are arbitrary sequences of instructions.

[1] similarly for $Mod(\texttt{val x = } E)$

Hoare Logic

In 1969 Sir Tony Hoare developed a new logic which allows for formal reasoning *directly* with Hoare Triples (hence the name).
For example there is an axiom for dealing with assignment:

$$\frac{P[\mathtt{x} \mapsto \mathtt{x}_{old}] \land \mathtt{x} = E[\mathtt{x} \mapsto \mathtt{x}_{old}] \ \longrightarrow \ Q}{\{\, P \,\} \ \ \mathtt{x = } E \ \ \{\, Q \,\}}$$

This axiom states that after the assignment we can establish any property that is derivable from the pre-condition and the effect of the assignment. The substitution $[\mathtt{x} \mapsto \mathtt{x}_{old}]$ reflects the modification of the variable $\mathtt{x}$.
For example:

$$\{\, \mathtt{x} = 5 \,\} \ \ \mathtt{x = x + 2} \ \ \{\, \mathtt{x} = 7 \,\}$$

can be proven by showing that:

$$\mathtt{x}_{old} = 5 \ \land \ \mathtt{x} = \mathtt{x}_{old} + 2 \ \longrightarrow \ \mathtt{x} = 7$$

premise / conclusion

Straight Line Code

How do we reason about *sequences* of state-changing statements?

We reason with *Mid-conditions* – checkpoints in the code.
For example:

```
// PRE: P
code1
// MID: R
code2
// POST: Q
```

$$\frac{\{\, P \,\} \ \ \mathtt{code1} \ \ \{\, R \,\} \qquad \{\, R \,\} \ \ \mathtt{code2} \ \ \{\, Q \,\}}{\{\, P \,\} \ \ \mathtt{code1} \ \ ; \ \ \mathtt{code2} \ \ \{\, Q \,\}}$$

Straight Line Code: An Example

```
1   // PRE: a = x ∧ b = y ∧ c = z              (P)
2   c = a * b
3   // MID: a = x ∧ b = y ∧ c = xy             (M_1)
4   b = b * b
5   // MID: a = x ∧ b = y^2 ∧ c = xy           (M_2)
6   a = a * a
7   // MID: a = x^2 ∧ b = y^2 ∧ c = xy         (M_3)
8   c = c + c
9   // MID: a = x^2 ∧ b = y^2 ∧ c = 2xy        (M_4)
10  val result = a + b + c
11  //POST: result = (x + y)^2                 (Q)
```

Change result to $x^2 + 2xy + y^2$

The proof obligations at the informal level:

- **line 2:** The pre-condition P and code line 2 must establish the mid-condition M_1.
  $P[\mathtt{c} \mapsto \mathtt{c}_{old}] \wedge \mathtt{c = a * b} \longrightarrow M_1$
- **line 4:** The mid-condition M_1 and code line 4 must establish the mid-condition M_2.
  $M_1[\mathtt{b} \mapsto \mathtt{b}_{old}] \wedge \mathtt{b = b * b} \longrightarrow M_2$
- **line 6:** The mid-condition M_2 and code line 6 must establish the mid-condition M_3.
  $M_2[\mathtt{a} \mapsto \mathtt{a}_{old}] \wedge \mathtt{a = a * a} \longrightarrow M_3$
- **line 8:** The mid-condition M_3 and code line 8 must establish the mid-condition M_4.
  $M_3[\mathtt{c} \mapsto \mathtt{c}_{old}] \wedge \mathtt{c = c + c} \longrightarrow M_4$
- **line 10:** The mid-condition M_4 and code line 10 must establish the post-condition Q.
  $M_4 \wedge \mathtt{val\ result = a + b + c} \longrightarrow Q$

Never needed in the exam, but makes what you are doing clearer

Full Logical Assertions:

- **line 2:** The pre-condition P and code line 2 must establish the mid-condition M_1.

$$\mathtt{a} = x \;\wedge\; \mathtt{b} = y \;\wedge\; \mathtt{c}_{old} = z \;\wedge\; \mathtt{c} = \mathtt{a} * \mathtt{b} \quad\longrightarrow\quad \mathtt{a} = x \;\wedge\; \mathtt{b} = y \;\wedge\; \mathtt{c} = xy$$

- **line 4:** The mid-condition M_1 and code line 4 must establish the mid-condition M_2.

$$\mathtt{a} = x \;\wedge\; \mathtt{b}_{old} = y \;\wedge\; \mathtt{c} = xy \;\wedge\; \mathtt{b} = \mathtt{b}_{old} * \mathtt{b}_{old} \quad\longrightarrow\quad \mathtt{a} = x \;\wedge\; \mathtt{b} = y^2 \;\wedge\; \mathtt{c} = xy$$

- **line 6:** The mid-condition M_2 and code line 6 must establish the mid-condition M_3.

$$\mathtt{a}_{old} = x \;\wedge\; \mathtt{b} = y^2 \;\wedge\; \mathtt{c} = xy \;\wedge\; \mathtt{a} = \mathtt{a}_{old} * \mathtt{a}_{old} \quad\longrightarrow\quad \mathtt{a} = x^2 \;\wedge\; \mathtt{b} = y^2 \;\wedge\; \mathtt{c} = xy$$

- **line 8:** The mid-condition M_3 and code line 8 must establish the mid-condition M_4.

$$\mathtt{a} = x^2 \;\wedge\; \mathtt{b} = y^2 \;\wedge\; \mathtt{c}_{old} = xy \;\wedge\; \mathtt{c} = \mathtt{c}_{old} + \mathtt{c}_{old} \quad\longrightarrow\quad \mathtt{a} = x^2 \;\wedge\; \mathtt{b} = y^2 \;\wedge\; \mathtt{c} = 2xy$$

- **line 10:** The mid-condition M_4 and code line 10 must establish the post-condition Q.

$$\mathtt{a} = x^2 \;\wedge\; \mathtt{b} = y^2 \;\wedge\; \mathtt{c} = 2xy \;\wedge\; \mathtt{result} = \mathtt{a} + \mathtt{b} + \mathtt{c} \quad\longrightarrow\quad \mathtt{result} = (x + y)^2$$

This is the proper stuff.

Pre-/Post-/Mid-conditions

- **Pre-condition**:
  - required to hold before some code is run
  - an assumption that the code can make
- **Post-condition**:
  - expected to hold after the code has been executed (assuming termination and pre-condition held before)
  - a guarantee that the code must make
- **Mid-condition**:
  - an assumption made at a specific point in the code
  - must be guaranteed by preceding code
  - can be assumed by subsequent code
  - a “stepping stone” in reasoning about correctness

Important: mid-conditions are actually post-conditions for the preceding lines of code and pre-conditions for the subsequent lines of code.

Function Specifications

Function specifications are given in terms of pre- and post- conditions:

```
fun someFunc(x1: type, ... , xn: type): type
// PRE: P
// POST: Q
{
    ...
}
```

The above specification promises that:

for all $v_1, ..., v_n$
if $P[\mathtt{x}_1 \mapsto v_1, ..., \mathtt{x}_n \mapsto v_n]$ holds before a call to $\mathtt{someFunc}(v_1, ..., v_n)$
then $Q[\mathtt{x}_1 \mapsto v_1, ..., \mathtt{x}_n \mapsto v_n]$ will hold upon return.

In the above we write $P\,[x_1 \rightarrow v_1, \ldots, x_n \rightarrow v_n]$ to denote the predicate P with all free occurrences of $x_1 \ldots x_n$ replaced with $v_1, \ldots v_n$ respectively.

Note that $v_1, \ldots, v_n$ are values, while $x_1, \ldots, x_n$ are program variables. We will sometimes refer to such lists of values/variables with vector notation to simplify the presentation. e.g. $\overline{x}$ or $\overline{v}$.

Notational Conventions - Variables in Specifications

Just as in our proofs, it is important to be precise about the meaning of variables in our specifications.

- We use a `monospace` font when referring to program variables,
  e.g. `x`, `i` or `count`
- We use an *italic* font when refering to value variables,
  e.g. u or v
- In pre-/post-/mid-conditions we use the subscript $_{pre}$ to refer to the initial value of an input variable on entry to the function,
  e.g. $\mathtt{x}_{pre}$ or $\mathtt{a}_{pre}$
- In post-conditions we use a **bold** r to refer to the return value of the function (if there is one),
  e.g. $\mathbf{r}$

Important: all specifications refer to the **current** program state, so we **never** use $\mathtt{x}_{old}$ in our assertions.

Obviously, making such type-setting distinctions by hand is rather tricky. We encourage you to use different variables (as we have also done) to avoid any potential confusion. By convention, we will only use the pre annotations when a variable (or an array's contents) could actually be modified by the code. In particular, Kotlin is call by value, so no variable of any primitive type (int, bool, char, etc.) can be updated by a function call. We do not, therefore need to annotate such variables in our function specifications.

Function Specifications - Example

```
fun squareOfSum(x: Int, y: Int): Int
// PRE: true
// POST: r = (x + y)^2
{
    var c = x * y
    val b = y * y
    val a = x * x
    c = c + c
    return a + b + c
}
```

Function Bodies

How do we prove that a function satisfies its specification?

```
fun someFunc(x1: type, ..., xn: type): type
// PRE: P
// POST: Q
{
    code
}
```

If property P holds before the execution of code then after the execution of code property Q must hold.

i.e.

$$\{\ P\ \}\quad \text{code}\quad \{\ Q\ \}$$

The above assumes that there is no shadowing of the function parameters within the function body (which is good practice), otherwise some substitutions ($[\overline{\mathrm{x}} \rightarrow \mathrm{x}_{\overline{\mathrm{pre}}}]$) would be needed on P and Q to respect Kotlin's call-by-value semantics (that the code inside the function body cannot change the values that were passed to it). We will see later that we can still reason about functions that shadow their parameters in their body, but doing so introduces extra notational workload for the prover.

If the body of the function consists of multiple lines of code?

```
fun someFunc(x1: type, ..., xn: type): type
// PRE: P
// POST: Q
{   // MID: R
    code1
    // MID: S
    code2
    // MID: T
}
```

Then, as before, we introduce appropriate mid-conditions, such that:

1. We can establish R from P.
2. If R holds before the execution of `code1`, then S holds after.
3. If S holds before the execution of `code2`, then T holds after.
4. We can establish Q from T.

Using properties R and T helps us to account for the function parameter book-keeping. Sometimes we omit properties R and T if the code's behaviour is straight-forward.

Function Bodies - Example

```
1   fun squareOfSum(x: Int, y: Int): Int
2   // PRE: true                                  (P)
3   // POST: r = (x + y)^2                        (Q)
4   {
5       var c = x * y
6       // MID: c = xy                            (M1)
7       val b = y * y
8       // MID: b = y^2 ∧ c = xy                  (M2)
9       val a = x * x
10      // MID: a = x^2 ∧ b = y^2 ∧ c = xy        (M3)
11      c = c + c
12      // MID: a = x^2 ∧ b = y^2 ∧ c = 2xy       (M4)
13      return a + b + c
14  }
```

Informally:

- **line 5:** The pre-condition P and code line 5 must establish the mid-condition M_1.
  $P \ \wedge$ `var c = x * y` $\longrightarrow \ M_1$
- **line 7:** The mid-condition $M1$ and code line 7 must establish the mid-condition M_2.
  $M_1 \ \wedge$ `val b = y * y` $\longrightarrow \ M_2$
- **line 9:** The mid-condition $M2$ and code line 9 must establish the mid-condition M_3.
  $M_2 \ \wedge$ `val a = x * x` $\longrightarrow \ M_3$
- **line 11:** The mid-condition $M3$ and code line 11 must establish the mid-condition M_4.
  $M_3[\mathtt{c} \mapsto \mathtt{c}_{old}] \ \wedge$ `c = c + c` $\longrightarrow \ M_4$
- **line 13:** The mid-condition M_4 and code line 13 must establish the post-condition Q.
  $M_4 \ \wedge$ `return a + b + c` $\longrightarrow \ Q$

Formally:

- **line 5:** The pre-condition P and code line 5 must establish the mid-condition M_1.

$$\texttt{true} \ \wedge \ \texttt{c} = \texttt{x} * \texttt{y}$$
$$\longrightarrow$$
$$\texttt{c} = \texttt{xy}$$

- **line 7:** The mid-condition M_1 and code line 7 must establish the mid-condition M_2.

$$\texttt{c} = \texttt{xy} \ \wedge \ \texttt{b} = \texttt{y} * \texttt{y}$$
$$\longrightarrow$$
$$\texttt{b} = \texttt{y}^2 \ \wedge \ \texttt{c} = \texttt{xy}$$

- **line 9:** The mid-condition M_2 and code line 9 must establish the mid-condition M_3.

$$\texttt{b} = \texttt{y}^2 \ \wedge \ \texttt{c} = \texttt{xy} \ \wedge \ \texttt{a} = \texttt{x} * \texttt{x}$$
$$\longrightarrow$$
$$\texttt{a} = \texttt{x}^2 \ \wedge \ \texttt{b} = \texttt{y}^2 \ \wedge \ \texttt{c} = \texttt{xy}$$

- **line 11:** The mid-condition M_3 and code line 11 must establish the mid-condition M_4.

$$\texttt{a} = \texttt{x}^2 \ \wedge \ \texttt{b} = \texttt{y}^2 \ \wedge \ \texttt{c}_{old} = \texttt{xy} \ \wedge \ \texttt{c} = \texttt{c}_{old} + \texttt{c}_{old}$$
$$\longrightarrow$$
$$\texttt{a} = \texttt{x}^2 \ \wedge \ \texttt{b} = \texttt{y}^2 \ \wedge \ \texttt{c} = 2\texttt{xy}$$

- **line 13:** The mid-condition M_4 and code line 13 must establish the post-condition Q.

$$\texttt{a} = \texttt{x}^2 \ \wedge \ \texttt{b} = \texttt{y}^2 \ \wedge \ \texttt{c} = 2\texttt{xy} \ \wedge \ \texttt{r} = \texttt{a} + \texttt{b} + \texttt{c}$$
$$\longrightarrow$$
$$\texttt{r} = (\texttt{x} + \texttt{y})^2$$

Specification of Kotlin Programs - Conclusions

- A piece of code may have more than one pre-condition/post-condition.
- The post-condition depends on the code as well as the pre-condition.
- We use mid-conditions as “stepping-stones” in our reasoning.

Food for thought:

- In general, given some code and a post-condition, there exists a *Weakest* pre-condition.
- In general, given some code and a pre-condition, there exists a *Strongest* post-condition.