Function Calls

An Example Program with a Function Call

```
1   fun sort(b: Array<Int>)
2   // PRE: true                                                          (P1)
3   // POST: b[..) ~ b[..)_pre  ∧  Sorted( b[..) )                        (Q1)
4   {
5       ...
6   }
7
8   fun smallest(a: Array<Int>): Int
9   // PRE: a.size > 0                                                    (P2)
10  // POST: r = min( a[..)_pre )                                         (Q2)
11  {
12      // MID: a[..) ≈ a[..)_pre  ∧  a.size > 0                          (M0)
13      sort(a)
14      // MID: a[..) ~ a[..)_pre  ∧  Sorted( a[..) )  ∧  a.size > 0      (M1)
15      val res = a[0]
16      // MID: a[..) ~ a[..)_pre  ∧  res = min( a[..) )                  (M2)
17      return res
18  }
```

However, this program can potentially update the contents of the array a, so we do have to take some care to track the values of these contents throughout. We add an extra mid-condition $M0$ to capture the initial (unmodified) state of the input array, along with the other information from the pre-condition P_2

Up until the point where we call `sort(a)` we can see that the array's contents are unmodified, but after this function call we only know that the array's contents are a permutation of its original contents.

Reasoning about Function Calls

How do we show that a function call satisfies its caller's specification?

```
1    // MID: P
2    someFunc(v_1, ..., v_n)
3    // MID: Q
```

We have to show that P satisfies the function's pre-condition and that, in turn, the function's post-condition satisfies Q.

$$\frac{P \longrightarrow R[\overline{x} \mapsto \overline{v}] \qquad P[\overline{v}[..) \mapsto \overline{v}[..)_{old}] \wedge S[\overline{x} \mapsto \overline{v}][\overline{v}[..)_{pre} \mapsto \overline{v}[..)_{old}] \longrightarrow Q}{\{\ P\ \}\ \ \texttt{someFunc}(v_1, ..., v_n)\ \ \{\ Q\ \}}$$

where

```
fun someFunc(x_1: type, ..., x_n: type)
// PRE: R
// POST: S
```

Recall also that we write $P[x \to y]$ to represent the predicate P with all free occurrences of x replaced with y. The substitutions of $[x \to v]$ in R and S ensure that we replace the abstract variables in the function's specification with the concrete values used at its call point in our code.

The additional substitutions of $[v[..) \to v[..)_{old}]$ in P and $[v[..)_{pre} \to v[..)_{old}]$ in S account for the potential update in program state caused by the function call (side effects).

We have to take particular care with the function post-condition S to remember that any reference to an array's contents $a[..)_{pre}$ refers to the contents of that array before the function is called.

Kotlin is call by value, so no variable of any primitive type (int, bool, char, etc.) can be updated by the function call. However, when we pass variables of reference type (in this module just arrays, but in general any object reference) the contents of these references can be updated. i.e. when we pass an array to a function, we could modify the contents of that array, but not the array reference itself.

Important: the substitutions $[v[..)_{pre} \to v[..)_{old}]$ in S takes place after the call value substitutions $[x \to v]$ to ensure that there is no ambiguity in the effect of these two sets of substitutions.

To reason about a function call:
we must show that pre-condition $R[\overline{\mathtt{x}} \mapsto \overline{v}]$ is established before the call
we may assume that post-condition $S[\overline{\mathtt{x}} \mapsto \overline{v}]$ will hold after the call

```
code1
// MID: P
someFunc(v_1, ..., v_n)
// MID: Q
code2
```

where

```
fun someFunc(x_1: type, ..., x_n: type)
// PRE: R
// POST: S
```

- It is the "responsibility" of `code1` to establish $R[\overline{\mathtt{x}} \mapsto \overline{v}]$ through P.
- In return, `code2` may assume $S[\overline{\mathtt{x}} \mapsto \overline{v}]$ "for free" as part of Q.

In practice, we will usually need to hold on to some additional information about the state that is not affected by the function call. This will be captured by P and Q in the above proof outline.

Function Call - Example

```
fun sort(b: Array<Int>)
// PRE: true                                                    (P_1)
// POST: b[..] ~ b[..]_pre  ∧  Sorted(b[..])                    (Q_1)
{
    ...
}

fun smallest(a: Array<Int>): Int
// PRE: a.size > 0                                              (P_2)
// POST: r = min(a[..]_pre)                                     (Q_2)
{
    // MID: a[..] ≈ a[..]_pre  ∧  a.size > 0                    (M_0)
    sort(a)
    // MID: a[..] ~ a[..]_pre  ∧  Sorted(a[..])  ∧  a.size > 0  (M_1)
    val res = a[0]
    // MID: a[..] ~ a[..]_pre  ∧  res = min(a[..])              (M_2)
    return res
}
```

In the code above, we find the smallest element of the input array a by sorting the array and then returning the first element. This is allowed by the specification of smallest, as this does not require the input array to be unmodified, just that we return the smallest element of the input array.

Reasoning about `smallest` - Proof Obs. (Informal)

The correctness of the `smallest` function relies on the following argument:

line 12: The pre-condition of `smallest` must establish the mid-condition M_1.
$P_2[\,\mathtt{a}[..) \mapsto \mathtt{a}[..)_{pre}\,] \;\wedge\; \mathtt{a}[..) \approx \mathtt{a}[..)_{pre} \;\longrightarrow\; M_1$

line 13: The mid-condition M_1 must establish the pre-condition of `sort`.
$M_1 \;\longrightarrow\; P_1[\,\mathtt{b} \mapsto \mathtt{a}\,]$

line 14: The post-condition of `sort` must establish the mid-condition M_2.
$M_1[\,\mathtt{a}[..) \mapsto \mathtt{a}[..)_{old}\,] \;\wedge\; Q_1[\,\mathtt{b} \mapsto \mathtt{a}\,][\,\mathtt{a}[..)_{pre} \mapsto \mathtt{a}[..)_{old}\,] \;\longrightarrow\; M_2$

line 15: The mid-condition M_2 and code line 15 must establish the mid-condition M_3.
$M_2 \;\wedge\; \mathtt{val\ res\ =\ a[0]} \;\longrightarrow\; M_3$

line 17: The mid-condition M_3 and code line 17 must establish the post-condition of `smallest`.
$M_3 \;\wedge\; \mathtt{return\ res} \;\longrightarrow\; Q_2$

Important: care must be taken to track the **effects** of the code.

The exact number of obligations which we will have to prove is determined by the number of mid-conditions in our code and the number of function calls we make. Each mid-condition will require a corresponding proof obligation to establish that it holds. Each function call will require an additional proof obligation to establish that the function's pre-condition holds.

Reasoning about `smallest` - Care with Variables

Remember, a naive translation of the arguments can lead to incorrect proof obligations.

We must carefully track which variable values we are referring to. Are they **current** values, **old** values or **initial** values with respect to the code?

We achieve this by substituting the variables in function specifications (pre- and post-conditions) by the arguments that were *passed* to them and by tracking the state modified by the code

e.g. on line 14, when we use the post-condition of `sort` to establish M_2, we are given:

$$Q_1[\,\mathtt{b} \mapsto \mathtt{a}\,][\,\mathtt{a}[..)_{pre} \mapsto \mathtt{a}[..)_{old}\,] \iff \mathtt{a}[..) \sim \mathtt{a}[..)_{old} \wedge \mathit{Sorted}(\,\mathtt{a}[..)\,)$$

Notice how our substitution above converts the abstract variable b from the function definition of sort into the concrete reference value a that was passed into it.

We also carefully track the state of the contents of array a, as we know that these will be modified by the sort function. Any mention of the initial (pre) state of the array contents at the point that sort was called are actually referring to the old state of the array contents before the function call was executed. That is, the callees initial state is the same as the callers old state. We often need this information for our proofs.

Reasoning about `smallest` - Proof Obs. (Final)

line 12: The pre-condition of `smallest` must establish the mid-condition M_1

$$P_2[\,\mathtt{a}[..) \mapsto \mathtt{a}[..)_{pre}\,] \wedge \mathtt{a}[..) \approx \mathtt{a}[..)_{pre} \longrightarrow M_1$$

$$\mathtt{a.size} > 0 \wedge \mathtt{a}[..) \approx \mathtt{a}[..)_{pre} \longrightarrow \mathtt{a}[..) \approx \mathtt{a}[..)_{pre} \wedge \mathtt{a.size} > 0$$

In this first obligation we have to remember that the pre-condition P_2 only describes the variables that were passed into the function call. In order to use the pre-condition in our proof, we must substitute all of the variables for their initial $_{pre}$ versions.

Moreover, as no code has been executed so far in the function, we gain the implicit information that $a[..) \approx a[..)_{pre}$ in our premise.

Reasoning about `smallest` - Proof Obs. (Final)

line 13: The mid-condition M_1 must establish the pre-condition of `sort`.

$$M_1 \longrightarrow P_1[\texttt{b} \mapsto \texttt{a}]$$

$$\texttt{a}[..) \approx \texttt{a}[..)_{pre} \ \wedge \ \texttt{a.size} > 0 \longrightarrow \texttt{true}$$

In this proof obligation, the $[b \rightarrow a]$ substitution for P_1 on line 13 is handling the fact that the sort function is defined in terms of the integer array parameter b, but it has been called with the integer array reference a.

We always have to perform this substitution of the abstract variables from the function's specification to the concrete values used in the function call in the code we are reasoning about. In this obligation no code is actually being executed, so we do not need to worry about the current (x) or old (x_{old}) states of the variables.

The precondition P_1 is actually trivial to satisfy, so this proof obligation is not challenging to prove.

Reasoning about `smallest` - Proof Obs. (Final)

line 14: The post-condition of `sort` must establish the mid-condition M_2.

$$M_1[\,\mathtt{a}[..) \mapsto \mathtt{a}[..)_{old}\,] \;\wedge\; Q_1[\,\mathtt{b} \mapsto \mathtt{a}\,][\,\mathtt{a}[..)_{pre} \mapsto \mathtt{a}[..)_{old}\,] \;\longrightarrow\; M_2$$

$$\begin{array}{c} \mathtt{a}[..)_{old} \approx \mathtt{a}[..)_{pre} \;\wedge\; \mathtt{a.size} > 0 \\ \wedge\; \mathtt{a}[..) \sim \mathtt{a}[..)_{old} \;\wedge\; \mathit{Sorted}(\,\mathtt{a}[..)\,) \\ \longrightarrow \\ \mathtt{a}[..) \sim \mathtt{a}[..)_{pre} \;\wedge\; \mathit{Sorted}(\,\mathtt{a}[..)\,) \;\wedge\; \mathtt{a.size} > 0 \end{array}$$

This proof relies on the following property:

$$\mathtt{a}[..) \sim \mathtt{b}[..) \;\wedge\; \mathtt{b}[..) \approx \mathtt{c}[..) \;\longrightarrow\; \mathtt{a}[..) \sim \mathtt{c}[..)$$

In this obligation we are considering the effect of the call to the sort function. This means that we must track the state of the input array both before and after the function call. The same "call value for specification variable" substitution as in the previous proof obligation from line 13 ($[b \to a]$) is happening for Q_1 in this proof obligation for line 14. However, in this case we also have the extra substitution $[a[..)_{pre} \to a[..)_{old}]$, which is handling the fact that we are passing an array reference into the sort function.

The Kotlin function call semantics ("call by value") ensures that the call to sort cannot modify the variables for any primitive type (int, bool, char, etc.). Similarly, sort cannot change the value of a reference variable (e.g. a in our example), but the contents of such a reference (e.g. the array elements) could be updated (indeed, they are rearranged in this example).

This means that some care is needed to track what is going on. The $[b \to a][a[..)_{pre} \to a[..)_{old}]$ substitutions are helping us with this. Together, they say that any reference to the initial/provided array contents ($b[..)_{pre}$) in the post-condition of sort is actually referring to the contents of the array a at the point the function call was made (i.e. $a[..)_{old}$). This allows us to carry though information about the array's contents from the mid-condition M_1 (note the similar $[a[..) \to a[..)_{old}]$ substitution occurring there), as we still have access to knowledge about the state of the program was before making the function call.

line 15: The mid-condition M_2 and code line 15 must establish the mid-condition M_3.
$M_2 \wedge$ `val res = a[0]` $\longrightarrow M_3$

$$\mathtt{a}[..) \sim \mathtt{a}[..)_{pre} \wedge Sorted(\,\mathtt{a}[..)\,) \wedge \mathtt{a.size} > 0 \wedge \mathtt{res} = \mathtt{a}[0]$$
$$\longrightarrow$$
$$\mathtt{a}[..) \sim \mathtt{a}[..)_{pre} \wedge \mathtt{res} = min(\,\mathtt{a}[..)\,)$$

This obligation concerns the execution of the code `val res = a[0]` which is a read-only local variable declaration and assignment instruction. This means that we need to track the current and old values of any variables modified by the code. However, the read-only local variable res does not actually exist before this code is executed, so there are no variables modified by this code, and thus no substitutions are required

line 17: The mid-condition M_3 and code line 17 must establish the post-condition of `smallest`.
$M_3 \wedge$ `return res` $\longrightarrow Q_2$

$$\mathtt{a}[..) \sim \mathtt{a}[..)_{pre} \wedge \mathtt{res} = min(\,\mathtt{a}[..)\,) \wedge \mathbf{r} = \mathtt{res}$$
$$\longrightarrow$$
$$\mathbf{r} = min(\,\mathtt{a}[..)_{pre}\,)$$

This last obligation simply tracks the assignment of the return value for the function. As this assignment has no other side effects, we do not need to track the old values of any of our program state. However, we do need to be sure that the function's post-condition is satisfied w.r.t. the original input parameter values (remember that Kotlin is call-by-value). Thankfully, the function body does not shadow the input array reference parameter a. So, we can dispense with the need to distinguish between a and a_{pre} throughout the function's body, as they will

always be the same. This means that no substitution is required on the post-condition Q_2.

What about `sort`?

We have been able to reason about the correctness of `smallest` without knowing the implementation details of `sort`

This is known as modular verification

We have assumed that `sort` satisifes its specification
but could there still be a problem...?

Modular verification is a very helpful property of our program reasoning, as this allows us to break down complex systems into smaller, more manageable chunks.

We can then reason about these chunks independently (with each chunk assuming that the others satisfy their specifications) and once all chunks have been verified we can then establish the correctness of the whole system.

Can anyone propose a problematic implementation of the `sort` function that still satisfies its specification?

```
1   fun sort(b: Array<Int>)
2   // PRE: true                                                    (P1)
3   // POST: b[..) ~ b[..)pre  ∧  Sorted( b[..) )                  (Q1)
4   {
5
6
7       sort(b)
8
9
10  }
```

Recall that our intuitive interpretation of the Hoare Triple

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

is of the form:

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

The problem here is that we are only verifying the behaviour of our code **after** it has executed. If our code never terminates, then we have nothing to prove.

This is known as the "**Partial Correctness Interpretation of Hoare Triples**".

If we care also about guaranteeing termination, then we would be talking instead about **Total Correctness**.

Total vs. Partial Correctness

Partial Correctness: if the code is executed in a state satisfying its precondition, then *if* it terminates it must reach a state that satisfies its postcondition.

Total Correctness: as partial correctness above, but we also know that the code will terminate.

Why distinguish between partial correctness and total correctness?

Some reasons include:

- Different Proof Obligations
- Desired Property Types
- Non-Terminating Code

There are several reasons why we distinguish between partial and total correctness. Firstly, depending on the code in question, it may be quite hard to prove one of partial correctness or termination, even though the other might be easy. In fact, in some cases one of these proofs might actually be impossible (e.g. The Halting Problem). Secondly, there might be some properties of your code that are more important to you than termination. For example, you might not know if your code will terminate, but still want to be able to prove that if it does terminate then it will have solved your problem. Thirdly, there are some pieces of code that are legitimately non-terminating (web-servers, operating systems, etc...) but we

may still want or need to verify their correctness. There are probably also many other good reasons for the distinction.

Reasoning about Function Calls

What if the function has a return value?

```
1    // MID: P
2    res = someFunc(v_1, ..., v_n)
3    // MID: Q
```

Just as with assignment, we must carefully track the modification of the variable `res`. This requires some additional substitutions in our assertions:

$$\frac{\begin{array}{c} P \longrightarrow R[\overline{\mathtt{x}} \mapsto \overline{v}] \\ \left(\begin{array}{c} P[\overline{v}[..) \mapsto \overline{v}[..)_{old}][\mathtt{res} \mapsto \mathtt{res}_{old}] \ \wedge \ \mathtt{res} = \mathtt{r} \\ \wedge \ S[\overline{\mathtt{x}} \mapsto \overline{v}][\overline{v}[..)_{pre} \mapsto \overline{v}[..)_{old}][\mathtt{res} \mapsto \mathtt{res}_{old}] \end{array} \right) \longrightarrow Q \end{array}}{\{\ P\ \}\ \ \mathtt{res} = \mathtt{someFunc}(v_1, ..., v_n)\ \ \{\ Q\ \}}$$

where

```
someFunc(x_1: type, ..., x_n: type): type
// PRE: R
// POST: S
```

The additional substitutions of $[res \rightarrow res_{old}]$ in S and P and the extra conjunct $res = r$ account for the update in program state caused by the assignment of the function's return value. The rest of the rule is the same as before.

Important: the substitution $[res \rightarrow res_{old}]$ must take place after all other substitutions to properly reflect the order of execution of the code. That is, the function is run in a state where the variable res still has its old value.

Reasoning about Function Calls with Returns

To reason about a function call with a return value:
we must show that pre-condition $R[\overline{\mathtt{x}} \mapsto \overline{v}]$ is established before the call
we may assume that post-condition $S[\overline{\mathtt{x}} \mapsto \overline{v}]$ will hold after the call
and also that $\mathtt{res} = \mathbf{r}$ after the call

```
code1
// MID: P
res = someFunc(v_1, ..., v_n)
// MID: Q
code2
```

where

```
someFunc(x_1: type, ..., x_n: type): type
// PRE: R
// POST: S
```

- It is the "responsibility" of `code1` to establish $R[\overline{\mathtt{x}} \mapsto \overline{v}]$ through P.
- In return, `code2` may assume $S[\overline{\mathtt{x}} \mapsto \overline{v}] \ \wedge \ \mathtt{res} = \mathbf{r}$ "for free" in Q.

The requirements on the pre-condition are the same as in the case where there is not return value, but we have an additional update to the program state to be aware of after the call is complete.

Function Call with Return - Example

```
fun increment(i: Int): Int
// PRE: true                (P_inc)
// POST: r = i + 1          (Q_inc)
{ ... }

fun decrement(i: Int): Int
// PRE: true                (P_dec)
// POST: r = i - 1          (Q_dec)
{ ... }

fun sillyUpDown(x: Int): Int
// PRE: true                                  (P)
// POST: r = x                                (Q)
{
    // MID: x = x_pre                        (M_0)
    var x = increment(x)
    // MID: x = x_pre + 1                    (M_1)
    x = decrement(x)
    // MID: x = x_pre                        (M_2)
    return x
}
```

We assume that both `increment` and `decrement` are **totally correct** w.r.t their respective function specifications

Note that the function body above does shadow the input parameter x, so we will need to be careful to track the value of this variable throughout our reasoning. In particular, we add an extra mid-condition M_0 to capture the initial (unmodified) value of the input parameter.

This is probably the only time that we will actually do this with a function definition (as it really isn't a good idea in general), but it is good to see that our reasoning system can handle this situation. We could also have solved the problem by renaming the local variable x used in within `sillyUpDown`, but we prefer not to modify the programs that we are trying to prove if possible.

Reasoning about `sillyUpDown` - Proof Obs. (Informal)

The correctness of `sillyUpDown` relies on the following argument:

line 10: The pre-condition of `sillyUpDown` must establish mid-condition M_0.
$P[\mathtt{x} \mapsto \mathtt{x}_{pre}] \wedge \mathtt{x} = \mathtt{x}_{pre} \longrightarrow M_0$

line 11: The mid-condition M_0 must establish the pre-condition of `increment`
$M_0 \longrightarrow P_{inc}[\mathtt{i} \mapsto \mathtt{x}]$

line 12: The post-condition of `increment` must establish mid-condition M_1.
$M_0[\mathtt{x} \mapsto \mathtt{x}_{old}] \wedge Q_{inc}[\mathtt{i} \mapsto \mathtt{x}_{old}] \wedge \mathtt{var\ x = increment(x)} \longrightarrow M_1$

line 13: The mid-condition M_1 must establish the pre-condition of `decrement`
$M_1 \longrightarrow P_{dec}[\mathtt{i} \mapsto \mathtt{x}]$

line 14: The post-condition of `decrement` must establish mid-condition M_2.
$M_1[\mathtt{x} \mapsto \mathtt{x}_{old}] \wedge Q_{dec}[\mathtt{i} \mapsto \mathtt{x}_{old}] \wedge \mathtt{x = decrement(x)} \longrightarrow M_2$

line 15: The mid-condition M_2 and code line 15 must establish the post-condition of `sillyUpDown`.
$M_2 \wedge \mathtt{return\ x} \longrightarrow Q[\mathtt{x} \mapsto \mathtt{x}_{pre}]$

We discuss the development of each of these proof obligations over the following slides. The function calls to increment and decrement incur extra proof obligations that checks that their pre-conditions are satisfied before they are called (as usual), but we will see that these are relatively trivial in this instance.

Reasoning about `sillyUpDown` - Proof Obs.

line 10: The pre-condition of `sillyUpDown` must establish mid-condition M_0.

$$P[\mathtt{x} \mapsto \mathtt{x}_{pre}] \ \wedge \ \mathtt{x} = \mathtt{x}_{pre} \ \longrightarrow \ M_0$$

$$\mathtt{true} \ \wedge \ \mathtt{x} = \mathtt{x}_{pre}$$
$$\longrightarrow$$
$$\mathtt{x} = \mathtt{x}_{pre}$$

In this first obligation we have to remember that the pre-condition P only describes the variables that were passed into the function call. In order to use the pre-condition in our proof, we must substitute all of the variables for their initial $_{pre}$ versions. Of course, in this case the pre-condition doesn't actually refer to x, so the substitution has no effect.

Moreover, as no code has been executed so far in the function, we gain the implicit information that $x = x_{pre}$. We also do not need to worry about the current (x) or old (x_{old}) states of the variables.

line 11: The mid-condition M_0 must establish the pre-condition of `increment`

$$M_0 \ \longrightarrow \ P_{inc}[\mathtt{i} \mapsto \mathtt{x}]$$

$$\mathtt{x} = \mathtt{x}_{pre}$$
$$\longrightarrow$$
$$\mathtt{true}$$

In this obligation no code is actually being executed, so we do not need to worry about the current (x) or old (x_{old}) states of the variables. However, we do need to substitute the call value of x into the pre-condition P_{inc} of the increment function. Again, this pre-condition doesn't actually refer to x, so the substitution also has no effect.

line 12: The post-condition of `increment` must establish mid-condition M_1.

$M_0[\mathtt{x} \mapsto \mathtt{x}_{old}] \wedge Q_{inc}[\mathtt{i} \mapsto \mathtt{x}_{old}] \wedge \texttt{var x = increment(x)} \longrightarrow M_1$

$$\mathtt{x}_{old} = \mathtt{x}_{pre} \ \wedge \ \mathbf{r} = \mathtt{x}_{old} + 1 \ \wedge \ \mathtt{x} = \mathbf{r} \longrightarrow \mathtt{x} = \mathtt{x}_{pre} + 1$$

In this obligation we are considering the effect of the call to the increment function. This means that we must track the state of the program both before and after the function call. Thus, in the post-condition Q_{inc} we replace the function's argument variable i with the call value x.

Note that as well as knowing the post-condition of the increment function, we also still know what the state of the program was before the function call, which in this case was the mid-condition M_0. However, we must update any reference to the variable x from before the function call to x_{old} to reflect that x is assigned the return value of the increment function call by the code.

line 13: The mid-condition M_1 must establish the pre-condition of `decrement`

$M_1 \longrightarrow P_{dec}[\mathtt{i} \mapsto \mathtt{x}]$

$$\mathtt{x} = \mathtt{x}_{pre} + 1 \longrightarrow \mathtt{true}$$

In this obligation no code is actually being executed, so we do not need to worry about the current (x) or old (x_{old}) states of the variables.

However, we do need to substitute the call value of x into the pre-condition P_{dec} of the decrement function. As before, this pre-condition doesn't actually refer to x, so the substitution also has no effect.

line 14: The post-condition of `decrement` must establish mid-condition M_2.
$M_1[\mathtt{x} \mapsto \mathtt{x}_{old}] \wedge Q_{dec}[\mathtt{i} \mapsto \mathtt{x}_{old}] \wedge \mathtt{x} = \mathtt{decrement(x)} \longrightarrow M_2$

$$\mathbf{x}_{old} = \mathbf{x}_{pre} + 1 \wedge \mathbf{r} = \mathbf{x}_{old} - 1 \wedge \mathbf{x} = \mathbf{r}$$
$$\longrightarrow$$
$$\mathbf{x} = \mathbf{x}_{pre}$$

In this obligation we are considering the effect of the call to the increment function. This means that we must track the state of the program both before and after the function call. Thus, in the post-condition Q_{dec} we replace the function's argument variable i with the call value x.

Note that as well as knowing the post-condition of the decrement function, we also still know what the state of the program was before the function call (i.e. M_1). However, we must update any reference to the variable x from before the function call to x_{old} to reflect that x is assigned the return value of the increment function call by the code.

line 15: The mid-condition M_2 and code line 15 must establish the post-condition of `sillyUpDown`.
$M_2 \wedge \mathtt{return\ x} \longrightarrow Q[\mathtt{x} \mapsto \mathtt{x}_{pre}]$

$$\mathbf{x} = \mathbf{x}_{pre} \wedge \mathbf{r} = \mathbf{x}$$
$$\longrightarrow$$
$$\mathbf{r} = \mathbf{x}_{pre}$$

This last obligation simply tracks the assignment of the return value for the overall `sillyUpDown` function. As this assignment has no other side effects, we do not need to track the old values of any of our program state. However, we do need to be sure that the function's post-condition Q is satisfied w.r.t. the original value for the integer input parameter x (i.e. x_{pre}). Hence the substitution of $[x \rightarrow x_{pre}]$ on Q.

In practice, we avoid such potential confusion by not shadowing the input parameters of our functions. This is the approach that we will normally take from this point forwards.