Recursion

Recursion - Example

```
1   fun sum(a: Array<Int>): Int
2   // PRE: true                                                    (P1)
3   // POST: a[..) ≈ a[..)pre  ∧  r = ∑ a[..)                       (Q1)
4   {
5       val res = sumAux(a,0)
6       // MID: a[..) ≈ a[..)pre  ∧  res = ∑ a[..)                  (M1)
7       return res
8   }
```

where sumAux satisfies the following specification:

```
fun sumAux(a: Array<Int>, i: Int): Int
// PRE: 0 ≤ i ≤ a.size                                             (P2)
// POST: a[..) ≈ a[..)pre  ∧  r = ∑ a[i..)                         (Q2)
```

Note that we can derive the mid-condition M_1 without any knowledge of the specification of `sumAux` by working backwards from the post-condition Q_1 of `sum`. The only way that the return statement can satisfy the post-condition is if `res` has been set up to be equal to $\sum$ a[..).

Reasoning about `sum`

The correctness of `sum` and `sumAux` can be shown independently. This is another example of modular verification.

Reasoning about `sum` - Proof Obligations (Informal)

The correctness of `sum` relies on the following arguments:

line 5: The pre-condition of `sum` must establish the pre-condition of `sumAux`.
$P_1 \longrightarrow P_2[\texttt{i} \mapsto 0]$

line 6: The post-condition of `sumAux` must establish the mid-condition M_1.
$P_1 \wedge Q_2[\texttt{i} \mapsto 0] \wedge \texttt{val res = sumAux(a,0)} \longrightarrow M_1$

line 7: The mid-condition M_1 must establish the post-condition of `sum`.
$M_1 \wedge \texttt{return res} \longrightarrow Q_1$

Important: in the above we use `sumAux` as a "black box".

Reasoning about `sum` - Proof Obligations (Final)

line 5: The pre-condition of `sum` must establish the pre-condition of `sumAux`.
$P_1 \longrightarrow P_2[\texttt{i} \mapsto 0]$

$$\texttt{true} \longrightarrow 0 \leq 0 \leq \texttt{a.size}$$

There is no reference to the array contents in P_1 or P_2, nor has any code run before line 5, so all we need to substitute here are the call values for `sumAux` on line 5. The proof would rely on the fact that a is of type `Array<Int>` and thus cannot be null. Thus, `a.size` is defined and also ≥ 0.

line 6: The post-condition of `sumAux` must establish the mid-condition M_1.

$$P_1 \ \wedge \ Q_2[\mathtt{i} \mapsto 0] \ \wedge \ \mathtt{val\ res\ =\ sumAux(a,0)} \ \longrightarrow \ M_1$$

$$\mathtt{true} \ \wedge \ \mathtt{a[..)} \approx \mathtt{a[..)}_{pre} \ \wedge \ \mathbf{r} = \sum \mathtt{a[0..)} \ \wedge \ \mathtt{res} = \mathbf{r}$$
$$\longrightarrow$$
$$\mathtt{a[..)} \approx \mathtt{a[..)}_{pre} \ \wedge \ \mathtt{res} = \sum \mathtt{a[..)}$$

In general, when we make a function call that could potentially make updates to an array, we need to carefully distinguish the contents of the array passed into the function ($\mathrm{a[..)_{old}}$) from the initial array contents ($\mathrm{a[..)_{pre}}$) and the current array contents ($\mathrm{a[..)}$). However, in this case no code has actually run before line 5, so we know that the array contents are not modified before the call to `sumAux`. Thus, we know that $\mathrm{a[..)_{pre}}$ passed into `sumAux` will be the same as $\mathrm{a[..)_{pre}}$ passed into `sum`, and we choose not to make any substitutions for this proof obligation. Another way to look at this is to observe that in tracking this explicitly we would end up writing:

$$\mathrm{Q_2[i \rightarrow 0][a[..)_{pre} \rightarrow a[..)_{old}] \cap a[..)_{old} \approx \ a[..)_{pre}}$$

which is somewhat redundant, as the the second assertion could be folded up into the substitution to give us:

$$\mathrm{Q_2[i \rightarrow 0][a[..)_{pre} \rightarrow a[..)_{pre}]}$$

the second part of which is an identity substitution, and not worth applying (as it makes no difference to the overall assertion).

line 7: The mid-condition M_1 must establish the post-condition of `sum`.

$$M_1 \ \wedge \ \mathtt{return\ res} \ \longrightarrow \ Q_1$$

$$\mathtt{a[..)} \approx \mathtt{a[..)}_{pre} \ \wedge \ \mathtt{res} = \sum \mathtt{a[..)} \ \wedge \ \mathbf{r} = \mathtt{res}$$
$$\longrightarrow$$
$$\mathtt{a[..)} \approx \mathtt{a[..)}_{pre} \ \wedge \ \mathbf{r} = \sum \mathtt{a[..)}$$

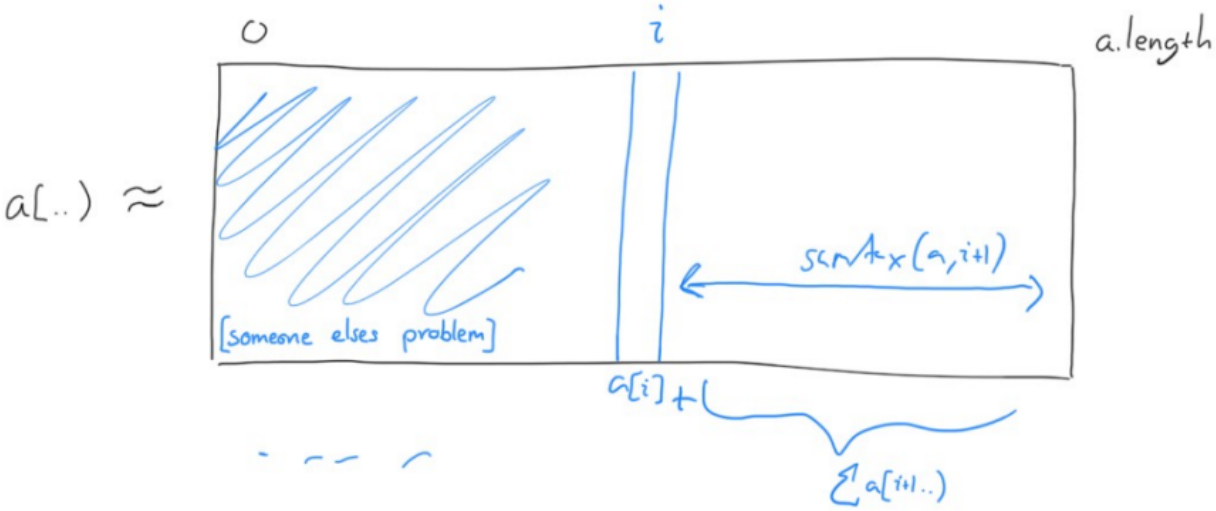
The final proof obligation for `sum` is for a simple return statement.

Recursion - Example

```
1   fun sumAux(a: Array<Int>, i: Int): Int
2   // PRE: 0 ≤ i ≤ a.size                                          (P2)
3   // POST: a[..) ≈ a[..)pre  ∧  r = ∑a[i..)                      (Q2)
4   {
5       if (i == a.size) {
6           // MID: a[..) ≈ a[..)pre  ∧  i = a.size                (M2)
7           return 0
8       } else {
9           // MID: a[..) ≈ a[..)pre  ∧  0 ≤ i < a.size            (M3)
10          val res = a[i] + sumAux(a,i+1)
11          // MID: a[..) ≈ a[..)pre  ∧  res = ∑a[i..)             (M4)
12          return res
13      }
14  }
```

We will need to carefully track the contents of the array a.

The Recursion Graphically



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

The correctness of `sumAux` relies on the following arguments:

line 6: The pre-condition of `sumAux` must establish the mid-condition M_2.
$P_2 \ \wedge\ \texttt{a[..)} \approx \texttt{a[..)}_{pre} \ \wedge\ \texttt{(i == a.size)} \ \longrightarrow\ M_2$

line 7: The mid-condition M_2 must establish the post-condition of `sumAux`.
$M_2 \ \wedge\ \texttt{return 0} \ \longrightarrow\ Q_2$

line 9: The pre-condition of `sumAux` must establish the mid-condition M_3.
$P_2 \ \wedge\ \texttt{a[..)} \approx \texttt{a[..)}_{pre} \ \wedge\ \neg\texttt{(i == a.size)} \ \longrightarrow\ M_3$

line 10: The mid-condition M_3 must establish the pre-condition of `sumAux`.
$M_3 \ \longrightarrow\ P_2[\texttt{i} \mapsto \texttt{i+1}]$

line 11: The post-condition of `sumAux` must establish the mid-condition M_4.
$M_3[\texttt{a[..)} \mapsto \texttt{a[..)}_{old}] \ \wedge\ Q_2[\texttt{i} \mapsto \texttt{i+1}][\texttt{a[..)}_{pre} \mapsto \texttt{a[..)}_{old}]$
$\wedge\ \texttt{val res = a[i] + sumAux(a,i+1)} \ \longrightarrow\ M_4$

line 12: The mid-condition M_4 must establish the post-condition of `sumAux`.
$M_4 \ \wedge\ \texttt{return res} \ \longrightarrow\ Q_2$

Important: obligations for lines 10 and 11 reference the **recursive** `sumAux` call.

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

line 6: The pre-condition of `sumAux` must establish the mid-condition M_2.
$P_2 \ \wedge\ \texttt{a[..)} \approx \texttt{a[..)}_{pre} \ \wedge\ \texttt{(i == a.size)} \ \longrightarrow\ M_2$

$$0 \leq \texttt{i} \leq \texttt{a.size} \ \wedge\ \texttt{a[..)} \approx \texttt{a[..)}_{pre} \ \wedge\ \texttt{i} = \texttt{a.size}$$
$$\longrightarrow$$
$$\texttt{a[..)} \approx \texttt{a[..)}_{pre} \ \wedge\ \texttt{i} = \texttt{a.size}$$

In order to arrive at line 6 of the function, the code must have entered the then branch of the if statement, so we know that `i = a.size`. We can also observe that the code has not modified the array or its contents on this path, which gives us the implicit knowledge that $\texttt{a[..)} \approx \texttt{a[..)}_{\text{pre}}$ in the premise of this obligation.

line 7: The mid-condition M_2 must establish the post-condition of `sumAux`.
$M_2 \ \wedge$ `return 0` $\longrightarrow \ Q_2$

$$\mathtt{a}[..) \approx \mathtt{a}[..)_{pre} \ \wedge \ \mathtt{i} = \mathtt{a.size} \ \wedge \ \mathbf{r} = 0$$
$$\longrightarrow$$
$$\mathtt{a}[..) \approx \mathtt{a}[..)_{pre} \ \wedge \ \mathbf{r} = \sum \mathtt{a}[\mathtt{i}..)$$

This proof relies on the following property:

$$\forall k.[\ \textstyle\sum \mathtt{a}[\mathtt{k}..\mathtt{k}) = 0\ ]$$

This proof obligation is ensuring that the post-condition of the `sumAux` function holds in the base-case of the recursion. Proving that r has the desired return value relies on recalling from the definition of P that: $\forall \mathrm{k}.\ [\sum\ \mathrm{a}[\mathrm{k}..\mathrm{k}) = 0]$

That is, the sum of any empty range is always equal to 0 (the identity element of addition).

line 9: The pre-condition of `sumAux` must establish the mid-condition M_3.
$P_2 \ \wedge \ \mathtt{a}[..) \approx \mathtt{a}[..)_{pre} \ \wedge \ \neg(\mathtt{i} == \mathtt{a.size}) \ \longrightarrow \ M_3$

$$0 \leq \mathtt{i} \leq \mathtt{a.size} \ \wedge \ \mathtt{a}[..) \approx \mathtt{a}[..)_{pre} \ \wedge \ \mathtt{i} \neq \mathtt{a.size}$$
$$\longrightarrow$$
$$\mathtt{a}[..) \approx \mathtt{a}[..)_{pre} \ \wedge \ 0 \leq \mathtt{i} < \mathtt{a.size}$$

This proof obligation is set up in a similar way as that on line 6, but entering the else branch of the if statement instead. We still have the implicit knowledge that the code has not modified the input array or its contents at this point in the function.

line 10: The mid-condition M_3 must establish the pre-condition of `sumAux`.

$$M_3 \longrightarrow P_2[\texttt{i} \mapsto \texttt{i+1}]$$

$$\texttt{a}[..) \approx \texttt{a}[..)_{pre} \ \wedge \ 0 \leq \texttt{i} < \texttt{a.size} \longrightarrow 0 \leq \texttt{i+1} \leq \texttt{a.size}$$

In this proof obligation we are ensuring that the program state satisfies the pre-condition of the recursive call to `sumAux`. So, it is important that we substitute the function parameters with their call values on line 10. We could also explicitly denote the substitution on P_2 of $\mathrm{a} \rightarrow \mathrm{a}$. However, recall that we choose to omit unnecessary identity substitutions to ease the notational burden of our proof obligations.

line 11: The post-condition of `sumAux` must establish the mid-condition M_4.

$$M_3[\texttt{a}[..) \mapsto \texttt{a}[..)_{old}] \ \wedge \ Q_2[\texttt{i} \mapsto \texttt{i+1}][\texttt{a}[..)_{pre} \mapsto \texttt{a}[..)_{old}] \ \wedge \ \texttt{val res = a[i] + sumAux(a,i+1)} \longrightarrow M_4$$

$$\texttt{a}[..)_{old} \approx \texttt{a}[..)_{pre} \ \wedge \ 0 \leq \texttt{i} < \texttt{a.size} \ \wedge \ \texttt{a}[..) \approx \texttt{a}[..)_{old} \ \wedge \ \texttt{r} = \sum \texttt{a[i+1..)} \ \wedge \ \texttt{res} = \texttt{a[i]}_{old} + \texttt{r} \longrightarrow \texttt{a}[..) \approx \texttt{a}[..)_{pre} \ \wedge \ \texttt{res} = \sum \texttt{a[i..)}$$

In this proof obligation we are applying the effects of the recursive call to `sumAux` to our program state. It is important that the premise of our proof obligation includes the fact that $0 \leq \mathrm{i} < \mathrm{a.size}$ so that we know the array dereference of $\mathrm{a}[\mathrm{i}]$ in the code is valid (does not result in an `ArrayOutOfBounds` error).

Also note that if line 10 of our program were modified to:

```
val res = sumAux(a,i+1) + a[i]}
```

then the proof obligation above would have to be slightly modified to track the new order of execution. In particular, the effect of the code would now be described as:

$$\texttt{res} = \mathbf{r} \texttt{ + a[i]}$$

In this case, when we add the i^{th} element of the array to the sum, we do so from the array that was modified by the call to `sumAux`. Of course, our specification for `sumAux` actually guarantees that the array is unmodified, so in this case the ordering makes no difference. However, in general the ordering of such operations can be very important.

line 12: The mid-condition M_4 must establish the post-condition of `sumAux`.
$M_4 \wedge \texttt{return res} \longrightarrow Q_2$

$$\texttt{a}[..) \approx \texttt{a}[..)_{pre} \wedge \texttt{res} = \sum \texttt{a}[\texttt{i}..) \wedge \mathbf{r} = \texttt{res}$$
$$\longrightarrow$$
$$\texttt{a}[..) \approx \texttt{a}[..)_{pre} \wedge \mathbf{r} = \sum \texttt{a}[\texttt{i}..)$$

Finally, we check that the return from the recursive case of `sumAux` also satisfies the overall function's post-condition.

Is There Anything We Have Overlooked?

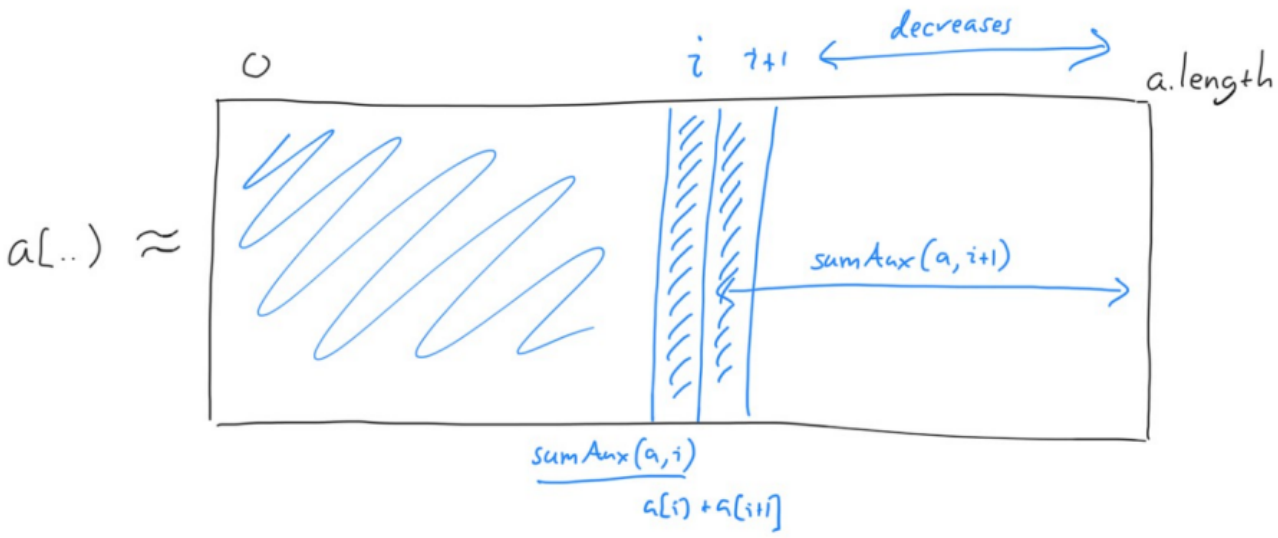
Consider the following Kotlin function `sillySum`:

```
1  fun sillySum(a: Array<Int>): Int
2  // PRE: a ≠ null                                   (P)
3  // POST: a[..) ≈ a[..)pre ∧ r = ∑a[..)             (Q)
4  {
5      return sillySum(a)
6  }
```

...so what about Termination?

Note that the `sillySum` function would satisfy its post-condition if it were to terminate, so the function is partially correct. However, termination of `sillySum` is clearly not guaranteed (or in this case even possible), so the function is definitely not totally correct.

Proving Termination - Measure of Progress



The length of the array that remains to be processed ($\mathrm{a.size} - \mathrm{i}$) decreases with each recursive call to `sumAux`. We can use this “Measure of Progress” to reason about the termination of the `sumAux` function.

Proving Termination of `sum`

To show that `sum` terminates, we need to prove that

$$\forall a \in \texttt{int[]}, i \in \mathbb{Z}.[\ 0 \leq i \leq a\texttt{.size} \longrightarrow \texttt{sumAux}(a,i)\ \text{terminates}\ ]$$

which we could tackle using mathematical induction considering the length of the remaining array to process i.e.

$$a\texttt{.size} - i$$

We prove termination of the `sumAux` function via mathematical induction on the length of the remaining array to be processed, i.e. $\mathrm{a.size} - \mathrm{i}$.

Base case:

To show:
$\forall a \in \texttt{int[]}, i \in \mathbb{Z}.$
$[\ 0 \leq i \leq a.\texttt{size} \longrightarrow \texttt{sumAux}(a,i)$ terminates when $a.\texttt{size} - i = 0\ ]$

Proof:
Take arbitrary $a \in \texttt{int[]}$ and $i \in \mathbb{Z}$ where $0 \leq i \leq a.\texttt{size}$.
$a.\texttt{size} - i = 0 \longrightarrow i = a.\texttt{size}$
When $i = a.\texttt{size}$, `SumAux` enters the first branch of the conditional and terminates on line 7.

Inductive Step:

Take arbitrary $k \in \mathbb{N}$.

Inductive Hypothesis:
(IH): $\forall a \in \texttt{int[]}, i' \in \mathbb{Z}.$
$[\ 0 \leq i' \leq a.\texttt{size} \longrightarrow \texttt{sumAux}(a,i')$ terminates when $a.\texttt{size} - i' = k\ ]$

To show:
$\forall a \in \texttt{int[]}, i \in \mathbb{Z}.$
$[\ 0 \leq i \leq a.\texttt{size} \longrightarrow \texttt{sumAux}(a,i)$ terminates when $a.\texttt{size} - i = k + 1\ ]$

Proof:
Take arbitrary $a \in \texttt{int[]}$, $i \in \mathbb{Z}$ where $0 \leq i \leq a.\texttt{size}$.
$a.\texttt{size} - i = k + 1 \longrightarrow i = a.\texttt{size} - k - 1$
$i \neq a.\texttt{size}$ (since $k \in \mathbb{N}$) so `sumAux` enters the second branch of the conditional.
$a.\texttt{size} - i = k + 1 \longrightarrow i + 1 = a.\texttt{size} - k$
and $i + 1 > 0$, by choice of i, so applying **(IH)** (with i' as $i + 1$) we know that `sumAux(a,i+1)` will terminate.
So, `SumAux` will then terminate on line 12.

Comparison with Induction

Recall the Haskell function sum and its tail-recursive version sum_tr from earlier in the module:

```
sum :: [Int] -> Int
sum [] = 0
sum i:is = i + sum is

sum_tr :: [Int] -> Int -> Int
sum_tr [] k = k
sum_tr (i:is) k = sum_tr is (i+k)
```

Using Structural Induction we were able to prove:

$$\forall \texttt{is:[Int]}.\ \texttt{sum is} = \texttt{sum\_tr is 0}$$

However, we have just proven a very similar property for our Kotlin function sumAux, namely that it satisfies the specification:

$$\{\ \texttt{true}\ \}\quad \texttt{val res = sumAux(a,0)}\quad \{\ \texttt{res} = \textstyle\sum \texttt{a[..)}\ \}$$

but here we did not make use of any form of Induction ... or did we ... ?

The correctness of sumAux relies on the following arguments:

line 6: The pre-condition of sumAux must establish the mid-condition M_2.
$P_2 \wedge \texttt{a[..)} \approx \texttt{a[..)}_{pre} \wedge \texttt{(i == a.size)} \longrightarrow M_2$

line 7: The mid-condition M_2 must establish the post-condition of sumAux.
$M_2 \wedge \texttt{return 0} \longrightarrow Q_2$

line 9: The pre-condition of sumAux must establish the mid-condition M_3.
$P_2 \wedge \texttt{a[..)} \approx \texttt{a[..)}_{pre} \wedge \neg\texttt{(i == a.size)} \longrightarrow M_3$

line 10: The mid-condition M_3 must establish the pre-condition of sumAux.
$M_3 \longrightarrow P_2[\texttt{i} \mapsto \texttt{i+1}]$

line 11: The post-condition of sumAux must establish the mid-condition M_4.
$M_3[\texttt{a[..)} \mapsto \texttt{a[..)}_{old}] \wedge Q_2[\texttt{i} \mapsto \texttt{i+1}][\texttt{a[..)}_{pre} \mapsto \texttt{a[..)}_{old}]$
$\wedge$ `val res = a[i] + sumAux(a,i+1)` $\longrightarrow M_4$

line 12: The mid-condition M_4 must establish the post-condition of sumAux.
$M_4 \wedge \texttt{return res} \longrightarrow Q_2$

- lines 6 - 7 corresponds to proving the Base Case
- lines 9 - 12 correspond to proving the Inductive Step
- line 10 corresponds to checking that we can apply the Inductive Hypothesis
- line 11 use of Q_2 corresponds to use of the Inductive Hypothesis

Reasoning about Recursion - Conclusions

Reasoning about recursive functions in Kotlin resembles reasoning about recursive functions in Haskell:

- Reasoning about the cases where a function terminates corresponds to proving the base cases.
- Reasoning about the cases where a function calls itself recursively corresponds to proving the inductive steps.
- The point where we establish that the pre-condition of the callee holds, corresponds to establishing that the inductive hypothesis holds.
- The point where we establish that the post-condition of the callee implies the mid-condition of the caller, corresponds to the conclusion of the inductive step.