CO141 - Reasoning About Programs

Prelude

The content discussed here is part of CO141 - Reasoning About Programs (Computing MEng); taught by Sophia Drossopoulou, and Mark Wheelhouse, in Imperial College London during the academic year 2018/19. The notes are written for my personal use, and have no guarantee of being correct (although I hope it is, for my own sake). This should be used in conjunction with the (extremely detailed) notes.

Material Order

These notes are primarily based off the notes on CATe, as they cover the lecture slides in great detail. This is the order in which they are uploaded (and I'd assume the order in which they are taught).

1. *Introduction and Motivation (full notes).pdf*
2. *Stylised Proofs (full notes).pdf*
3. *Induction over natural numbers (full notes).pdf*
4. *Induction over Haskell data structures (full notes).pdf*
5. *Induction over recursive relations and functions (full notes).pdf*
6. *Java - Program Specifications (full notes).pdf*
7. *Java - Conditional Branches (full notes).pdf*
8. *Java - Method Calls (full notes).pdf*
9. *Java - Recursion (full notes).pdf*
10. *Java - Iteration Informal (full notes).pdf*
11. *Java Reasoning - summary.pdf*
12. *Loop case study.pdf*
13. *Java - Iteration Formal (full notes).pdf*
14. *Case Studies - overview (full notes).pdf*
15. *Case Studies - Dutch Flag Problem (full notes).pdf*
16. *Quicksort (full notes).pdf*

Introduction

This module will cover Proof by Induction from first principles, and shows how a recursive definition can implicitly introduce an inductive principle, how the inductive principle introduces a proof schema, and how the schema can be used to prove a property of a inductively defined set, relation or function. This will go into more detail regarding valid uses of quantifiers, when we're able to use the induction hypothesis, how auxiliary lemmas can help, as well as what cases we will need to strengthen properties to prove weaker ones.

Binding Conventions

The binding conventions in this module are the same as the ones used in **CO140 - Logic**; with the addition of $\forall x$, and $\exists x$ before $\neg$.

Formalising a Proof

For this section, we'll work on one example proof, with the given facts;

(1) a person is happy if all their children are rich

(2) someone is a supervillain if at least one of their parents is a supervillain

(3) all supervillains are rich

We want to show that "all supervillains are happy".

Proof in Natural Language

The given argument is that "All of a supervillain's children must therefore also be supervillains; and as all supervillains are rich, all the children of a supervillain are rich. Therefore, any supervillain is happy". However; we've made a few assumptions in this proof - we assume that a supervillain is always a person, and that a supervillain has children (as well as the fact that parent, and child aren't formally defined to be related concepts).

Therefore, we need to generalise statement (1) OR add an additional assumption (4);

(1) **someone** is happy if all their children are happy

(4) a supervillain is also a person

Formal Argument

Given:

(1) $\forall x[\text{person}(x) \land \forall y[\text{childof}(y, x) \to \text{rich}(y)] \to \text{happy}(x)]$

(2) $\forall x[\exists y[\text{childof}(x, y) \land \text{supervillain}(y)] \to \text{supervillain}(x)]$

(3) $\forall x[\text{supervillain}(x) \to \text{rich}(x)]$

(4) $\forall x[\text{supervillain}(x) \to \text{person}(x)]$

To show:

(α) $\forall x[\text{supervillain}(x) \to \text{happy}(x)]$

(Stylised) Proof:

	take arbitrary G	
(a1)	$\text{supervillain}(G)$	
(5)	$\text{person}(G) \land \forall y[\text{childof}(y, G) \to \text{rich}(y)] \to \text{happy}(G)$	from (1)
(6)	$\text{person}(G)$	from (a1), and (4)
	take arbitrary E	
(a2)	$\text{childof}(E, G)$	
(7)	$\text{supervillain}(E)$	from (a1), (a2), and (2)
(8)	$\text{rich}(E)$	from (3), and (7)
(9)	$\forall y[\text{childof}(y, G) \to \text{rich}(y)]$	from (a2), (8), and arbitrary E
(10)	$\text{happy}(G)$	from (5), (6), and (9)
(α)		from (a1), (10), and arb. G

While this can be proven fairly easily, and with great confidence, via first-order natural deduction, the proof is often tedious, and the intuition might be lost. On the other hand, stylised proofs have an explicit structure, few errors (compared to free-form) - although errors are still possible.

Our goal for our proofs are that they should only prove valid statements, are easy to read / check, and are able to highlight intuition behind arguments. The rules for a stylised proof are as follows;

1. write out, and name each given formula
2. write out, and name each goal formula
3. plan out proof, and name intermediate results
4. justify each step
5. size of each step can vary as appropriate

Planning, and justifying the the steps follow extremely similar rules to natural deduction - the rules for proving P are as follows;

• $P = Q \wedge R$	prove both Q, and R ($\wedge$I)
• $P = Q \vee R$	prove either Q, or R ($\vee$I)
• $P = Q \to R$	prove R from assuming Q ($\to$I)
• $P = \neg Q$	prove $\bot$ from assuming Q ($\neg$I)
• $P = \forall x[Q(x)]$	show $Q(c)$ from arbitrary c ($\forall$I)
• $P = \exists x[Q(x)]$	find some c, and show $Q(c)$ ($\exists$I)
• P	prove $\bot$ from assuming $\neg P$ (PC)

On the other hand, if we have proven P, we can do the following;

• $P = Q \wedge R$	both Q, and R hold ($\wedge$E)
• $P = Q \vee R$	case analysis ($\vee$I)
• $P = Q \wedge (Q \to R)$	R holds ($\to$E)
• $P = \forall x[Q(x)]$	$Q(c)$ holds for any c ($\forall$E)
• $P = \exists x[Q(x)]$	$Q(c)$ holds for some c ($\exists$E)
• $P = \bot$	anything holds ($\bot$E)
• $P = \neg Q$	$Q \to \bot$ holds ($\neg$E)
• P	use a lemma, or any logical equivalence

Another Example

Facts in Natural Language:

(i) a dragon is happy if all of its children can fly
(ii) all green dragons can fly
(iii) something is green if at least one of its parents is green
(iv) all the children of a dragon are also dragons
(v) if y is a child of x, then x is a parent of y

Given:

(1)	$\forall x[\text{dragon}(x) \wedge \forall y[\text{childof}(x, y) \to \text{fly}(y)] \to \text{happy(x)}]$	from (i)
(2)	$\forall x[\text{green}(x) \wedge \text{dragon}(x) \to \text{fly}(x)]$	from (ii)
(3)	$\forall x[\exists y[\text{parent of}(y, x) \wedge \text{green}(y)] \to \text{green}(y)]$	from (iii)
(4)	$\forall x[\forall y[\text{childof}(x, y) \wedge \text{dragon}(y) \to \text{dragon}(x)]]$	from (iv)
(5)	$\forall x[\forall y[\text{childof}(y, x) \to \text{parentof}(x, y)]]$	from (v)

To show:

(α)	$\forall x[\text{dragon}(x) \to (\text{green}(x) \to \text{happy}(x))]$	
($\times$)	$\forall x[\text{dragon}(x) \wedge \text{green}(x) \to \text{happy}(x)]$	(note - equivalent)

Proof:

	take arbitrary S	
(a1)	dragon(S)	
(a2)	green(S)	
(6)	$\forall x \forall y$[parentof(y, x) $\land$ green(y) $\to$ green(x)]	from (3)
(7)	$\forall x \forall y$[childof(x, y) $\land$ green(y) $\to$ green(x)]	from (5), and (6)
(8)	$\forall x$[childof(x, S) $\to$ green(x)]	from (a2), and (7)
(9)	$\forall x$[childof(x, S) $\to$ dragon(x)]	from (a1), and (4)
(10)	$\forall x$[childof(x, S) $\to$ green(x) $\land$ dragon(x)]	from (8), and (9)
(11)	$\forall x$[childof(x, S) $\to$ fly(x)]	from (2), and (10)
(12)	happy(S)	from (a1), (1), and (11)
(α)		from (a1), (a2), (12), and arb. S

Steps (6), (7), and (10) in particular require more justification; the justification of (7) requires us to prove something else, which can be done trivially with ND. Therefore only (6) will be proven;

Given:

(1) $\forall x[\exists y[P(x, y)] \to Q(x)]$

To show:

(α) $\forall x \forall y[P(x, y) \to Q(x)]$

Proof:

	take arbitrary c_1	
	take arbitrary c_2	
(a1)	$P(c_1, c_2)$	
(2)	$\exists y[P(c_1, y)] \to Q(c_1)$	from (1), where $x = c_1$
(3)	$\exists y[P(c_1, y)]$	from (a1), where $c_2 = y$
(4)	$Q(c_1)$	from (2), and (3)
(α)		from (a1), (4), and arbitrary c_1, c_2

Note that (7) requires us to prove $\forall u \forall v[R(v, y) \to Q(u, v)] \land \forall w \forall z[Q(z, w) \land S(z) \to S(w)] \to \forall x \forall y[R(x, y) \land S(y) \to S(x)]$, which isn't actually as difficult as it looks (only 16 lines in steps in ND). On the other hand, the proof for (10) requires $(A \to B) \land (A \to C) \to (A \to B \land C)$ - assume A, and you get both B, and C very quickly.

Induction over Natural Numbers

The notation used here is as follows; $\forall x : S[P(x)]$, where S is an **enumerable** set and $P \subseteq S$. Note that the notes use $\forall x : S.P(x)$, but I'm choosing to use the same notation as used in **CO140**, just to maintain consistency. The notation $P \subseteq S$ means that P is a property of elements in the set S. pos $\subset \mathbb{Z}$. The natural numbers, sequences, strings, or recursively defined data structures are enumerable sets, whereas $\mathbb{R}$ is not an enumerable set. These are some examples of enumerable sets;

- $\forall n : \mathbb{N}[7^n + 5$ is divisible by 3$]$
- $\forall$`xs` : `[a]`$\forall$`ys` : `[a]`[`length(xs ++ ys)` = `length(xs)` + `length(ys)`]

Mathematical Induction Principle

For any $P \subseteq \mathbb{N}$: $P(0) \land \forall k : \mathbb{N}[P(k) \to P(k + 1)] \to \forall n : \mathbb{N}[P(n)]$

This mirrors the definition in **CO142 - Discrete Structures**, by using Peano's axiom. Given a unary predicate P, and $P(0)$ is true, and for all natural numbers k, if $P(k)$ is true, then it follows that $P(\text{Succ}(k))$ is true. Then it follows that $P(n)$ is true for every natural number $n \in \mathbb{N}$.

Example - Sum of Natural Numbers

We want to prove $P(n)$, where $P(n) \triangleq \sum\limits_{i=0}^{n} i = \frac{n(n+1)}{2}$ - a formula which we should be used to seeing.

We need to formally write this as;

$$\sum_{i=0}^{0} i = \frac{0(0+1)}{2} \land \forall k : \mathbb{N}[\sum_{i=0}^{k} i = \frac{k(k+1)}{2} \to \sum_{i=0}^{k+1} i = \frac{(k+1)((k+1)+1)}{2}] \to \forall n : \mathbb{N}[\sum_{i=0}^{n} i = \frac{n(n+1)}{2}]$$

Remember that our aim is to create proofs that can be checked by others. This means justifying each step; writing what we know (givens), and what we aim to prove. All the steps should be explicit, but the granularity can vary depending on the confidence of the step. Intermediate results should be named, so that they can be used later, and variables that we are applying the induction principle on should be stated.

Base Case

Our aim here is to show $\sum\limits_{i=0}^{0} i = \frac{0(0+1)}{2}$

$$\begin{aligned}
\sum_{i=0}^{0} i &= 0 && \text{by definition of } \textstyle\sum \\
&= \tfrac{0(1)}{2} && \text{by arithmetic} \\
&= \tfrac{0(0+1)}{2} && \text{by arithmetic}
\end{aligned}$$

Inductive Step

Take arbitrary $k : \mathbb{N}$

Inductive hypothesis: $\sum\limits_{i=0}^{k} i = \frac{k(k+1)}{2}$

To show: $\sum\limits_{i=0}^{k+1} i = \frac{(k+1)((k+1)+1)}{2}$

$$\begin{aligned}
\sum_{i=0}^{k+1} i &= \sum_{i=0}^{k} i + (k+1) && \text{by definition of } \textstyle\sum \\
&= \tfrac{k(k+1)}{2} + (k+1) && \text{by inductive hypothesis} \\
&= \tfrac{k^2+3k+2}{2} && \text{by arithmetic} \\
&= \tfrac{(k+1)(k+2)}{2} && \text{by arithmetic} \\
&= \tfrac{(k+1)((k+1)+1)}{2} && \text{by arithmetic}
\end{aligned}$$

Example - $7^n + 5$ is divisible by 3

We want to prove $P(n)$, where $P(n) \triangleq 7^n + 5$ is divisible by 3. However, this isn't exactly a very formal defined, so we will rewrite it as $P(n) \triangleq \exists m : \mathbb{N}[7^n + 5 = 3m]$

We need to formally write this as;

$$\exists m : \mathbb{N}[7^0 + 5 = 3m] \land \forall k : \mathbb{N}[\exists m : \mathbb{N}[7^k + 5 = 3m] \to \exists m' : \mathbb{N}[7^{k+1} + 5 = 3m']] \to \forall n : \mathbb{N}[\exists m : \mathbb{N}[7^n + 5 = 3m]]$$

Base Case

Our aim here is to show $\exists m : \mathbb{N}[7^0 + 5 = 3m]$

$$\begin{aligned}
7^0 + 5 &= 1 + 5 && \text{by arithmetic} \\
&= 6 && \text{by arithmetic}
\end{aligned}$$

$= 3 \cdot 2$ by arithmetic

$\therefore \exists m : \mathbb{N}[7^0 + 5 = 3m]$

Inductive Step

Take arbitrary $k : \mathbb{N}$

Inductive hypothesis: $\exists m : \mathbb{N}[7^k + 5 = 3m]$

(1) $7^k + 5 = 3 \cdot m_1$ by inductive hypothesis, for some $m_1 : \mathbb{N}$

To show: $\exists m' : \mathbb{N}[7^{k+1} + 5 = 3m']$

$$
\begin{aligned}
7^{k+1} + 5 &= 7 \cdot 7^k + 5 && \text{by arithmetic} \\
&= (6 + 1) \cdot 7^k + 5 && \text{by arithmetic} \\
&= (6 \cdot 7^k + 7^k) + 5 && \text{by arithmetic} \\
&= 3 \cdot (2 \cdot 7^k) + (7^k + 5) && \text{by arithmetic} \\
&= 3 \cdot (2 \cdot 7^k) + 3 \cdot m_1 && \text{by (1)} \\
&= 3 \cdot [2 \cdot 7^k + m_1] && \text{by arithmetic}
\end{aligned}
$$

$\therefore \exists m' : \mathbb{N}[7^{k+1} + 5 = 3m']$

New Technique

Consider the Haskell program defined as;

```haskell
f :: Int -> Ratio Int
f 1 = 1/2
f n = 1/(n * (n + 1)) + f (n - 1)
```

And, we want to prove $\forall n \geq 1[\texttt{f}\ n = \frac{n}{n+1}]$. However, we cannot directly apply the mathematic induction principle on this, since the conclusion has a different form, and it's not defined for `f` 0 - hence we have no base case. Instead, we can do one of the following approaches;

1. prove $\forall n : \mathbb{N}[n \geq 1 \rightarrow \texttt{f}\ n = \frac{n}{n+1}]$
2. prove $\forall n : \mathbb{N}[\texttt{f}\ (n+1) = \frac{n+1}{n+2}]$
3. apply the mathematical induction technique

For practice, we will do the first approach; first we must formally write this out as;

$0 \geq 1 \rightarrow \texttt{f}\ 0 = \frac{0}{0+1} \wedge \forall k : \mathbb{N}[(k \geq 1 \rightarrow \texttt{f}\ k = \frac{k}{k+1}) \rightarrow ((k + 1) \geq 1 \rightarrow \texttt{f}\ (k + 1) = \frac{k+1}{k+2})] \rightarrow$
$\forall n : \mathbb{N}[n \geq 1 \rightarrow \texttt{f}\ n = \frac{n}{n+1}]$

Base Case

Our aim here is to show $0 \geq 1 \rightarrow \texttt{f}\ 0 = \frac{0}{0+1}$

This holds trivially, as we know $0 \geq 1$ is false, and anything follows from a falsity

Inductive Step

Take arbitrary $k : \mathbb{N}$

Inductive hypothesis: $k \geq 1 \rightarrow \texttt{f}\ k = \frac{k}{k+1}$

To show: $(k + 1) \geq 1 \rightarrow \texttt{f}\ (k + 1) = \frac{k+1}{k+2}$

(1.0) $k = 0$ by case

(1.1) $k + 1 = 1$ by arithmetic

(1.2) $\texttt{f}(k + 1) = \frac{1}{2}$ by def. of `f`, and (1.1)

(1.3) $\frac{k+1}{k+2} = \frac{1}{2}$ by (1.1)

$\therefore \texttt{f}(k + 1) = \frac{k+1}{k+2}$ by (1.2), and (1.3)

(2.0) $k > 0$ by case

(2.1) $k \geq 1$ by case

(2.2) $k + 1 \geq 2$ by (2.1), and arithmetic

(2.3) $\texttt{f}\ (k+1) = \frac{1}{(k+1)(k+2)} + \texttt{f}\ k$ by (2.2), and def. of `f`

(2.4) $\texttt{f}\ (k+1) = \frac{1}{(k+1)(k+2)} + \frac{k}{k+1}$ by (2.3), and inductive hypothesis

(2.5) $\texttt{f}\ (k+1) = \frac{1}{(k+1)(k+2)} + \frac{k(k+2)}{(k+1)(k+2)}$ by (2.4), and arithmetic

$\therefore \texttt{f}(k+1) = \frac{k+1}{k+2}$ by (2.5), and arithmetic

The third approach follows this; for any $P \subseteq \mathbb{Z}$, and any $m : \mathbb{Z}$, we have $P(m) \wedge \forall k \geq m[P(k) \rightarrow P(k+1)] \rightarrow \forall n \geq m[P(n)]$. This uses $\forall n \geq m[P(n)]$, as a shorthand for $\forall n : \mathbb{Z}[n \geq m \rightarrow P(n)]$. Note how this isn't any different from the principle; in reality, the principle is a "specific case" of the technique, where $m = 0$.

Now, using the technique with $m = 1$, we can do the original proof inductively;

$\texttt{f}\ 1 = \frac{1}{1+1} \wedge \forall k \geq 1[\texttt{f}\ k = \frac{k}{k+1} \rightarrow \texttt{f}\ (k+1) = \frac{k+1}{k+2}] \rightarrow \forall n \geq 1[\texttt{f}\ n = \frac{n}{n+1}]$

Base Case

Our aim here is to show $\texttt{f}\ 1 = \frac{1}{1+1}$

$$\begin{aligned} \texttt{f}\ 1 &= \tfrac{1}{2} && \text{by def. of } \texttt{f} \\ &= \tfrac{1}{1+1} && \text{by arithmetic} \end{aligned}$$

Inductive Step

Take arbitrary $k : \mathbb{Z}$

(a1) $k \geq 1$ assumption

Inductive hypothesis: $\texttt{f}\ k = \frac{k}{k+1}$

To show: $\texttt{f}\ (k+1) = \frac{k+1}{k+2}$

$$\begin{aligned} \texttt{f}\ (k+1) &= \tfrac{1}{(k+1)(k+2)} + \texttt{f}\ k && \text{by def. of } \texttt{f}\text{, and (a1)} \\ &= \tfrac{1}{(k+1)(k+2)} + \tfrac{k}{k+1} && \text{by inductive hypothesis} \\ &= \tfrac{1}{(k+1)(k+2)} + \tfrac{k(k+2)}{(k+1)(k+2)} && \text{by arithmetic} \\ &= \tfrac{k^2+2k+1}{(k+1)(k+2)} && \text{by arithmetic} \\ &= \tfrac{(k+1)^2}{(k+1)(k+2)} && \text{by arithmetic} \\ &= \tfrac{k+1}{k+2} && \text{by arithmetic} \end{aligned}$$

Strong Induction

While mathematical induction is powerful, it only allows for the inductive step $(k+1)$ to refer to the direct predecessor (k). On the other hand, strong induction allows the inductive step to refer to any predecessor, such as $k-1$, $k-2$, and so on. For example, if an algorithm was to recurse down by two units, we wouldn't be able to use the inductive hypothesis to replace `g (k - 1)` from doing the inductive step on $k+1$. An example of this is this Haskell program, with the property $\forall n : \mathbb{N}[\texttt{g}\ n = 3^n - 2^n]$;

```haskell
g :: Int -> Int
g 0 = 0
g 1 = 1
g n = (5 * g (n - 1)) - (6 * g (n - 2))
```

As such, we need the principle of strong induction; $P(0) \wedge \forall k : \mathbb{N}[\forall j \in [0..k][P(j)] \rightarrow P(k+1)] \rightarrow \forall n : \mathbb{N}[P(n)]$. Here we are using the following shorthand; $j \in [m..n]$ means $m \leq j \leq n$, and similarly $j \in [m..n)$ means $m \leq j < n$. Complete induction has the same base case, where we need to verify $P(0)$, however, the inductive step differs as we have to assume that $P(i)$ holds for all $i \leq k$, then show that $P(k+1)$ holds. If both of those hold, then it follows that $P(n)$ follows for all $n \in \mathbb{N}$ With this, we can now write down the strong induction principle on $\texttt{g}\ n = 3^n - 2^n$.

$$\texttt{g}\ 0 = 3^0 - 2^0 \land \forall k[\forall j \in [0..k][\texttt{g}\ j = 3^j - 2^j] \to \texttt{g}\ (k+1) = 3^{k+1} - 2^{k+1}] \to \forall n : \mathbb{N}[\texttt{g}\ n = 3^n - 2^n]$$

Base Case

Our aim here is to show $\texttt{g}\ 0 = 3^0 - 2^0$

$$\begin{aligned} \texttt{g}\ 0 &= 0 && \text{by def. of } \texttt{g} \\ &= 1 - 1 && \text{by arithmetic} \\ &= 3^0 - 2^0 && \text{by arithmetic} \end{aligned}$$

Inductive Step

Take arbitrary $k : \mathbb{N}$ $(k \neq 0)$

Case 1: $k = 0$

To show: $\texttt{g}\ 1 = 3^1 - 2^1$

$$\begin{aligned} \texttt{g}\ 1 &= 1 && \text{by def. of g} \\ &= 3 - 2 && \text{by arithmetic} \\ &= 3^1 - 2^1 && \text{by arithmetic} \end{aligned}$$

Case 2: $k \neq 0$

$$\begin{aligned} &(1) \quad k \geq 1 && \text{because } k : \mathbb{N}, \text{ and } k \neq 0 \text{ by case} \\ &(2) \quad k, k-1 \in [0..k] && \text{from (1)} \end{aligned}$$

Inductive hypothesis: $\forall j \in [0..k][\texttt{g}\ j = 3^j - 2^j]$

To show: $\texttt{g}\ (k+1) = 3^{k+1} - 2^{k+1}$

$$\begin{aligned} \texttt{g}\ (k+1) &= 5 \cdot \texttt{g}\ k - 6 \cdot \texttt{g}\ (k-1) && \text{by (1), and def. of g} \\ &= 5 \cdot (3^k - 2^k) - 6 \cdot (3^{k-1} - 2^{k-1}) && \text{by (2), and inductive hypothesis} \\ &= 5 \cdot 3 \cdot 3^{k-1} - 5 \cdot 2 \cdot 2^{k-1} - 6 \cdot 3^{k-1} + 6 \cdot 2^{k-1} && \text{by arithmetic} \\ &= 9 \cdot 3^{k-1} - 4 \cdot 2^{k-1} && \text{by arithmetic} \\ &= 3^{k+1} - 2^{k+1} && \text{by arithmetic} \end{aligned}$$

Once again, this principle can be applied for some m, and is therefore modified to be $P(m) \land \forall k : \mathbb{Z}[\forall j \in [m..k][P(j)] \to P(k+1)] \to \forall n \geq m[P(n)]$. The same shorthand we used before applies here.

Induction over Haskell Lists

The example we will be working on involves the follow Haskell functions;

```haskell
elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys) = x == y || elem x ys

subList :: Eq a => [a] -> [a] -> [a]
subList [] ys = []
subList (x:xs) ys
  | elem x ys = subList xs ys
  | otherwise = x:(subList xs ys)
```

As well as the specification $\forall xs : \texttt{[a]}\forall ys : \texttt{[a]}\forall z : \texttt{a}[z \in ys \to z \notin \texttt{subList}\ xs\ ys]$. Note that we use $z \in ys$ as shorthand for $\texttt{elem}\ z\ ys$. The function $\texttt{subList}\ xs\ ys$ removes all elements of ys from xs.

The goal here is to prove $\forall xs : \texttt{[a]}[Q(xs)]$, where $Q(xs) \triangleq \forall ys : \texttt{[a]}\forall z : \texttt{a}[z \in ys \to z \notin \texttt{subList}\ xs\ ys]$. Induction doesn't simply work here, as Q isn't defined over a set of numbers, but rather over the lists of `a`. Once again, we can take multiple approaches to this problem;

1. map lists to numbers, by expressing $Q \subseteq \texttt{[a]}$, with an equivalent $P \subseteq \mathbb{N}$

there's a proof of this in *Induction over Haskell data structures (full notes).pdf* - the point is that it's a bad idea, and structural induction should be used instead.

this method of reasoning is indirect - the property `length` is unrelated to P

2. use structural induction

   in a step of mathematical induction, we need to argue a property is inherited by the **predecessor** to its **successor**, such that 4 succeeds 3

   this concept needs to be generalised to other data structures, for example; can we say `1:2:3:[]` is a successor to `2:3:[]`?

Structural Induction Principle (Lists)

For any type `T`, and $P \subseteq$ `[T]`; we can say $P(\texttt{[]}) \land \forall vs : \texttt{[T]}\, \forall v : \texttt{T}[P(vs) \to P(v : vs)] \to \forall xs : \texttt{[T]}\,[P(xs)]$

Now, we can apply the structural induction principle to xs for `subList`;

$$\begin{aligned}
&\forall ys : \texttt{[a]}\forall z : \texttt{a}[z \in ys \to z \notin \texttt{subList}\ \texttt{[]}\ \ ys] \land \\
&\forall vs : \texttt{[a]}\forall v : \texttt{a}[(\forall ys : \texttt{[a]}\forall z : \texttt{a}[z \in ys \to z \notin \texttt{subList}\ \ vs\ ys]) \to \\
&\quad (\forall ys : \texttt{[a]}\forall z : \texttt{a}[z \in ys \to z \notin \texttt{subList}\ \ (v : vs)\ ys])] \to \\
&\forall xs : \texttt{[a]}[\forall ys : \texttt{[a]}\forall z : \texttt{a}[z \in ys \to z \notin \texttt{subList}\ \ xs\ ys]]
\end{aligned}$$

Base Case

Our aim here is to show $\forall ys : \texttt{[a]}\forall z : \texttt{a}[z \in ys \to z \notin \texttt{subList}\ \texttt{[]}\ \ ys]$

(a1)	$z \in ys$	otherwise, it's trivial to prove
(1)	`subList [] ` ys	by def. of `subList`
(2)	$z \notin$ `subList []`ys	by (1), and def. of `elem`

Inductive Step

Take arbitrary v : `a`, and vs : `[a]`

Inductive hypothesis: $\forall ys : \texttt{[a]}\forall z : \texttt{a}[z \in ys \to z \notin \texttt{subList}\ \ vs\ ys]$

To show: $\forall ys : \texttt{[a]}\forall z : \texttt{a}[z \in ys \to z \notin \texttt{subList}\ \ (v : vs)\ ys]$

(a1)	$z \in ys$	otherwise, it's trivial to prove
(0.1)	$z \notin \texttt{subList}\ \ vs\ ys$	by (a1), and inductive hypothesis
(1.0)	$v \in ys$	by case
(1.1)	$\texttt{subList}\ \ (v : vs)\ ys = \texttt{subList}\ \ vs\ ys$	by (1.0), and def. of `subList`
(1.2)	$z \notin \texttt{subList}\ \ (v : vs)\ ys$	by (0.1), and (1.1)
(2.0)	$v \notin ys$	by case
(2.1)	$\texttt{subList}\ \ (v : vs)\ ys = v : (\texttt{subList}\ \ vs\ ys)$	by (2.0), and def. of `subList`
(2.2)	$z \neq v$	by (a1), and (2.0)
(2.3)	$z \notin \texttt{subList}\ \ (v : vs)\ ys$	by (0.1), (2.2), (2.3), and def. of `elem`

Lemmas for Lists

The following lemmas apply for arbitrary `u:a`, and `us,vs,ws:[a]`

(A) `us++[] = us`

(B) `[]++us = us`

(C) `(u:us)++vs = u:(us++vs)`

(D) `(us++vs)++ws = us++(vs++ws)`

We will be using these lemmas on to prove $\forall xs : \texttt{[a]}\forall ys : \texttt{[a]}[\texttt{rev}\ (xs{+}{+}ys) = (\texttt{rev}\ ys){+}{+}(\texttt{rev}\ xs)]$, on the following Haskell function;

```
rev :: [a] -> [a]
rev []     = []
rev (x:xs) = (rev xs) ++ [x]
```

$$\begin{aligned}&\forall ys : \texttt{[a]}[\texttt{rev}\ (\texttt{[]} ++ ys) = (\texttt{rev}\ ys) ++ (\texttt{rev}\ \texttt{[]})] \land \\ &\forall vs : \texttt{[a]}\forall v : \texttt{a}[(\forall ys : \texttt{[a]}[\texttt{rev}\ (vs ++ ys) = (\texttt{rev}\ ys) ++ (\texttt{rev}\ vs)]) \to \\ &\quad(\forall ys : \texttt{[a]}[\texttt{rev}\ ((v : vs) ++ ys) = (\texttt{rev}\ ys) ++ (\texttt{rev}\ (v : vs))])] \to \\ &\forall xs : \texttt{[a]}[\forall ys : \texttt{[a]}[\texttt{rev}\ (xs ++ ys) = (\texttt{rev}\ ys) ++ (\texttt{rev}\ xs)]]\end{aligned}$$

Base Case

Our aim here is to show $\forall ys : \texttt{[a]}[\texttt{rev}\ (\texttt{[]} ++ ys) = (\texttt{rev}\ ys) ++ (\texttt{rev}\ \texttt{[]})]$

$$\begin{aligned}\texttt{rev}\ (\texttt{[]} ++ ys) &= \texttt{rev}\ ys && \text{by (B)}\\ &= (\texttt{rev}\ ys) ++ \texttt{[]} && \text{by (A)}\\ &= (\texttt{rev}\ ys) ++ (\texttt{rev}\ \texttt{[]}) && \text{by def. of } \texttt{rev}\end{aligned}$$

Inductive Step

Take arbitrary $z : \texttt{a}$, and $zs : \texttt{[a]}$

Inductive hypothesis: $\forall ys : \texttt{[a]}[\texttt{rev}\ (zs ++ ys) = (\texttt{rev}\ ys) ++ (\texttt{rev}\ zs)]$

To show: $\forall ys : \texttt{[a]}[\texttt{rev}\ ((z : zs) ++ ys) = (\texttt{rev}\ ys) ++ (\texttt{rev}\ (z : zs))]$

$$\begin{aligned}\texttt{rev}\ ((z : zs) ++ ys) &= \texttt{rev}\ (z : (zs ++ ys)) && \text{by (C)}\\ &= \texttt{rev}\ (zs ++ ys) ++ [z] && \text{by def. of } \texttt{rev}\\ &= ((\texttt{rev}\ ys) ++ (\texttt{rev}\ zs)) ++ [z] && \text{by inductive hypothesis}\\ &= (\texttt{rev}\ ys) ++ ((\texttt{rev}\ zs) ++ [z]) && \text{by (D)}\\ &= (\texttt{rev}\ ys) ++ (\texttt{rev}\ (z : zs)) && \text{by def. of } \texttt{rev}\end{aligned}$$

Unlike in induction over natural numbers, each list has an infinite number of successors (for example, both `3:[]`, and 877:[] are successors of `[]`), which is why it's important for us to consider arbitrary values. Intuitively, assuming the base case of $P(\texttt{[]})$ holds, we have $P(\texttt{[]}) \to (P(\texttt{x:[]}) \to (P(\texttt{y:x:[]}) \to P(\texttt{z:y:x[]})))$, and so on, hence it holds for all lists.

Induction over Haskell Data Structures

While we can now perform induction over the natural numbers, as well as lists of arbitrary type, we cannot yet perform induction over arbitrary data structures, which are recursively defined. Consider the following data structures;

- `data Nat = Zero | Succ Nat`

  $P(\texttt{Zero}) \land \forall n : \texttt{Nat}[P(n) \to P(\texttt{Succ}\ n)] \to \forall n : \texttt{Nat}[P(n)]$

- `data Tree a = Empty | Node (Tree a) a (Tree a)`

  $P(\texttt{Empty}) \land \forall t_1, t_2 : \texttt{Tree T}\forall x\texttt{T}[P(t_1) \land P(t_2) \to P(\texttt{Node}\ t_1\ x\ t_2)] \to \forall t : \texttt{Tree T}[P(t)]$

- `data BExp = T | F | BNt BExp | BAnd BExp BExp`

  $P(\texttt{T}) \land P(\texttt{F}) \land \forall b : \texttt{BExp}[P(b) \to P(\texttt{BNt}\ b)] \land \forall b_1, b_2 : \texttt{BExp}[P(b_1) \land P(b_2) \to P(\texttt{BAnd}\ b_1 b_2)] \to \forall b : \texttt{BExp}[P(b)]$

Proof Strategies

A situation may arise where a statement cannot be proven directly from induction; for example $\forall is\texttt{[Int]}[\texttt{sum}\ is = \texttt{sum\_tr}\ is\ 0]$, on the following functions;

```haskell
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)
```

Base Case

This is trivial to prove, also I'm tired

Inductive Step

Take arbitrary i : `Int`, and is : `[Int]`

Inductive hypothesis: $\texttt{sum}\ is = \texttt{sum\_tr}\ is\ 0$

To show: $\texttt{sum}\ (i : is) = \texttt{sum\_tr}\ (i : is)\ 0$

$$\begin{aligned}
\texttt{sum}\ (i : is) &= i + \texttt{sum}\ is && \text{by def. of } \texttt{sum} \\
&= i + \texttt{sum\_tr}\ is\ 0 && \text{by inductive hypothesis} \\
&= ??? && ??? \\
&= \texttt{sum\_tr}\ is\ i && ??? \\
&= \texttt{sum\_tr}\ (i : is)\ 0 && \text{by def. of } \texttt{sum\_tr}
\end{aligned}$$

As we cannot prove this directly by induction, we need to use one of the following approaches;

1. invent an auxiliary lemma

   the lemma in this case would be as follows; $\forall i : \texttt{Int} \forall k : \texttt{Int} \forall is : \texttt{[Int]}[i + (\texttt{sum\_tr}\ is\ k) = \texttt{sum\_tr}\ is\ (i + k)]$, which would be the justification for the penultimate line in the proof (skipping the i + 0 step).

2. strengthen original property

   instead of proving the original property, we will prove $\forall k : \texttt{Int} \forall is : \texttt{[Int]}[k + (\texttt{sum}\ is) = \texttt{sum\_tr}\ is\ k]$

   note that the stronger property is a general form of the original property (which is a specific case, where $k = 0$)

Base Case

Our aim here is to show $\forall k : \texttt{Int}[k + (\texttt{sum}\ \texttt{[]}) = \texttt{sum\_tr}\ \texttt{[]}\ k]$

Take arbitrary k : `Int`

$$\begin{aligned}
k + (\texttt{sum}\ \texttt{[]}) &= k + 0 && \text{by def. of } \texttt{sum} \\
&= k && \text{by arithmetic} \\
&= \texttt{sum\_tr}\ \texttt{[]}\ k && \text{by def. of } \texttt{sum\_tr}
\end{aligned}$$

Inductive Step

Take arbitrary i : `Int`, and is : `[Int]`

Inductive hypothesis: $\forall m : \texttt{Int}[m + (\texttt{sum}\ is) = \texttt{sum\_tr}\ is\ m]$

To show: $\forall n : \texttt{Int}[k + (\texttt{sum}\ (i : is)) = \texttt{sum\_tr}\ (i : is)\ n]$

Take arbitrary n : `Int`

$$\begin{aligned}
n + \texttt{sum}\ (i : is) &= n + i + \texttt{sum}\ is && \text{by def. of } \texttt{sum}\text{, and arithmetic} \\
&= \texttt{sum\_tr}\ is\ (n + i) && \text{by inductive hypothesis, and } m = n + i \\
&= \texttt{sum\_tr}\ (i : is)\ n && \text{by def. of } \texttt{sum\_tr}
\end{aligned}$$

There are also induction principles for more complex cases;

- `data T = C1 [Int] | C2 Int T`

  $$\forall is : \texttt{[Int]}[P(\texttt{C1}\ is)] \wedge \forall i : \texttt{Int}\forall t : \texttt{T}[P(t) \to P(\texttt{C2}\ i\ t)] \to \forall t : \texttt{T}[P(t)]$$

- `data Reds = BaseR | Red Greens`
  `data Greens = BaseG | Green Reds`

  $P(\texttt{BaseR} \wedge \forall g : \texttt{Greens}[Q(g) \to P(\texttt{Red}\ g)]) \wedge Q(\texttt{BaseG}) \wedge \forall r : \texttt{Reds}[P(r) \to Q(\texttt{Green}\ r)] \to \forall r : \texttt{Red}[P(r)] \wedge \forall g : \texttt{Green}[Q(g)]$

  $$P(\texttt{BaseR}) \wedge P(\texttt{Red BaseG}) \wedge \forall r : \texttt{Reds}[P(r) \to P(\texttt{Red Green}\ r)] \to \forall r : \texttt{Reds}[P(r)]$$

Going back to the cactus problem; we can express the cactus as a set of data types;

```
data Cactus = Root Tree
data Tree   = Leaf | Node Trees
data Trees  = Empty | Cons Tree Trees
```

This (`Tree`) then has the inductive principle $P(\text{Leaf}) \wedge \forall ts : \texttt{Trees}[Q(ts) \to P(\texttt{Node}\ ts)] \wedge Q(\texttt{Empty}) \wedge \forall t : \texttt{Tree}\forall ts : \texttt{Trees}[P(\texttt{t}) \wedge Q(\texttt{ts}) \to Q(\texttt{Cons}\ t\ ts)] \to \forall t : \texttt{Tree}[P(t)] \wedge \forall ts : \texttt{Trees}[Q(ts)]$.
However, it's trivial to see that `Trees` is essentially equivalent to `[Tree]` - therefore we can simplify the data types to;

```
data Cactus = Root Tree
data Tree   = Leaf | Node [Tree]
```

Now; the new `Tree` structure has the inductive principle; $P(\texttt{Leaf}) \wedge \forall ts : \texttt{Trees}[\forall t : \texttt{Tree}, \forall ts_1, ts_2 : \texttt{Trees}[ts = ts_1 + +[t] + +ts_2 \to P(t)] \to P(\texttt{Node}\ ts)] \to \forall t : \texttt{Tree}[P(t)]$. The meaning of the part in violet is that every item in ts satisfies the property P, I think.

In summary, by having a recursively defined data type, it allows us to construct an induction principle, which then leads to a proof schema. We are able to use first-order equivalences to swap quantifiers, which may allow for a proof on an easier equivalent property. Occasionally, lemmas will need to be strengthened, or we will need to create auxiliary lemmas.

Induction over Recursive Relations and Functions

As every inductively defined set creates a successor relation (+1 for $\mathbb{N}$, or `Node` for `Tree`), all inductively defined relations, and functions give rise to successor relations, therefore all inductively defined sets, relations, and functions give rise to an induction induction principle. Generalising the principle of induction from sets to functions, and relations, follows from the idea that relations, and functions, can both be represented through sets. Consider the "mystery" function `M`;

```
M :: Int -> Int
M m = M'(m, 0, 1)

M' :: (Int, Int, Int) -> Int
M' (i, cnt, acc)
  | i == cnt  = acc
  | otherwise = M'(i, cnt+1, 2*acc)
```

We can then evaluate the value of $\texttt{M}(n)$, let $n = 4$;

$$\begin{aligned}
\texttt{M}(4) &= \texttt{M'}(4, 0, 1) && \text{by def. of } \texttt{M} \\
&= \texttt{M'}(4, 0, 1) && \text{by def. of } \texttt{M'} \\
&= \texttt{M'}(4, 1, 2) && \text{by def. of } \texttt{M'} \\
&= \texttt{M'}(4, 2, 4) && \text{by def. of } \texttt{M'} \\
&= \texttt{M'}(4, 3, 8) && \text{by def. of } \texttt{M'}
\end{aligned}$$

$$= \texttt{M'}(4,4,16) \qquad \text{by def. of } \texttt{M'}$$
$$= 16 \qquad \text{by def. of } \texttt{M'}$$

In general, let it be **Assrt_1**, $\forall m : \mathbb{N}[\texttt{M}(m) = 2^m]$. While the definition for `M'` might feel forced, or deliberately implemented due to the increasing argument size, we care about it for a few reasons. Firstly, it's a tail-recursive function, and also once translated to an imperative language, it acts as a loop. For example, we can represent it in Java in a similar way;

```
cnt = 0;
acc = 0;
while !(m == cnt) {
  cnt = cnt + 1;
  acc = 2*acc;
}
return acc;
```

We want to establish **Assrt_2**: $\forall m, cnt, acc, p : \mathbb{N}[\texttt{M'}(m, cnt, acc) = p \to p = 2^{m-cnt} \cdot acc]$. The challenge with this is how it's recursively defined with increasing arguments. Due to this, we have to take one of the following approaches;

1. find some measure which decreases, rather than increases, with each level of recursion
2. count the number of recursive calls in `M'`
3. a new induction principle

In the first approach, we show that if $m < cnt$, then the function does not terminate ($m < cnt \to m \neq cnt$, as the termination condition is $m = cnt$). Hence **Assrt_2** can be shown as **Assrt_2'** $= \forall m, cnt, acc, p : \mathbb{N}[m \geq cnt \to \texttt{M'}(m, cnt, acc) = 2^{m-cnt} \cdot acc]$. However, in order to be able to prove **Assrt_2'**, we need to prove **Assrt_3**: $\forall k, m, cnt, acc : \mathbb{N}[k = m - cnt \to \texttt{M'}(m, cnt, acc) = 2^{m-cnt} \cdot acc]$. We also need to prove that **Assrt_3** $\to$ **Assrt_2'**.

Proving the latter is fairly straightforward, if $k = m - cnt$, and $k \geq 0$ (since it's a natural number), then it follows that $m - cnt \geq 0$, therefore $m \geq cnt$.

In order to prove **Assrt_3**, we need to apply mathematical induction over k, and formally apply the principle as;

- $\forall m, cnt, acc : \mathbb{N}[m = cnt \to \texttt{M'}(i, cnt, acc) = acc]$
- $\forall m, cnt, acc : \mathbb{N}[m > cnt \to]$

$$\begin{aligned}
&\forall m, cnt, acc : \mathbb{N}[0 = m - cnt \to \texttt{M'}(m, cnt, acc) = 2^{m-cnt} \cdot acc] \wedge \\
\forall k : \mathbb{N}[&\forall m, cnt, acc : \mathbb{N}[k = m - cnt \to \texttt{M'}(m, cnt, acc) = 2^{m-cnt} \cdot acc] \to \\
&\forall m, cnt, acc : \mathbb{N}[k + 1 = m - cnt \to \texttt{M'}(m, cnt, acc) = 2^{m-cnt} \cdot acc]] \to \\
&\forall k, m, cnt, acc : \mathbb{N}[k = m - cnt \to \texttt{M'}(m, cnt, acc) = 2^{m-cnt} \cdot acc]
\end{aligned}$$

Base Case

Our aim here is to show $\forall m, cnt, acc : \mathbb{N}[0 = m - cnt \to \texttt{M'}(m, cnt, acc) = 2^{m-cnt} \cdot acc]$

Take arbitrary $m, cnt, acc : \mathbb{N}$

$$\begin{aligned}
(a1)\quad & 0 = m - cnt && \text{assumption, otherwise the proof is trivial} \\
(1)\quad & m = cnt && \text{by (a1), and arithmetic} \\
\texttt{M'}(m, cnt, acc)\quad & = acc && \text{by (1), and def. of } \texttt{M'} \\
& = 2^0 \cdot acc && \text{by arithmetic} \\
& = 2^{m-cnt} \cdot acc && \text{by (a1)}
\end{aligned}$$

Inductive Step

Take arbitrary $m : \mathbb{N}$

Inductive hypothesis: $\forall acc_1, cnt_1 : \mathbb{N}[k = m - cnt_1 \to \texttt{M'}(m, cnt_1, acc_1) = 2^{m-cnt_1} \cdot acc_1]$

To show: $k + 1 = m - cnt \to \texttt{M'}(m, cnt, acc) = 2^{m-cnt} \cdot acc$

Take arbitrary $cnt, acc : \mathbb{N}$

(a1)	$k + 1 = m - cnt$	assumption, otherwise the proof is trivial
(1)	$k \geq 0$	by $k : \mathbb{N}$
(2)	$k + 1 \geq 1$	by arithmetic
(3)	$m - cnt \geq 1$	by (2), and arithmetic
(4)	$m \geq cnt + 1$	by arithmetic
(5)	$m > cnt$	by definition of $\geq$, arithmetic, and $m, cnt : \mathbb{N}$
(6)	$m \neq cnt$	by (5)
$\texttt{M'}(m, cnt, acc)$	$= \texttt{M'}(m, cnt + 1, 2 \cdot acc)$	by (6), and def. of `M'`
	$= \texttt{M'}(m, cnt_1, acc_1)$	where $cnt_1 = cnt + 1$, $acc_1 = 2 \cdot acc$
	$= 2^{m-cnt_1} \cdot acc_1$	by inductive hypothesis
	$= 2^{m-cnt-1} \cdot 2 \cdot acc$	by substitution
	$= 2^{m-cnt} \cdot acc$	by arithmetic

In the second approach, we can define `M'` in a new function `M''` as follows;

M_4: $m = cnt \to \texttt{M''}(m, cnt, acc) = (0, acc)$

M_5: $m \neq cnt \land \texttt{M''}(m, cnt + 1, 2 \cdot acc) = (k, n) \to \texttt{M''}(m, cnt, acc) = (k + 1, n)$

This then leads to the following;

Assrt_4: $\forall s : \mathbb{N} \forall m, cnt, acc, n : \mathbb{N}[\texttt{M''}(m, cnt, acc) = (s, n) \to n = 2^{m-cnt} \cdot acc]$ - proven by induction over s

Assrt_5: $\forall m, cnt, acc, n : \mathbb{N}[\texttt{M'}(m, cnt, acc) = n \to \texttt{M''}(m, cnt, acc) = (m - cnt, n)]$ - proven by induction over $m - cnt$ (basically, do induction over k as in; $\forall k, m, cnt, acc, n : \mathbb{N}[m - cnt = k \land \texttt{M'}(m, cnt, acc) = n \to \texttt{M''}(m, cnt, acc) = (m - cnt, n)]$)

Once again, we have to then prove that the two assertions implies the original; as in **Assrt_4** $\land$ **Assrt_5** $\to$ **Assrt_2**.

Generalising the Successor

While it is possible to reason about functions by purely mathematical induction, it's often indirect. For example, having to do induction over $m - cnt$, in the first approach, or counting recursive depth in the second. In both mathematical, and structural induction, we want to argue that properties are inherited from an element to their successor, which means "is constructed from" (hence, `x:[]` is constructed from `[]`, or 4 is constructed from 3). If we generalise the concept of a successor to **recursively defined relations, and functions**, we can say that it means "is defined in terms of", hence, `M''`(3, 2, 4) succeeds `M''`(3, 3, 8).

Generally, an inductive definition requires a finite set of "primitive" (base) cases, and a finite set of "composite" (inductive) derived cases. This inductive definition will lead to an inductive principle. Sets can be defined inductively, and relations, and functions **may** be defined inductively. The latter two can be defined in terms of a set (relations defined as the set of elements in the relation, and functions represented as a set of pairs) - if we can apply an induction to reason about elements in the sets, the induction follows for the relation, or function.

Inductively Defined Sets

Example 1 - Natural Numbers

Consider the set $S_{\mathbb{N}}$, defined over the alphabet `Zero`, and `Succ`, with the rules;

R1 $\texttt{Zero} \in S_{\mathbb{N}}$

R2 $\forall n[n \in S_{\mathbb{N}} \to \texttt{Succ}\ \ n \in S_{\mathbb{N}}]$

For example, in order to show that $\texttt{Succ Succ Succ Zero} \in S_{\mathbb{N}}$, we'd need to do the following;

(1)	$\texttt{Zero} \in S_{\mathbb{N}}$	by (R1)
(2)	$\texttt{Succ Zero} \in S_{\mathbb{N}}$	by (1), and (R2)
(3)	$\texttt{Succ Succ Zero} \in S_{\mathbb{N}}$	by (2), and (R2)
(4)	$\texttt{Succ Succ Succ Zero} \in S_{\mathbb{N}}$	by (3), and (R2)

For some property $Q \subseteq S_{\mathbb{N}}$, we have the primitive case `Zero`, and the composite case `Succ`; therefore it follows that the inductive principle for Q is $Q(\texttt{Zero}) \land \forall m \in S_{\mathbb{N}}[Q(m) \to Q(\texttt{Succ}\ \ m)] \to \forall n \in S_{\mathbb{N}}[Q(n)]$.

Example 2 - Tree

Consider the set `Tree`, defined with the rules;

R1 $i \in \mathbb{N} \to (\texttt{Leaf}\ \ i) \in \texttt{Tree}$

R2 $\forall t1, t2 \in \texttt{Tree}, c \in \texttt{Char}[(\texttt{Node}\ \ c\ t1\ t2) \in \texttt{Tree}]$

For example, to show that $(\texttt{Node}\ \ 'a'\ (\texttt{Leaf}\ \ 5)\ (\texttt{Node}\ \ 'b'\ (\texttt{Leaf}\ \ 9)\ (\texttt{Leaf}\ \ 3))) \in \texttt{Tree}$, we'd need to do the following;

(1)	$(\texttt{Leaf}\ \ 3) \in \texttt{Tree}$	by (R1), and $3 \in \mathbb{N}$
(2)	$(\texttt{Leaf}\ \ 9) \in \texttt{Tree}$	by (R1), and $9 \in \mathbb{N}$
(3)	$(\texttt{Leaf}\ \ 5) \in \texttt{Tree}$	by (R1), and $5 \in \mathbb{N}$
(4)	$(\texttt{Node}\ \ 'b'\ (\texttt{Leaf}\ \ 9)\ (\texttt{Leaf}\ \ 3))$	by (R2), (1), (2), and $'b' \in \texttt{Char}$
(5)	$(\texttt{Node}\ \ 'a'\ (\texttt{Leaf}\ \ 5)\ (\texttt{Node}\ \ 'b'\ (\texttt{Leaf}\ \ 9)\ (\texttt{Leaf}\ \ 3))) \in \texttt{Tree}$	by (R2), (3), (4), and $'a' \in \texttt{Char}$

Once again, for some property $Q \subseteq \texttt{Tree}$, we have the inductive principle; $\forall i \in \mathbb{N}[Q(\texttt{Leaf}\ \ i)] \land \forall t_1, t_2 \in \texttt{Tree} \forall c \in \texttt{Char}[Q(t_1) \land Q(t_2) \to Q(\texttt{Node}\ \ c\ t_1\ t_2)] \to \forall t \in \texttt{Tree}[Q(t)]$.

Example 3 - Ordered List

Considered the set $OL \subseteq N^*$, defined with the rules;

R1 $[] \in OL$

R2 $\forall i \in \mathbb{N}[i : [] \in OL]$

R3 $\forall i, j \in \mathbb{N}, js \in \mathbb{N}^*[i \leq j \land (j : js) \in OL \to i : j : js \in OL]$

For some property $Q \subseteq \mathbb{N}^*$, we get the inductive principle $Q([]) \land \forall i \in \mathbb{N}[Q(i : [])] \land$
$\forall i, j \in \mathbb{N}, js \in \mathbb{N}^*[i \leq j \land (j : js) \in OL \land Q(j : js) \to Q(i : j : js)] \to \forall ns \in OL[Q(ns)]$.

Inductively Defined Relations, and Predicates

Example 1 - "Strictly Less Than"

The predicate $SL \subseteq \mathbb{N} \times \mathbb{N}$, is defined by the rules;

R1 $\forall k \in \mathbb{N}[SL(0, k+1)]$

R2 $\forall m, n \in \mathbb{N}[SL(m, n) \to SL(m+1, n+1)]$

For some property $Q \subseteq \mathbb{N} \times \mathbb{N}$, SL gives the following inductive principle; $\forall k \in \mathbb{N}[Q(0, k+1)] \to \forall m, n \in \mathbb{N}[SL(m, n) \land Q(m, n) \to Q(m+1, n+1)] \to \forall m, n \in \mathbb{N}[SL(m, n) \to Q(m, n)]$

Example 2 - Even

Consider the predicate $E \subseteq S_{\mathbb{N}}$, which is defined by;

R1 $E(\texttt{Zero})$

R2 $\forall n \in S_{\mathbb{N}}[E(n) \to E(\texttt{Succ Succ Zero})]$

If we wanted to establish the property $Q \subseteq S_{\mathbb{N}}$, we're able to use the definition of E to obtain the inductive principle; $Q(\texttt{Zero}) \land \forall n \in S_{\mathbb{N}}[E(n) \land Q(n) \to Q(\texttt{Succ Succ } n)] \to \forall n \in S_{\mathbb{N}}[E(n) \to Q(n)]$.

Inductively Defined Functions

Example 1

Consider the function `F`, defined in Haskell as

```
F :: Int -> Int
F 0 = 0
F i = 1 + (i - 3)
```

Thus, this can be described in the following rules;

R1 $\forall i : \mathbb{Z}[i = 0 \to \texttt{F}\ i = 0]$

R2 $\forall i, j : \mathbb{Z}[i \neq 0 \land \texttt{F}\ (i-3) = j \to \texttt{F}\ i = 1 + j]$

Given a predicate $Q \subseteq \mathbb{Z} \times \mathbb{Z}$, we have the following inductive principle, used to prove $\forall i, j : \mathbb{Z}[\texttt{F}\ i = j \to Q(i, j)]$;

$Q(0, 0) \land \forall i, j : \mathbb{Z}[i \neq 0 \land \texttt{F}\ (i-3) = k \land Q(i-3, j) \to Q(i, 1+j)] \to \forall i, j : \mathbb{Z}[\texttt{F}\ i = j \to Q(i, j)]$

It's important to note that the function doesn't say that $\forall i : \mathbb{Z}[Q(i, \texttt{F}\ i)]$, since we have no guarantee of function termination. The property established is that **if** the function terminates ($\texttt{F}\ i = j$), then $Q(i, \texttt{F}\ i)$ is satisfied.

Example 2

The next example works on the function `G`, defined in Haskell as;

```
G :: (Nat, Nat) -> Nat
G (i, j) = G' (i, j, 0, 0)

G' :: (Nat, Nat, Nat, Nat) -> Nat
G' (i, j, cnt, acc)
  | i == cnt  = acc
  | otherwise = G' (i, j, cnt + 1, acc + j)
```

Like one of our previous induction over recursive functions questions, the arguments continue to increase in size, but also the number of execution steps decreases. Once again we can define a set of rules for the aforementioned function;

R1 $\forall i, j : \mathbb{N}[\texttt{G}\ (i, j) = \texttt{G'}\ (i, j, 0, 0)]$

R2 $\forall i, j, acc : \mathbb{N}[\texttt{G'}\ (i, j, i, acc) = acc]$

R3 $\forall i, j, cnt, acc, r : \mathbb{N}[i \neq cnt \land \texttt{G'}\ (i, j, cnt+1, acc+j) = r \rightarrow \texttt{G'}\ (i, j, cnt, acc) = r]$

The first rule establishes that $\texttt{G'}\ (i, j, 0, 0)$ terminates, and it has an equivalent value to $\texttt{G}\ (i, j)$. This can also be expressed as $\forall i, k : \mathbb{N}\exists k : \mathbb{N}[\texttt{G'}\ (i, j, 0, 0) = k \land \texttt{G}\ (i, j) = k]$

If we apply the inductive principle to the predicate $Q \subseteq \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, we get;

$\forall i, j, acc : \mathbb{N}[Q(i, j, i, acc, acc)] \land \forall i, j, acc, cnt, r : \mathbb{N}[i \neq cnt \land$
$\texttt{G'}\ (i, j, cnt+1, acc+j) = r \land Q(i, j, cnt+1, acc+j, r) \rightarrow Q(i, j, cnt, acc, r)] \rightarrow$
$\forall i, j, acc, cnt, r : \mathbb{N}[\texttt{G'}\ (i, j, cnt, acc) = r \rightarrow Q(i, j, cnt, acc, r)]$

The goal here is to prove $\forall i, j : \mathbb{N}[\texttt{G}\ (i, j) = i \cdot j]$. In order to prove this, we have to show $\forall i, j : \mathbb{N}\exists r : \mathbb{N}[\texttt{G'}\ (i, j, 0, 0) = r \land r = i \cdot j]$. However, to show this we need to show $\forall i, j : \mathbb{N}\exists r : \mathbb{N}[\texttt{G'}\ (i, j, 0, 0) = r]$, as well as $\forall i, j, r \in \mathbb{N}[\texttt{G'}\ (i, j, 0, 0) = r \rightarrow r = i \cdot j]$. In order to show this, we can show $\forall i, j, cnt, acc, n : \mathbb{N}[i - cnt = n \rightarrow \exists r : \mathbb{N}[\texttt{G'}\ (i, j, cnt, acc) = r]]$, and $\forall i, j, cnt, acc, r : \mathbb{N}[\texttt{G'}\ (i, j, cnt, acc) = r \rightarrow r = (i - cnt) \cdot j + acc]$, respectively. I have no idea why.

Take the property $Q(i, j, cnt, acc, r)$ to mean $r = (i - cnt) \cdot j + acc$

Base Case

Our aim here is to show $\forall i, j, acc : \mathbb{N} : acc = (i - i) \cdot j + acc$

Take arbitrary $i, j, acc : \mathbb{N}$

$$\begin{aligned} acc &= 0 + acc && \text{by arithmetic} \\ &= 0 \cdot j + acc && \text{by arithmetic} \\ &= (i - i) \cdot j + acc && \text{by arithmetic} \end{aligned}$$

Inductive Step

Take arbitrary $i, j, cnt, acc, r : \mathbb{N}$

(a1) $i \neq cnt$ — assumption, otherwise the proof is trivial

(a2) $\texttt{G'}\ (i, j, cnt+1, acc+j) = r$ — assumption, otherwise the proof is trivial

Inductive hypothesis: $r = (i - (cnt + 1)) \cdot j + (acc + j)$

To show: $r = (i - cnt) \cdot j + acc$

This is a simple proof, as the work went into creating the last property

$$\begin{aligned} r &= (i - (cnt + 1)) \cdot j + (acc + j) && \text{by inductive hypothesis} \\ &= (i - cnt - 1) \cdot j + acc + j && \text{by arithmetic} \\ &= (i - cnt) \cdot j - j + acc + j && \text{by arithmetic} \\ &= (i - cnt) \cdot j + acc && \text{by arithmetic} \end{aligned}$$

Example 3

Consider the DivMod function, defined as $\texttt{DM} : \mathbb{Z} \times \mathbb{Z} \mapsto \mathbb{Z} \times \mathbb{Z}$, and $\texttt{DM'} : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \mapsto \mathbb{Z} \times \mathbb{Z}$, which is defined in Haskell as;

```haskell
DM :: (Int, Int) -> (Int, Int)
DM (i, j) = DM'(i, j, 0, 0)

DM' :: (Int, Int, Int, Int) -> (Int, Int)
DM' (i, j, cnt, acc)
  | acc + j > i = (cnt, i - acc)
  | otherwise   = DM' (i, j, cnt + 1, acc + j)
```

Once again, we're able to obtain a set of rules for this;

R1 $\forall i, j : \mathbb{Z}[\texttt{DM}\ (i, j) = \texttt{DM'}\ (i, j, 0, 0)]$

R2 $\forall i, j, cnt, acc : \mathbb{Z}[acc + j > i \rightarrow \texttt{DM'}\ (i, j, cnt, acc) = (cnt, i - acc)]$

R3 $\forall i, j, cnt, acc, k1, k2 : \mathbb{Z}[\texttt{DM'}\ (i, j, cnt + 1, acc + j) = (k1, k2) \rightarrow \texttt{DM'}\ (i, j, cnt, acc) = (k1, k2)]$

At this point, we can spot a general pattern for functions with recursively defined helper functions. The first rule tends to establish that the original function calls the helper. The second is the non-recursive case of the helper function, and the third is in the format `(recursive condition)` $\land$ `(function next step)` $= (k_1, k_2, ..., k_n) \rightarrow$ `(function current step)` $= (k_1, k_2, ..., k_n)$. For some predicate $Q \subseteq \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$, we have the following inductive principle for `DM'`;

$\forall i, j, cnt, acc : \mathbb{Z}[acc + j > i \rightarrow Q(i, j, cnt, acc, cnt, i - acc)] \land$
$\forall i, j, cnt, acc, k1, k2 : \mathbb{Z}[acc + j \leq i \land \texttt{DM'}\ (i, j, cnt + 1, acc + j) = (k1, k2) \land$
$Q(i, j, cnt + 1, acc + j, k1, k2)] \rightarrow Q(i, j, cnt, acc, k1, k2)] \rightarrow$
$\forall i, j, cnt, acc, k1, k2 : \mathbb{Z}[\texttt{DM'}\ (i, j, cnt, acc) = (k1, k2) \rightarrow Q(i, j, cnt, acc, k1, k2)]$

The goal here is to show $\forall i, j \in \mathbb{Z}[\texttt{DM}\ (i, j) = (k1, k2) \rightarrow i = k1 \cdot j + k2 \land k2 < j]$

Conclusion to Haskell Induction

In conclusion, every inductively defined set, relation, or function has a successor relation. By having this successor relation, it allows for there to be an inductive principle (generic). Once this inductive principle can be specialised for a specific predicate, it gives rise for a proof schema, which can be proven with our rules of logic. By considering our `M` function, we can say the following;

R1 $\forall m : \mathbb{N}[\texttt{M}\ m = \texttt{M'}\ (m, 0, 1)]$

R2 $\forall m, cnt, acc : \mathbb{N}[m = cnt \rightarrow \texttt{M'}\ (m, cnt, acc) = acc]$

R3 $\forall m, cnt, acc, r : \mathbb{N}[m \neq cnt \land \texttt{M'}\ (m, cnt + 1, 2 \cdot acc) = r \rightarrow \texttt{M'}\ (m, cnt, acc)]$

This allows us to derive the general inductive principle as;

$\forall m, acc : \mathbb{N}[Q(m, m, acc, acc)] \land$
$\forall m, acc, cnt, r : \mathbb{N}[m \neq cnt \land \texttt{M'}\ (m, cnt + 1, 2 \cdot acc) = r \land Q(m, cnt + 1, 2 \cdot acc, r) \rightarrow Q(m, cnt, acc, r)]$
$\rightarrow \forall m, cnt, acc, r : \mathbb{N}[\texttt{M'}\ (m, cnt, acc) = r \rightarrow Q(m, cnt, acc, r)]$

However, we're defining $Q(m, cnt, acc, r)$ to be $r = 2^{m-cnt} \cdot acc$, therefore we can specialise the principle to be;

$\forall m, acc : \mathbb{N}[acc = 2^{m-m} \cdot acc] \land$
$\forall m, acc, cnt, r : \mathbb{N}[m \neq cnt \land \texttt{M'}\ (i, cnt + 1, 2 \cdot acc) = r \land r = 2^{m-(cnt+1)} \cdot 2 \cdot acc \rightarrow r = 2^{m-cnt} \cdot acc] \rightarrow$
$\forall m, cnt, acc, r : \mathbb{N}[\texttt{M'}\ (m, cnt, acc) = r \rightarrow r = 2^{m-cnt} \cdot acc]$

As the work went mostly into setting up the proof schema, it follows that the proof by induction here is trivial.

Specifications for Java Programs

Sequential Specification

As Java isn't a functional language, it therefore has side-effects, and is therefore more difficult to reason about as we need to consider how values change.

For example, consider an extremely simple Java program;

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

This can be translated into the following; for any integers u, and v, if the start value of `a` is u, and `x` has value v, after executing the code, both `a`, and `x` retain their original values, and `y` now has the value $u + v$. There is an **implicit universal quantification** of u, and v over the whole program specification. It's crucial that the quantification is over the entire specification, as having the post condition be represented as $\forall u, v[\texttt{a} = u \land \texttt{x} = v \land \texttt{y} = u + v]$ is incorrect, since that suggests `a`, and `x` holds all possible values after the code has been executed.

Unsurprisingly, not all programs consist of one line, therefore we may need to reason about a sequence of state changes as follows;

```
// PRE: P
somecodeblock0
// MID: P1
somecodeblock
// MID: P2
somecodeblock2
// POST: Q
```

We can assume P to show that P_1 holds after `somecodeblock0`, and so on. This can be expressed as a **Hoare Triple**; which is in the form $\{P\}$ `code` $\{Q\}$, which means "assuming that some property P holds, show that after `code` executes, Q will hold". For example, we could have the Hoare triple $\{0 \leq \texttt{x} < 10\}$ `x++;` $\{0 \leq \texttt{x} \leq 10\}$.

In order to prove that example, we'd need to show $0 \leq \texttt{x}_{\text{old}} < 10 \land \texttt{x} = \texttt{x}_{\text{old}} + 1 \to 0 \leq \texttt{x} \leq 10$. It's important to note the use of $\texttt{x} = \texttt{x}_{\text{old}} + 1$, as if we were to naively write $\texttt{x} = \texttt{x} + 1$, we would be able to imply anything from an obvious falsity. In general, for some variable `x`, we will have `x` refer to the most recent value (which is **after** the code execution), $\texttt{x}_{\text{old}}$ will refer to the value **before** the code execution, and $\texttt{x}_{\text{pre}}$ refers to the **original** value passed in. The convention is that the annotations are only used when variables might be modified by the code.

Hoare's assignment axiom is as follows; $\dfrac{P[\texttt{x} \mapsto \texttt{x}_{\text{old}}] \land \texttt{x} = E[\texttt{x} \mapsto \texttt{x}_{\text{old}}] \to Q}{\{P\}\ \texttt{x} = E;\ \{Q\}}$

Note that $[\texttt{x} \mapsto \texttt{x}_{\text{old}}]$ means any occurrence of `x` is replaced with $\texttt{x}_{\text{old}}$.

Method Specification

Consider the general layout of a method as

```
type someMethod(type x1, ..., type xn)
// PRE: P
// POST: Q
{
somecodeblock0
// MID: R
somecodeblock1
// MID: S
somecodeblock2
}
```

The following specification suggests that if $P[\texttt{x}_1 \mapsto v_1, ...\texttt{x}_n \mapsto v_n]$ holds before the call to $\texttt{someMethod}(v_1, v_2..., v_n)$, then $Q[\texttt{x}_1 \mapsto v_1, ...\texttt{x}_n \mapsto v_n]$ holds on the return. Note that v_i is a value, and $\texttt{x}_i$ is variable. Sometimes, this will be simplified as $\bar{\texttt{x}}$, or $\bar{v}$. The rules for the mid-conditions are the same as before.

For the types of conditions, we have the following rules;

- pre-condition
  - required to hold before code execution
  - assumption the code can make
- post-condition
  - expected to hold after code execution, given it terminates and pre-condition held

    - guarantee the code must make

- mid-condition
    - acts as a post-condition for preceding lines, and pre-condition for subsequent lines
    - assumption made at specific point in code
    - guaranteed by preceding code
    - assumed by subsequent code
    - ”stepping stone” about correctness
    - if it evaluates to false, it means the program cannot reach that point - often it means the mid-condition is incorrect

The following shorthands will be used in order to describe number ranges, and arrays;

- $i \in (m..n) \triangleq m < i < n$
- $i \in [m..n) \triangleq m \leq i < n$
- $i \in (m..n] \triangleq m < i \leq n$
- $i \in [m..n] \triangleq m \leq i \leq n$
- $v \in \texttt{a[m..n)} \triangleq \exists i \in [m..n)[0 \leq i < \texttt{a.length} \land \texttt{a}[i] = v]$

$\texttt{a}(m..n)$ is referred to as an **array slice**, and closed-open intervals for slices is often useful for reasoning. We often simplify our notation by ignoring bounds if they refer to the start, or end, of an array. With some arbitrary index i (existing in the array, presumably), we use the following notation;

`a[..i)`	the array slice from 0 up to i, but not including i
`a[i..)`	the array slice from i up to the end
`a[..)`	the whole array
$\texttt{a}_{\text{old/pre}}\texttt{[..)}$	refers to the older / previous value of an array's reference
$\texttt{a[..)}_{\text{old/pre}}$	refers to the older / previous value of an array's contents
$\texttt{a}_{\text{old/pre}}\texttt{[..)}_{\text{old/pre}}$	refers to the older / previous value of both
$\texttt{a[..)} \sim \texttt{b[..)}$	`a` is a permutation of `b` (and vice versa)
$\texttt{a[..)} \approx \texttt{b[..)}$	`a`, and `b` have identical contents
$\sum \texttt{a[x..y)}$	the sum of elements from index x, up to (but not including) index y
$\prod \texttt{a[x..y)}$	the product of elements from index x, up to (but not including) index y

In Java, we rarely overwrite an entire array, so referencing a reference is rarely used, but we will often reference the contents of the array, as it is common for methods to be able to update contents. We also often make use of the predicates `sorted`, `min`, and `max`, which do what the name suggests. They are only well-defined for arrays with comparable contents.

Consider the following function;

```
int biggest(int x, int y, int z)
// PRE (P): ⊤
// POST (Q): r = max{x, y, z}
{
int res;
if (x >= y) {
    res = x;
} else {
    res = y;
10 }
11 // MID ($M_1$): res = max{x, y}
12 if (z >= res) {
13     res = z;
14 }
15 // MID ($M_2$): res = max{z, max{x, y}}
16 return res;
17 }
```

It's important to note the mid-condition (M_2) in line 15, since we are describing a logical assertion, not what the code is doing. For example, writing `res = max{res, z}` is invalid, as it is only true when $\texttt{res} \geq \texttt{z}$, but that isn't guaranteed in the program specification, which doesn't constrain any of the input values (other than type). We also cannot write $\texttt{res = max\{res}_{\text{old}}\texttt{, z\}}$, since that is only used in our proofs - as a tool to distinguish between values at different points in code execution.

With that code, we end up with the following proof obligations;

$$P \land \texttt{if (x >= y) \{ res = x; \} else \{ res = y; \}} \to M_1$$
$$\top \land \texttt{res} = \texttt{max\{x, y\}} \to \texttt{res} = \texttt{max\{x, y\}}$$

$$M_1[\texttt{res} \mapsto \texttt{res}_{\text{old}}] \land \texttt{if (z >= res) \{ res = z; \}} \to M_2$$
$$\texttt{res}_{\text{old}} = \texttt{max\{x, y\}} \land \texttt{res} = \texttt{max\{z, res}_{\text{old}}\texttt{\}} \to \texttt{res} = \texttt{max\{z, max\{x, y\}\}}$$

$$M_2 \land \texttt{return res;} \to Q$$
$$\texttt{res = max\{z, max\{x, y\}\}} \land \texttt{r} = \texttt{res} \to \texttt{r} = \texttt{max\{x, y, z\}}$$

Note how M_2 doesn't have the mapping applied, since there is no change to any of the values. It's important to note that a piece of code may have multiple pre/post-conditions, and the post-condition(s) depends on the code as well as the pre-condition(s). We can use mid-conditions as "stepping-stones" in our reasoning.

Conditional Branches

Conditional branching is another construct that we will be using fairly often; as such we need to prove that branching code will still satisfy the given specification. Consider this generic example of a conditional branch;

```
// PRE: $P$
if (cond) {
code1;
} else {
code2;
}
// POST: $Q$
```

We can then express this as $\dfrac{\{P \land \texttt{cond}\}\ \texttt{code1}\ \{Q\} \qquad \{P \land \neg\texttt{cond}\}\ \texttt{code2}\ \{Q\}}{\{P\}\ \texttt{if (cond) \{ code1 \} else \{ code2 \}}\ \{Q\}}$

```
// PRE: $P$
if (cond) {
// MID: $P \land$ cond
code1;
// MID: $R_1$
} else {
// MID: $P \land \neg$cond
code2;
// MID: $R_2$
}
// POST: $Q$
```

Both `code1`, and `code2` can assume P, as well as their condition, but they **both** have to establish Q, (hence $R_1 \to Q$, and $R_2 \to Q$). By doing this, we can reduce the complexity of our proof by doing a case-by-case analysis, instead of having some general condition within both branches. Consider the first half of the `biggest` function that we defined previously;

```
int biggest(int x, int y, int z)
// PRE ($P$): $\top$
...
{
int res;
if (x >= y) {
    // MID ($P \land$ cond): x $\geq$ y
    res = x;
    // MID ($R_1$): res = x $\land$ x $\geq$ y
} else {
    // MID ($P \land \neg$cond): y $>$ x
    res = y;
    // MID ($R_2$): res = y $\land$ y $>$ x
}
// MID ($M_1$): res = max{x, y}
...
}
```

$$P \land \texttt{cond} \land \texttt{res = x;} \to R_1$$
$$\texttt{x} \geq \texttt{y} \land \texttt{res} = \texttt{x} \to \texttt{res} = \texttt{x} \land \texttt{x} \geq \texttt{y}$$

$$P \land \neg\texttt{cond} \land \texttt{res = y;} \to R_2$$
$$\texttt{y} > \texttt{x} \land \texttt{res} = \texttt{y} \to \texttt{res} = \texttt{y} \land \texttt{y} > \texttt{x}$$

$$R_1 \to M_1$$
$$\texttt{res} = \texttt{x} \land \texttt{x} \geq \texttt{y} \to \texttt{res} = \texttt{max\{x, y\}}$$

$$R_2 \to M_1$$
$$\texttt{res} = \texttt{y} \land \texttt{y} > \texttt{x} \to \texttt{res} = \texttt{max\{x, y\}}$$

While this complicates the proof obligations in this case, sometimes the conditional branches may be significantly different from each other.

Method Calls

Consider the following methods here;

```
void mainMethod() {
...
// MID: $P$
someMethod($v_1$, ..., $v_n$);
// MID: $Q$
}
```

```
void someMethod(type x_1, ..., type x_n)
// PRE: R
// POST: S
{
code
}
```

The first requirement is to show that $P \to R[\bar{\mathtt{x}} \mapsto \bar{v}]$, once that's proven, we can apply the method as;

$$\frac{P[\bar{v}[..) \mapsto \bar{v}[..)_{\text{old}}] \land S[\bar{\mathtt{x}} \mapsto \bar{v}, \bar{\mathtt{x}}\mathtt{[..)}_{\text{pre}} \mapsto \bar{v}[..)_{\text{old}}] \to Q}{\{P\}\ \mathtt{someMethod}(v_1,\ \ldots,\ v_n)\ \{Q\}}$$

The substitutions done in R, and S replace the variables in the method specification with the values used at the call point in our code. The second substitution in S, and the substitution in P consider the (possibly) updated program state caused by `someMethod`. Care should be taken with post-condition belonging to `someMethod`, as a reference to $\mathtt{a[..)}_{\text{pre}}$ would refer to the contents of the array before the method was called. Since Java is a call by value language, variables of primitive type aren't updated by the method call. However, when we pass variables of reference type (such as the array in this context), the contents **can** be updated.

If we consider return values from a method, such that we can change the code to be the following;

```
void mainMethod() {
...
// MID: P
res = someMethod(v_1, ..., v_n);
// MID: Q
}
type someMethod(type x_1, ..., type x_n) {
...
```

We need to be careful with the modification of `res`. Once again, the pre-condition of `someMethod` still needs to be satisfied in the same way, but we will also need additional substitutions as follows;

$$\frac{P[\bar{v}[..) \mapsto \bar{v}[..)_{\text{old}}][\mathtt{res} \mapsto \mathtt{res}_{\text{old}}] \land \mathtt{res} = \mathbf{r} \land S[\bar{\mathtt{x}} \mapsto \bar{v}, \bar{\mathtt{x}}\mathtt{[..)}_{\text{pre}} \mapsto \bar{v}[..)_{\text{old}}][\mathtt{res} \mapsto \mathtt{res}_{\text{old}}] \to Q}{\{P\}\ \mathtt{res\ =\ someMethod}(v_1,\ \ldots,\ v_n)\ \{Q\}}$$

Note that the substitution is after every other substitution, since it's needed to reflect the order of the code execution. The rest of the rules are the same as before, except the extra conjuncts account for the updating program state due to the assignment of the return value.

Consider an example, where we get the smallest value of a list by taking the first item of the sorted version of the given list, with the code in Java being defined as follows;

```
void sort(int[] b)
// PRE (P_1): b ≠ null
// POST (Q_1): b[..) ∼ b[..)_pre ∧ sorted(b[..))
{
...
}

int smallest(int[] a)
// PRE (P_2): a ≠ null ∧ a.length > 0
// POST (Q_2): r = min(a[..)_pre)
{
// MID (M_1): a[..) ≈ a[..)_pre ∧ a ≠ null ∧ a.length > 0
sort(a);
// MID (M_2): a[..) ∼ a[..)_pre ∧ sorted(a[..)) ∧ a.length > 0
int res = a[0];
// MID (M_3): a[..) ∼ a[..)_pre ∧ res = min(a[..))
return res;
}
```

Informally (and formally below), the obligations are as follows;

- $P_2[\texttt{a[..)} \mapsto \texttt{a[..)}_{\text{pre}}] \land \texttt{a[..)} \approx \texttt{a[..)}_{\text{pre}} \to M_1$

  $\texttt{a} \neq \texttt{null} \land \texttt{a.length} > 0 \land \texttt{a[..)} \approx \texttt{a[..)}_{\text{pre}} \to \texttt{a[..)} \approx \texttt{a[..)}_{\text{pre}} \land \texttt{a} \neq \texttt{null} \land \texttt{a.length} > 0$

  here we should remember that P_2 only describes the variables passed into the initial method call - to use this pre-condition, we need to use the initial $_{\text{pre}}$ versions of all variables; we also obtain the 3rd conjunct implicitly because no code has executed yet

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

  $\texttt{a[..)} \approx \texttt{a[..)}_{\text{pre}} \land \texttt{a} \neq \texttt{null} \land \texttt{a.length} > 0 \to \texttt{a} \neq \texttt{null}$

  no code has been executed in this obligation, therefore we don't need to consider the $_{\text{old}}$ variants - however we do need to substitute `b` with `a` in P_1

- $M_1[\texttt{a[..)} \mapsto \texttt{a[..)}_{\text{old}}] \land Q_1[\texttt{b} \mapsto \texttt{a}, \texttt{b[..)}_{\text{pre}} \mapsto \texttt{a[..)}_{\text{old}}] \to M_2$

  $\texttt{a[..)} \approx \texttt{a[..)}_{\text{pre}} \land \texttt{a} \neq \texttt{null} \land \texttt{a.length} > 0 \land \texttt{a[..)} \sim \texttt{a[..)}_{\text{old}} \land \texttt{sorted(a[..))} \to \texttt{a[..)} \sim \texttt{a[..)}_{\text{pre}} \land \texttt{sorted(a[..))} \land \texttt{a.length} > 0$

  this obligation considers the effect of the `sort` method, therefore we need to track the state of the input before and after `sort` is called - once again `b` is replaced with `a`, and we use the state $\texttt{b[..)}_{\text{pre}}$ to represent the state of the array on entry to `sort`, and $\texttt{a[..)}_{\text{old}}$ to represent the value just before the method call therefore $\texttt{b[..)}_{\text{pre}}$ is replaced with $\texttt{a[..)}_{\text{old}}$ in Q_2

- $M_2[\texttt{res} \mapsto \texttt{res}_{\text{old}}] \land \texttt{res = a[0]} \to M_3$

  $\texttt{a[..)} \approx \texttt{a[..)}_{\text{pre}} \land \texttt{a.length} > 0 \land \texttt{a[..)} \sim \texttt{a[..)}_{\text{old}} \land \texttt{sorted(a[..))} \land \texttt{res = a[0]} \to \texttt{a[..)}_{\text{pre}} \land \texttt{res = min(a[..))}$

  because `res` didn't exist before the previous line, and M_2 makes no reference to `res`, the substitution makes no change to M_2

- $M_3 \land \texttt{return res} \to Q_2$

  $\texttt{a[..)} \sim \texttt{a[..)}_{\text{pre}} \land \texttt{res = min(a[..))} \land \mathbf{r} = \texttt{res} \to \mathbf{r} = \texttt{min(a[..))}$

  no changes are done to any of the variables, since it only tracks the return value, there is no need to track old values in the program's state

While we've reasoned about the correctness of `smallest`, we don't know anything about `sort`. All we have done regarding it is that it satisfies its given specification. For code to be partially correct, it requires the code to satisfy the post-condition given it starts in a state satisfying its pre-condition, and terminates. Total correctness is the same, but guarantees the termination of the code.

Recursion

The techniques required for recursion are the same as the ones we've already visited for reasoning regarding method calls. Like we did in reasoning, we assume that the method specifications hold for anything called within the body (even if we haven't proven it). The example this works on is a recursive method for summing an array;

```
int sum(int[] a)
// PRE (P1): a ≠ null
// POST (Q1): a[..) ≈ a[..)pre ∧ r = ∑a[..)
{
  int res = sumAux(a, 0);
  // MID (M1): a[..) ≈ a[..)pre ∧ res = ∑a[..)
  return res;
}

int sumAux(int[] a, int i)
// PRE (P_2): a ≠ null ∧ 0 ≤ i ≤ a.length
// POST (Q_2): a[..] ≈ a[..]_pre ∧ r = ∑a[i..)
{
  if (i == a.length) {
    // MID (M_2): a[..) ≈ a[..)_pre ∧ i = a.length
    return 0;
  } else {
    // MID (M_3): a[..) ≈ a[..)_pre ∧ a ≠ null ∧ 0 ≤ i < a.length
    int val = a[i] + sumAux(a, i + 1);
    // MID (M_4): a[..) ≈ a[..)_pre ∧ val = ∑a[i..)
    return val;
  }
}
```

Once again, due to modular verification, we can verify the correctness of `sum`, and `sumAux` independently of each other. The reasoning for `sum` is much simpler, since we're treating `sumAux` as a black-box that just works;

- $P_1 \to P_2[\texttt{i} \mapsto 0]$

  $\texttt{a} \neq \texttt{null} \to \texttt{a} \neq \texttt{null} \land 0 \leq 0 \leq \texttt{a.length}$

  since there's no reference to the contents of the array in either of the preconditions, and no code has actually run, all that is required in this step is to substitute the call value for `sumAux`

- $P_1 \land Q_2[\texttt{a[..)}_{\text{pre}} \mapsto \texttt{a[..)}_{\text{old}}, \texttt{i} \mapsto 0] \land \texttt{a[..)}_{\text{old}} \approx \texttt{a[..)}_{\text{pre}} \land \texttt{res} = \mathbf{r} \to M_1$

  $\texttt{a} \neq \texttt{null} \land \texttt{a[..)} \approx \texttt{a[..)}_{\text{old}} \land \mathbf{r} = \sum \texttt{a[..)} \land \texttt{a[..)}_{\text{old}} \approx \texttt{a[..)}_{\text{pre}} \land \texttt{res} = \mathbf{r} \to \texttt{a[..)} \approx \texttt{a[..)}_{\text{pre}} \land \texttt{res} = \sum \texttt{a[..)}$

  any potential updates to `a` from `sumAux` needs to be carefully tracked, which means we need to differentiate between the contents passed in ($\texttt{a[..)}_{\text{old}}$), from the initial contents ($\texttt{a[..)}_{\text{pre}}$), and the current ones (`a[..)`)

  however, we can see that the contents are not modified before the `sumAux` call therefore we can write, implicitly, $\texttt{a[..)}_{\text{old}} \approx \texttt{a[..)}_{\text{pre}}$

- $M_1 \land \mathbf{r} = \texttt{res} \to Q_1$

  $\texttt{a[..)} \approx \texttt{a[..)}_{\text{pre}} \land \texttt{res} = \sum \texttt{a[..)} \land \mathbf{r} = \texttt{res} \to \texttt{a[..)} \approx \texttt{a[..)}_{\text{pre}} \land \mathbf{r} = \sum \texttt{a[..)}$

However, reasoning about `sumAux` is slightly more complex due to the requirement of handling the recursive calls. The formal proof obligations are as follows;

- $P_2 \land \texttt{a[..)} \approx \texttt{a[..)}_{\text{pre}} \land \texttt{i} = \texttt{a.length} \to M_2$

  $\texttt{a} \neq \texttt{null} \land 0 \leq \texttt{i} \leq \texttt{a.length} \land \texttt{a[..)} \approx \texttt{a[..)}_{\text{pre}} \land \texttt{i} = \texttt{a.length} \to \texttt{a[..)} \approx \texttt{a[..)}_{\text{pre}} \land \texttt{i} = \texttt{a.length}$

  there's no reference to the array contents in P_2, and implicitly there is no modification to the array from the code

- $M_2 \land \mathbf{r} = 0 \to Q_2$

  $\texttt{a[..)} \approx \texttt{a[..)}_{\text{pre}} \land \texttt{i} = \texttt{a.length} \to \texttt{a[..)} \approx \texttt{a[..)}_{\text{pre}} \land \mathbf{r} = \sum \texttt{a[i..)}$

  this utilises the property $\forall k[\sum \texttt{a[}k\texttt{..)} = 0]$

  we know that the array exists, since we have a value for the length of it, and proving $\mathbf{r}$ is trivial due to the property regarding the sum of an empty range

- $P_3 \land$ `a[..)` $\approx$ `a[..)`$_{\text{pre}} \land$ `i` $\neq$ `a.length` $\to M_3$

  `a` $\neq$ `null` $\land 0 \leq$ `i` $\leq$ `a.length` $\land$ `a[..)` $\approx$ `a[..)`$_{\text{pre}} \land$ `i` $\neq$ `a.length` $\to$ `a[..)` $\approx$ `a[..)`$_{\text{pre}} \land$ `a` $\neq$ `null` $\land 0 \leq$ `i` $<$ `a.length`

- $M_3 \to P_2[$`i` $\mapsto$ `i` $+ 1]$

  `a[..)` $\approx$ `a[..)`$_{\text{pre}} \land$ `a` $\neq$ `null` $\land 0 \leq$ `i` $<$ `a.length` $\to$ `a` $\neq$ `null` $\land 0 \leq$ `i` $+ 1 \leq$ `a.length`

- $M_3[$`a[..)` $\mapsto$ `a[..)`$_{\text{old}}] \land Q_2[$`a[..)`$_{\text{pre}} \mapsto$ `a[..)`$_{\text{old}}$, `i` $\mapsto$ `i` $+ 1] \land$ `val` $=$ `a[i]`$_{\text{old}} + \mathbf{r} \to M_4$

  `a[..)`$_{\text{old}} \approx$ `a[..)`$_{\text{pre}} \land$ `a` $\neq$ `null` $\land 0 \leq$ `i` $<$ `a.length` $\land$ `a[..)` $\approx$ `a[..)`$_{\text{old}} \land \mathbf{r} = \sum$ `a[i+1..)` $\land$ `val` $=$ `a[i]`$_{\text{old}} + \mathbf{r} \to$ `a[..)` $\approx$ `a[..)`$_{\text{pre}} \land$ `val` $= \sum$ `a[..)`

  the range for `i` is required, to ensure that the array access is legal

- $M_4 \land \mathbf{r} =$ `val` $\to Q_2$

  `a[..)` $\approx$ `a[..)`$_{\text{pre}} \land$ `val` $= \sum$ `a[..)` $\land \mathbf{r} =$ `val` $\to$ `a[..)` $\approx$ `a[..)`$_{\text{pre}} \land \mathbf{r} = \sum$ `a[..)`

For obvious reasons, reasoning regarding recursive functions in Java resembles reasoning about recursively defined functions in Haskell. Reasoning about the when the function terminates corresponds to proving the bases cases in Haskell, and the inductive steps are reasoning about when the function calls itself. Establishing the inductive hypothesis holds corresponds to the point where we establish the pre-condition of the callee holds, and the point where we establish the post-condition implying the mid-condition mirrors the conclusion of the inductive step.

Iteration (Informal)

The final programming construct we'll be reasoning about in this course is iteration. Since the common loop blocks; `while`, `for`, and `repeat-until` can all be expressed as `while` loops, it follows that we can focus solely on that. For example, the following `for` loop can be expressed as a `while` loop;

```
// FOR LOOP
for (int i = 0; i < a.length; i++) {
  res = res + a[i];
}

// WHILE LOOP
int i = 0;
while (i < a.length) {
  res = res + a[i];
  i++;
}
```

In general, we can summrise the structure of a `while` loop as follows;

```
while (cond) { // condition
  somecode // loop body
}
// MID: M - summarises effect of a loop
```

For example, consider the `sum` function we created earlier, but define it with iteration instead of recursion;

```
int sum(int[] a)
// PRE (P): a ≠ null
// POST (Q): a[..) ≈ a[..)pre ∧ r = ∑a[..)
{
  int res = 0;
  int i = 0;
7      // INV (I): a[..) ≈ a[..)_pre ∧ a ≠ null ∧ 0 ≤ i ≤ a.length ∧ res = ∑a[..i)
8      while (i < a.length) { (cond)
9        res = res + a[i];
10       i++;
11       // MID: → I
12     }
13     // MID (M): a[..) ≈ a[..)_pre ∧ res = ∑a[..)
14     return res;
15   }
```

Consider a property P, which satisfies the following conditions; $\forall n \in \mathbb{N}[P$ holds after n iterations], and $P \land \neg\texttt{cond} \rightarrow M$ (such that if P holds, and the loop condition doesn't, it reaches the mid-condition directly **outside** the loop). This property is called the invariant. This invariant generalises the effect of the loop, on some arbitrary number of iterations. Therefore we no longer need to track the mid-condtions before, after, or during the loop. However, we will still write a mid-condition after the loop, as it's more convenient to do so, and then use that to construct our invariant.

There's a similarity between the first requirement of the loop invariant, and the induction principle. In order to show that P holds after n iterations, where n is an arbitrary natural number, we need to show that P holds after no iterations ($n = 0$), hence P holds before the loop, as well as showing that if P holds after k iterations, it implies that it holds after $k + 1$ iterations. The latter means that if both P, and `cond` hold, and after the execution of the loop body, P will still hold. Note that normally line 11 wouldn't be written, but it's there to show that the invariant is reestablished after the execution of the loop body. The following predicates are defined as follows;

$$
\begin{aligned}
I &\triangleq \texttt{a[..)} \approx \texttt{a[..)}_{\text{pre}} \land \texttt{a} \neq \texttt{null} \land 0 \leq \texttt{i} \leq \texttt{a.length} \land \texttt{res} = \sum \texttt{a[..i)} \\
\texttt{cond} &\triangleq \texttt{i} < \texttt{a.length} \\
M &\triangleq \texttt{a[..)} \approx \texttt{a[..)}_{\text{pre}} \land \texttt{res} = \sum \texttt{a[..)}
\end{aligned}
$$

This therefore comes with the following proof obligations - the code right before the loop must establish I, the body of the loop must show that I holds, as long as `cond` holds, and finally $I \land \neg\texttt{cond} \rightarrow M$, immediately after the loop. For us to be able to reason about the behaviour of the loop, we need the invariant to hold at the start, and end, of every iteration (including the before we run the loop body for the first time, and after we run it for the final time). As such, these are the proofs we have to do;

- $P \land \texttt{res} = 0 \land \texttt{i} = 0 \land \texttt{a[..)} \approx \texttt{a[..)}_{\text{pre}} \rightarrow I$

  here, we can assume the pre-condition, and list the explicit effects of the code (the variable declaration), as well as the implicit effects (the array not being modified)

- $I[\texttt{a[..)} \mapsto \texttt{a[..)}_{\text{old}}, \texttt{i} \mapsto \texttt{i}_{\text{old}}, \texttt{res} \mapsto \texttt{res}_{\text{old}}] \land \texttt{i}_{\text{old}} < \texttt{a.length} \land \texttt{res} = \texttt{res}_{\text{old}} + \texttt{a[i]}_{\text{old}} \land \texttt{i} = \texttt{i}_{\text{old}} + 1 \land \texttt{a[..)} \approx \texttt{a[..)}_{\text{old}} \rightarrow I$

  since the loop body potentially changes the states, all of the program variables are tracked in our invariant

- $I \land \texttt{i} \geq \texttt{a.length} \rightarrow M$

  since no code is being executed, there's no need to differentiate between the current, and older values of our program state

However, we currently have no way of tracking the progress of our iteration. This means we need to find some integer expression, called the **variant**, which is larger than some value after the loop body is executed, and decreases in every iteration. If this holds, then the loop will terminate, in our case, the variant would be `a.length - i`. The invariant lets us know that the program execution is going as planned, as long as the loop body maintains its validtiy. However, the variant measures the progress of the iteration, therfore by proving the variant is bounded, and decreasing every iteration, we can use the invariant to prove the mid-condition.

In general, to help us design an invariant, we should represent the state of the program at the beginning of each loop iteration as a diagram. The post-condition should be used to find a mid-condition after the loop, and this mid-condition (along with the loop condition) should be used to help find the invariant.

Iteration (Formal)

Consider the general form of a loop as;

```
// PRE: $P$
// INV: $I$
// VAR: $V$
while (cond) {
  body
}
// MID: $M$
// POST: $Q$
```

The Hoare logic rule displayed below captures **partial** correctness, and imposes the three proof obligations on the top line.

$$\frac{P \to I \qquad \{I \land \texttt{cond}\}\ \texttt{body}\ \{I\} \qquad I \land \neg\texttt{cond} \to Q}{\{P\}\ \texttt{while (cond) \{ body \}}\ \{Q\}}$$

In order for partial correctness to hold, we need the following;

1. I holds before the loop is entered
2. the loop body re-establishes I, if the condition holds
3. termination of the loop, and I, imply the mid-condition M

Total correctness needs the above, as well as;

4. V is bounded

   $\exists c \in \mathbb{Z}[\{I \land \texttt{cond}\}\ \texttt{body}\ \{V \geq c\}]$

   note that c represents a lower-bound for V, since it is decreasing

   generally any well-founded ordering with a limit works, but we will mainly look for an integer expression with a fixed lower bound (normally 0 in our cases)
5. V decreases with each iteration

   $\{I \land \texttt{cond}\}\ \texttt{body}\ \{V_{\text{old}} > V\}$

   note that V_{old} here is the value of V before running `body`

   this obligation requires us to prove the variant decreases towards c on **every** iteration

In order to prove $\{I \land \texttt{cond}\}\ \texttt{body}\ \{I\}$, we are required to show $I[\overline{\texttt{mod}} \mapsto \overline{\texttt{mod}}_{\text{old}}] \land \texttt{cond}[\overline{\texttt{mod}} \mapsto \overline{\texttt{mod}}_{\text{old}}] \land \texttt{body-effect} \to I$. Note that here, $\overline{\texttt{mod}}$ represents all the variables that ae modified by `body`.

Consider the following example, represented by the code below;

```
int culSum(int[] a)
// PRE ($P$): a $\neq$ null
// POST ($Q$): r = $\sum$a[..)$_{\texttt{pre}}$ $\land$ $\forall$k $\in$ [0..a.length)[a[k] = $\sum$a[..k+1)$_{\texttt{pre}}$]
{
  // MID ($M_0$): a $\neq$ null $\land$ a[..) $\approx$ a[..)$_{\texttt{pre}}$
  int res = 0;
  int i = 0;
  // INV ($I$): a $\neq$ null $\land$ 0 $\leq$ i $\leq$ a.length $\land$ a[i..) $\approx$ a[i..)$_{\texttt{pre}}$ $\land$ res = $\sum$a[..i)
      $_{\texttt{pre}}$ $\land$ $\forall$k $\in$ [0..i)[a[k] = $\sum$a[..k+1)$_{\texttt{pre}}$]
```

```
 9   // VAR (V): a.length - i
10   while (i < a.length) {
11     res = res + a[i];
12     a[i] = res;
13     i++;
14   }
15   // MID (M): res = ∑a[..)pre ∧ ∀k ∈ [0..a.length)[a[k] = ∑a[..k+1)pre]
16   return res;
17 }
```

This function also calculates the sum of an array, however it replaces each element of an array with the cumulative sum up to that point. Since this doesn't modify a, as Java is call by value, we don't need to track changes to a, or a.length. The entire proof is as follows;

Invariant holds before loop entry

Given:

(1)	$\texttt{a} \neq \texttt{null}$	from M_0
(2)	$\texttt{a[..)}_{\text{old}} \approx \texttt{a[..)}_{\text{pre}}$	from M_0
(3)	$\texttt{res} = 0$	code line 6
(4)	$\texttt{i} = 0$	code line 7
(5)	$\texttt{a[..)} \approx \texttt{a[..)}_{\text{old}}$	implciit from code

To show:

(α)	$\texttt{a} \neq \texttt{null}$	I
(β)	$0 \leq \texttt{i} \leq \texttt{a.length}$	I
(γ)	$\texttt{a[i..)} \approx \texttt{a[i..)}_{\text{pre}}$	I
(δ)	$\texttt{res} = \sum \texttt{a[..i)}_{\text{pre}}$	I
(ϵ)	$\forall \texttt{k} \in [0..\texttt{i})[\texttt{a[k]} = \sum \texttt{a[..k+1)}_{\text{pre}}]$	I

Proof:

(α)		follows immediately from (1)
(6)	$0 \leq 0 \leq \texttt{a.length}$	from (1)
(β)		follows from (4), and (6)
(7)	$\texttt{a[..)} \approx \texttt{a[..)}_{\text{pre}}$	from (2), and (5)
(8)	$\texttt{a[0..)} \approx \texttt{a[0..)}_{\text{pre}}$	from (7), and def. of a[..)
(γ)		follows from (4), and (8)
(9)	$\sum \texttt{a[..0)}_{\text{pre}} = 0$	from def. of $\sum$
(10)	$\sum \texttt{a[..i)}_{\text{pre}} = 0$	from (4), and (9)
(δ)		follows from (3), and (10)
(11)	$\forall \texttt{k} \in [0..0)[\texttt{a[k]} = \sum \texttt{a[..k+1)}_{\text{pre}}]$	from empty range
(ϵ)		follows from (4), and (11)

Loop body re-establishes invariant

Given:

(1)	$\texttt{a} \neq \texttt{null}$	I
(2)	$0 \leq \texttt{i}_{\text{old}} \leq \texttt{a.length}$	I
(3)	$\texttt{a[i}_{\text{old}}\texttt{..)}_{\text{old}} \approx \texttt{a[i}_{\text{old}}\texttt{..)}_{\text{pre}}$	I
(4)	$\texttt{res} = \sum \texttt{a[..i}_{\text{old}}\texttt{)}_{\text{pre}}$	I
(5)	$\forall \texttt{k} \in [0..\texttt{i}_{\text{old}})[\texttt{a[k]}_{\text{old}} = \sum \texttt{a[..k+1)}_{\text{pre}}]$	I
(6)	$\texttt{i}_{\text{old}} < \texttt{a.length}$	cond

(7)	$\texttt{res} = \texttt{res}_{\text{old}} + \texttt{a[i}_{\text{old}}\texttt{]}_{\text{old}}$	code line 12
(8)	$\texttt{a[i}_{\text{old}}\texttt{]} = \texttt{res}$	code line 13
(9)	$\texttt{i} = \texttt{i}_{\text{old}} + 1$	code line 14
(10)	$\texttt{a[..)} \approx \texttt{a[..i}_{\text{old}}\texttt{)}_{\text{old}} + +\texttt{a[i}_{\text{old}}\texttt{]} + +\texttt{a[i}_{\text{old}}\texttt{+1..)}_{\text{old}}$	implciit from code

To show:

(α)	$\texttt{a} \neq \texttt{null}$	I
(β)	$0 \leq \texttt{i} \leq \texttt{a.length}$	I
(γ)	$\texttt{a[i..)} \approx \texttt{a[i..)}_{\text{pre}}$	I
(δ)	$\texttt{res} = \sum \texttt{a[..i)}_{\text{pre}}$	I
(ϵ)	$\forall \texttt{k} \in [0..\texttt{i})[\texttt{a[k]} = \sum \texttt{a[..k+1)}_{\text{pre}}]$	I