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) $\forall x[\text{person}(x) \wedge \forall y[\text{childof}(y, x) \rightarrow \text{rich}(y)] \rightarrow \text{happy}(x)]$

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

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

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

To show:

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

(Stylised) Proof:

take arbitrary G

(a1) $\text{supervillain}(G)$

(5) $\text{person}(G) \wedge \forall y[\text{childof}(y, G) \rightarrow \text{rich}(y)] \rightarrow \text{happy}(G)$ — 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) $\forall y[\text{childof}(y, G) \rightarrow \text{rich}(y)]$ — 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)	$\text{dragon}(S)$	
(a2)	$\text{green}(S)$	
(6)	$\forall x \forall y[\text{parentof}(y, x) \wedge \text{green}(y) \rightarrow \text{green}(x)]$	from (3)
(7)	$\forall x \forall y[\text{childof}(x, y) \wedge \text{green(y)} \rightarrow \text{green}(x)]$	from (5), and (6)
(8)	$\forall x[\text{childof}(x, S) \rightarrow \text{green}(x)]$	from (a2), and (7)
(9)	$\forall x[\text{childof}(x, S) \rightarrow \text{dragon}(x)]$	from (a1), and (4)
(10)	$\forall x[\text{childof}(x, S) \rightarrow \text{green}(x) \wedge \text{dragon}(x)]$	from (8), and (9)
(11)	$\forall x[\text{childof}(x, S) \rightarrow \text{fly}(x)]$	from (2), and (10)
(12)	$\text{happy}(S)$	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) $\forall x[\exists y[P(x, y)] \rightarrow Q(x)]$

To show:

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

Proof:

	take arbitrary c_1	
	take arbitrary c_2	
(a1)	$P(c_1, c_2)$	
(2)	$\exists y[P(c_1, y)] \rightarrow Q(c_1)$	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 $\forall u \forall v[R(v, y) \rightarrow Q(u, v)] \wedge \forall w \forall z[Q(z, w) \wedge S(z) \rightarrow S(w)] \rightarrow \forall x \forall y[R(x, y) \wedge S(y) \rightarrow S(x)]$, 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 $(A \rightarrow B) \wedge (A \rightarrow C) \rightarrow (A \rightarrow B \wedge C)$ - 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. $\text{pos} \subset \mathbb{Z}$. 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;

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

Mathematical Induction Principle

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

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 $P(n) \triangleq \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ - a formula which we should be used to seeing.

We need to formally write this as;

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

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 $\sum_{i=0}^{0} i = \frac{0(0+1)}{2}$

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

Inductive Step

Take arbitrary $k : \mathbb{N}$

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

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

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

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

$7^{k+1} + 5 = 7 \cdot 7^k + 5$ by arithmetic

$= (6 + 1) \cdot 7^k + 5$ by arithmetic

$= (6 \cdot 7^k + 7^k) + 5$ by arithmetic

$= 3 \cdot (2 \cdot 7^k) + (7^k + 5)$ by arithmetic

$= 3 \cdot (2 \cdot 7^k) + 3 \cdot m_1$ by (1)

$= 3 \cdot [2 \cdot 7^k + m_1]$ by arithmetic

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

New Technique

Consider the Haskell program defined as;

```
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 $\texttt{f}$
(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 $\texttt{f}\ \ 1 = \frac{1}{1+1}$

$\texttt{f}\ \ 1$	$= \frac{1}{2}$	by def. of $\texttt{f}$
	$= \frac{1}{1+1}$	by arithmetic

Inductive Step

Take arbitrary $k : \mathbb{Z}$

(a1)	$k \geq 1$	assumption

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

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

$\texttt{f}\ \ (k+1)$	$= \frac{1}{(k+1)(k+2)} + \texttt{f}\ \ k$	by def. of $\texttt{f}$, and (a1)
	$= \frac{1}{(k+1)(k+2)} + \frac{k}{k+1}$	by inductive hypothesis
	$= \frac{1}{(k+1)(k+2)} + \frac{k(k+2)}{(k+1)(k+2)}$	by arithmetic
	$= \frac{k^2+2k+1}{(k+1)(k+2)}$	by arithmetic
	$= \frac{(k+1)^2}{(k+1)(k+2)}$	by arithmetic
	$= \frac{k+1}{k+2}$	by arithmetic

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 $\forall n : \mathbb{N}[\texttt{g}\ \ n = 3^n - 2^n]$;

```
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 $\mathtt{g}\ n = 3^n - 2^n$.

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

Base Case

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

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

Inductive Step

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

Case 1: $k = 0$

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

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

Case 2: $k \neq 0$

(1) $k \geq 1$ because $k : \mathbb{N}$, and $k \neq 0$ by case

(2) $k, k-1 \in [0..k]$ from (1)

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

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

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

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

Induction over Haskell Lists

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

```
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 $\forall xs : \mathtt{[a]} \forall ys : \mathtt{[a]} \forall z : \mathtt{a}[z \in ys \rightarrow z \notin \mathtt{subList}\ xs\ ys]$. Note that we use $z \in ys$ as shorthand for $\mathtt{elem}\ z\ ys$. The function $\mathtt{subList}\ xs\ ys$ removes all elements of ys from xs.

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

1. map lists to numbers, by expressing $Q \subseteq$ `[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 $P(\texttt{[]}) \wedge \forall vs : \texttt{[T]} \forall v : \texttt{T}[P(vs) \rightarrow P(v : vs)] \rightarrow \forall xs : \texttt{[T]}[P(xs)]$

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

$$\forall ys : \texttt{[a]} \forall z : \texttt{a}[z \in ys \rightarrow z \notin \texttt{subList [] } ys] \wedge$$
$$\forall vs : \texttt{[a]} \forall v : \texttt{a}[(\forall ys : \texttt{[a]} \forall z : \texttt{a}[z \in ys \rightarrow z \notin \texttt{subList } vs\ ys]) \rightarrow$$
$$(\forall ys : \texttt{[a]} \forall z : \texttt{a}[z \in ys \rightarrow z \notin \texttt{subList } (v : vs)\ ys])] \rightarrow$$
$$\forall xs : \texttt{[a]}[\forall ys : \texttt{[a]} \forall z : \texttt{a}[z \in ys \rightarrow z \notin \texttt{subList } xs\ ys]]$$

Base Case

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

(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: $\forall ys : \texttt{[a]} \forall z : \texttt{a}[z \in ys \rightarrow z \notin \texttt{subList } vs\ ys]$

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

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

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

Base Case

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

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

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 $P(\texttt{[]}) \rightarrow (P(\texttt{x:[]}) \rightarrow (P(\texttt{y:x:[]}) \rightarrow P(\texttt{z:y:x[]})))$, 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`

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

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

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

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

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

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;

```
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: `sum` is = `sum_tr` is 0

To show: `sum` $(i : is)$ = `sum_tr` $(i : is)$ 0

`sum` $(i : is)$	$= i +$ `sum` is	by def. of `sum`
	$= i +$ `sum_tr` is 0	by inductive hypothesis
	$=$???	???
	$=$ `sum_tr` is i	???
	$=$ `sum_tr` $(i : is)$ 0	by def. of `sum_tr`

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; $\forall i$: `Int`$\forall k$: `Int`$\forall is$: `[Int]`$[i +$ (`sum_tr` is $k) =$ `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 $\forall k$: `Int`$\forall is$: `[Int]`$[k +$ (`sum` is) = `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 $\forall k$: `Int`$[k +$ (`sum []`) = `sum_tr []` $k]$

Take arbitrary k : `Int`

$k +$ (`sum []`)	$= k + 0$	by def. of `sum`
	$= k$	by arithmetic
	$=$ `sum_tr []` k	by def. of `sum_tr`

Inductive Step

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

Inductive hypothesis: $\forall m$: `Int`$[m +$ (`sum` is) = `sum_tr` is $m]$

To show: $\forall n$: `Int`$[k +$ (`sum` $(i : is)$) = `sum_tr` $(i : is)$ $n]$

Take arbitrary n : `Int`

$n +$ `sum` $(i : is)$	$= n + i +$ `sum` is	by def. of `sum`, and arithmetic
	$=$ `sum_tr` is $(n + i)$	by inductive hypothesis, and $m = n + i$
	$=$ `sum_tr` $(i : is)$ n	by def. of `sum_tr`

There are also induction principles for more complex cases;

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

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

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

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

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

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 $P(\text{Leaf}) \land \forall ts : \texttt{Trees}[Q(ts) \to P(\texttt{Node } ts)] \land Q(\texttt{Empty}) \land \forall t : \texttt{Tree} \forall ts : \texttt{Trees}[P(\texttt{t}) \land Q(\texttt{ts}) \to Q(\texttt{Cons } t\ ts)] \to \forall t : \texttt{Tree}[P(t)] \land \forall ts : \texttt{Trees}[Q(ts)]$.
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; $P(\texttt{Leaf}) \land \forall ts : \texttt{Trees}[\forall t : \texttt{Tree}, \forall ts_1, ts_2 : \texttt{Trees}[ts = ts_1 + +[t] + +ts_2 \to P(t)] \to P(\texttt{Node } ts)] \to \forall t : \texttt{Tree}[P(t)]$. 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. Generalizing 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 $n = 4$;

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

$= \texttt{M'}(4, 3, 8)$ by def. of M'

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

$= 16$ by def. of 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**: $\forall m, cnt, acc, p : \mathbb{N}[\texttt{M'}(m, cnt, acc) = p \rightarrow p = 2^{m-cnt} \cdot acc]$. 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 ($m < cnt \rightarrow m \neq cnt$, as the termination condition is $m = cnt$). Hence **Assrt_2** can be shown as **Assrt_2'** $= \forall m, cnt, acc, p : \mathbb{N}[m \geq cnt \rightarrow \texttt{M'}(m, cnt, acc) = 2^{m-cnt} \cdot acc]$. However, in order to be able tp prove **Assrt_2'**, we need to prove **Assrt_3**: $\forall k, m, cnt, acc : \mathbb{N}[k = m - cnt \rightarrow \texttt{M'}(m, cnt, acc) = 2^{m-cnt} \cdot acc]$. We also need to prove that **Assrt_3** $\rightarrow$ **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;

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

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

Base Case

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

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

(a1) $0 = m - cnt$ assumption, otherwise the proof is trivial

(1) $m = cnt$ by (a1), and arithmetic

$\texttt{M'}(m, cnt, acc) = acc$ by (1), and def. of M'

$= 2^0 \cdot acc$ by arithmetic

$= 2^{m-cnt} \cdot acc$ by (a1)

Inductive Step

Take arbitrary $m, cnt_1, acc : \mathbb{N}$

Inductive hypothesis: $k = m - cnt_1 \to \texttt{M'}(m, cnt, acc) = 2^{m-cnt_1} \cdot acc$

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

Take arbitrary $cnt : \mathbb{N}$

(a1) $k + 1 = m - cnt$ assumption, otherwise the proof is trivial

(1)