Reasoning about Programs

Jeremy Bradley and Francesca Toni

Room 372. Office hour - Tuesdays at noon. Email: jb@doc.ic.ac.uk

Department of Computing, Imperial College London

Produced with prosper and LaTeX

Haskell Lectures I

Proving correctness of Haskell functions

- Induction over natural numbers
  - summing natural numbers: `sumInts`
  - summing fractions: `sumFracs`
  - natural number sequence: `uList`
  - proving induction works
- Structural induction
- Induction over Haskell data structures
  - induction over lists: `subList`, `revList`
  - induction over user-defined structures: `evalBoolExpr`

Haskell Lectures II

Proving correctness of Haskell functions

- Failed induction: `nub`
- Tree sort example: `sortInts`, `flattenTree`, `insTree`

Induction Example

Given the following Haskell program:

```
sumInts :: Int -> Int
sumInts 1 = 1
sumInts n = n + (sumInts (n-1))
```

- There are constraints on its input i.e. on the variable `r` in the function call `sumInts r`
- What is its output?

$$\begin{aligned} \texttt{sumInts r} &= r + (r-1) + \cdots + 2 + 1 \\ &= \sum_{n=1}^{r} n \end{aligned}$$

sumInts: Example

- Input constraints are the *pre-conditions* of a function
- Output requirements are the *post-conditions* for a function
- Function should be rewritten with conditions:

```
-- Pre-condition: n >= 1
-- Post-condition: sumInts r = ?
sumInts :: Int -> Int
sumInts 1 = 1
sumInts n = n + (sumInts (n-1))
```


sumInts: Example

Variable and output

n	sumInts n
1	1
2	3
3	6
4	10
5	15
6	21
7	28
8	36
9	45
10	55

```
-- Pre-condition: n >= 1
-- Post-condition: sumInts r = ?
sumInts :: Int -> Int
sumInts 1 = 1
sumInts n = n + (sumInts (n-1))
```


sumInts: Example

- Let's guess that the post-condition for `sumInts` should be:

$$\texttt{sumInts n} = \frac{\texttt{n}}{2}(\texttt{n}+1)$$

- How do we prove our conjecture?
- We use *induction*

Induction in General

The structure of an *induction proof* always follows the same pattern:

- State the proposition being proved: e.g. $P(n)$
- Identify and prove the *base case*: e.g. show true at $n = 1$
- Identify and state the *induction hypothesis* as assumed e.g. assumed true for the case, $n = k$
- Prove the $n = k + 1$ case is true as long as the $n = k$ case is assumed true. This is the *induction step*

sumInts: Induction

1. Base case, $n = 1$:
   $\texttt{sumInts 1} = \frac{1}{2} \times 2 = 1$
2. Induction hypothesis, $n = k$: Assume
   $\texttt{sumInts k} = \frac{\texttt{k}}{2}(\texttt{k} + 1)$
3. Induction step, $n = k + 1$: Using assumption, we need to show that:
   $\texttt{sumInts }(\texttt{k} + 1) = \frac{\texttt{k+1}}{2}(\texttt{k} + 2)$

Trying to prove for all $n \geq 1$:

$$\texttt{sumInts n} = \frac{\texttt{n}}{2}(\texttt{n} + 1)$$

sumInts: Induction Step

- Need to keep in mind 3 things:
  - Definition: $\texttt{sumInts n} = \texttt{n} + (\texttt{sumInts }(\texttt{n} - 1))$
  - Induction assumption: $\texttt{sumInts k} = \frac{\texttt{k}}{2}(\texttt{k} + 1)$
  - Need to prove: $\texttt{sumInts }(\texttt{k} + 1) = \frac{\texttt{k+1}}{2}(\texttt{k} + 2)$

Case, $n = k + 1$:

$$\begin{aligned} \texttt{sumInts }(\texttt{k} + 1) &= (k + 1) + \texttt{sumInts k} \\ &= (k + 1) + \frac{k}{2}(k + 1) \\ &= (k + 1)(1 + \frac{k}{2}) \\ &= \frac{k + 1}{2}(k + 2) \quad \Box \end{aligned}$$

Induction Argument

An *infinite* argument:

- Base case: $P(1)$ is true
- Induction Step: $P(k) \Rightarrow P(k + 1)$ for all $k \geq 1$
  - $P(1) \Rightarrow P(2)$ is true
  - $P(2) \Rightarrow P(3)$ is true
  - $P(3) \Rightarrow P(4)$ is true
  - $\ldots$
- and so $P(n)$ is true for any $n \geq 1$

Example: sumFracs

- Given the following program:

```
-- Pre-condition: n >= 1
-- Post-condition: sumFracs n = n / (n + 1)
sumFracs :: Int -> Ratio Int
sumFracs 1 = 1 % 2
sumFracs n = (1 % (n * (n + 1)))
                   + (sumFracs (n - 1))
```

- Equivalent to asking:

$$\frac{1}{1 \times 2} + \frac{1}{2 \times 3} + \frac{1}{3 \times 4} + \cdots + \frac{1}{n(n + 1)} = \frac{n}{n + 1}$$

sumFracs: Induction

- Proving that post-condition holds:
  - Base case, $n = 1$: $\texttt{sumFracs 1} = \texttt{1/2}$ (i.e. post-condition true)
  - Assume, $n = k$: $\texttt{sumFracs k} = \texttt{k}/(\texttt{k}+\texttt{1})$
  - Induction step, $n = k + 1$:

$$\begin{aligned}
\texttt{sumFracs}\ (\texttt{k}+\texttt{1}) &= \frac{1}{(k+1)(k+2)} + \texttt{sumFracs k} \\
&= \frac{1}{(k+1)(k+2)} + \frac{k}{k+1} \\
&= \frac{k^2+2k+1}{(k+1)(k+2)} \\
&= \frac{(k+1)^2}{(k+1)(k+2)} \\
&= \frac{k+1}{k+2} \quad \square
\end{aligned}$$

Strong Induction

- Induction arguments can have:
  - an induction step which depends on more than one assumption
    - as long as the assumption cases are $<$ the induction step case
    - e.g. it may be that $P(k-5)$ and $P(k-3)$ and $P(k-2)$ have to be assumed true to show $P(k+1)$ true
    - this is called *strong induction* and occasionally *course-of-values induction*
  - several base conditions if needed
    - e.g. $P(1), P(2), \ldots, P(5)$ may all be base cases

Example: uList function

Given the following program:

```
uList :: Int -> Int
uList 1 = 1
uList 2 = 5
uList n = 5 * (uList (n-1))
        - 6 * (uList (n-2))
```

- Pre-condition: call $\texttt{uList r}$ with $r \geq 1$
- Post-condition: require $\texttt{uList r} = 3^{\texttt{r}} - 2^{\texttt{r}}$

Induction Example

In mathematical terms induction problem looks like:

- We define a sequence of integers, u_n, where $u_n = 5u_{n-1} - 6u_{n-2}$ for $n \geq 2$ and base cases $u_1 = 1$, $u_2 = 5$.
- We want to prove, by induction, that: $u_n = \texttt{uList n} = 3^{\texttt{n}} - 2^{\texttt{n}}$
- (Note that this time we have two base cases)

Proof by Induction

- Start with the *base cases*, $n = 1, 2$
  - $\texttt{uList 1} = 3^1 - 2^1 = 1$
  - $\texttt{uList 2} = 3^2 - 2^2 = 5$
- State *induction hypothesis* for $n = k$ (that you're assuming is true for the next step):
  - $\texttt{uList k} = 3^k - 2^k$

Proof by Induction

- Looking to prove: $\texttt{uList (k + 1)} = 3^{k+1} - 2^{k+1}$
- Prove *induction step* for $n = k + 1$ case, by using the induction hypothesis case:

$$
\begin{aligned}
\texttt{uList (k + 1)} &= 5 * \texttt{uList k} - 6 * \texttt{uList (k - 1)} \\
&= 5(3^k - 2^k) - 6(3^{k-1} - 2^{k-1}) \\
&= 5(3^k - 2^k) - 2 \times 3^k + 3 \times 2^k \\
&= 3 \times 3^k - 2 \times 2^k \\
&= 3^{k+1} - 2^{k+1} \quad \square
\end{aligned}
$$

- Note we had to use the hypothesis twice

Induction Argument

An *infinite* argument for induction based on natural numbers:

- Base case: $P(0)$ is true
- Induction Step: $P(k) \Rightarrow P(k + 1)$ for all $k \in \mathbb{N}$
  - $P(0) \Rightarrow P(1)$ is true
  - $P(1) \Rightarrow P(2)$ is true
  - $P(2) \Rightarrow P(3)$ is true
  - ...
- and so $P(n)$ is true for any $n \in \mathbb{N}$

(Note: Induction can start with any value base case that is appropriate for the property being proved. It does not have to be 0 or 1)

Proof by Contradiction

- We have a proposition $P(n)$ which we have proved by induction, i.e.
  - $P(0)$ is true
  - $P(k) \Rightarrow P(k + 1)$ for all $k \in \mathbb{N}$
- Taken this to mean $P(n)$ is true for all $n \in \mathbb{N}$
- Let's assume instead that despite using induction on $P(n)$, $P(n)$ is not true for all $n \in \mathbb{N}$
- If we can show that this assumption gives us a logical contradiction, then we will know that the assumption was false

Proof of Induction

- Proof relies on fact that:
  - the set of natural numbers $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ has a least element
  - also any subset of natural numbers has a least element: e.g. $\{8, 13, 87, 112\}$ or $\{15, 17, 21, 32\}$
  - and so the natural numbers are ordered. i.e. $<$ is defined for all pairs of natural numbers (e.g. $4 < 7$)

Proof of Induction

- Assume $P(n)$ is not true for all $n \in \mathbb{N}$

$\Rightarrow$ There must be largest subset of natural numbers, $S \subset \mathbb{N}$, for which $P(n)$ is not true. ($0 \notin S$)

$\Rightarrow$ The set S must have a least element $m > 0$, as it is a subset of the natural numbers

$\Rightarrow$ $P(m)$ is false, but $P(m-1)$ must be true otherwise $m-1$ would be least element of S

- However we have proved that $P(k) \Rightarrow P(k+1)$ for all $k \in \mathbb{N}$

$\Rightarrow$ $P(m-1) \Rightarrow P(m)$ is true. Contradiction!

Induction in General

- In general we can perform induction across data structures (i.e. the same or similar proof works) if:
  1. the data structure has a least element or set of least elements
  2. an ordering exists between the elements of the data structure
- For example for a list:
  - $[\,]$ is the least element
  - $\mathtt{xs} < \mathtt{ys}$ if $\mathtt{length\ xs} < \mathtt{length\ ys}$

Induction over Data Structures

Given a conjecture $P(\mathtt{xs})$ to test:

- Induction on $[\mathtt{a}]$:
  - Base case: test true for $\mathtt{xs} = [\,]$
  - Assume true for $\mathtt{xs} = \mathtt{zs} :: [\mathtt{a}]$
  - Induction step: prove for $\mathtt{xs} = (\mathtt{z} : \mathtt{zs})$
- For structure MyList:

```
data MyList a = EmptyList | Cons a (MyList a)
```

  - Base case: test true for $\mathtt{xs} = \mathtt{EmptyList}$
  - Assume true for general $\mathtt{xs} = \mathtt{zs} :: \mathtt{MyList\ a}$
  - Induction step: prove for $\mathtt{xs} = \mathtt{Cons\ z\ zs}$ for any $\mathtt{z}$

Induction over Data Structures

Given a conjecture $P(\mathtt{xs})$ to test:

- For a binary tree:

```
data BTree a
        = BTempty
        | BTnode (BTree a) a (BTree a)
```

  - Base case: test true for $\mathtt{xs} = \mathtt{BTempty}$
  - Assume true for general cases: $\mathtt{xs} = \mathtt{t1} :: \mathtt{BTree\ a}$ and $\mathtt{xs} = \mathtt{t2} :: \mathtt{BTree\ a}$
  - Induction step: prove true for $\mathtt{xs} = \mathtt{BTnode\ t1\ z\ t2}$ for any $\mathtt{z}$

Structural Induction in General I

The structure of an *structural induction proof* always follows the same pattern:

- For generic data structure:

```
data DataS a
 = Rec1 (DataS a) | Rec2 (DataS a) (DataS a) |...
 | Base1 | Base2 | ...
```

  - State the proposition being proved: e.g. $P(\mathtt{xs} :: \mathtt{DataS\ a})$
  - Identify and prove the *base cases*: e.g. show $P(\mathtt{xs})$ true at $\mathtt{xs} = \mathtt{Base1}, \mathtt{Base2}, \ldots$

Structural Induction in General II

- For generic data structure:

```
data DataS a
 = Rec1 (DataS a) | Rec2 (DataS a) (DataS a) |...
 | Base1 | Base2 | ...
```

  - Identify and state the *induction hypothesis* as assumed e.g. assume $P(\mathtt{xs})$ true for all cases, $\mathtt{xs} = \mathtt{zs}$
  - Finally, assuming all the $\mathtt{xs} = \mathtt{zs}$ cases are true. Prove the *induction step* $P(\mathtt{xs})$ true for the cases $\mathtt{xs} = \mathtt{Rec1\ zs1}$, $\mathtt{xs} = \mathtt{Rec2\ zs1\ zs2}, \ldots$

Example: subList

- $\mathtt{subList\ xs\ ys}$ removes any element in $\mathtt{ys}$ from $\mathtt{xs}$

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

- $P(\mathtt{xs})$ = for any $\mathtt{ys}$, no elements of $\mathtt{ys}$ exist in $\mathtt{subList\ xs\ ys}$
- Is this a post-condition for $\mathtt{subList}$?

Induction: subList

- Base case, $\texttt{xs} = []$:
  - $P([])$ = for any $\texttt{ys}$, no elements of $\texttt{ys}$ exist in $(\texttt{subList}\ []\ \texttt{ys}) = []$. i.e. True.
- Assume case $\texttt{xs} = \texttt{zs}$:
  - $P(\texttt{zs})$ = for any $\texttt{ys}$, no elements of $\texttt{ys}$ exist in $(\texttt{subList zs ys})$
- Induction step, (require to prove) case $\texttt{xs} = (\texttt{z} : \texttt{zs})$:
  - $P(\texttt{z} : \texttt{zs})$ = for any $\texttt{ys}$, no elements of $\texttt{ys}$ exist in $(\texttt{subList}\ (\texttt{z} : \texttt{zs})\ \texttt{ys})$

Induction: subList

- Induction step, $\texttt{xs} = (\texttt{z} : \texttt{zs})$:
  - $P(\texttt{z} : \texttt{zs})$ = for any $\texttt{ys}$, no elements of $\texttt{ys}$ exist in $(\texttt{subList}\ (\texttt{z} : \texttt{zs})\ \texttt{ys})$

$$\texttt{subList}\ (\texttt{z} : \texttt{zs})\ \texttt{ys}$$

$$= \begin{cases} \texttt{subList zs ys} & : \text{ if } z \in \texttt{ys} \\ (\texttt{z} : \texttt{subList zs ys}) & : \text{ if } z \notin \texttt{ys} \end{cases}$$

$$P(\texttt{z} : \texttt{zs}) = \begin{cases} P(\texttt{zs}) & : \text{ if } z \in \texttt{ys} \\ (z \notin \texttt{ys})\ \texttt{AND}\ \texttt{P}(\texttt{zs}) & : \text{ if } z \notin \texttt{ys} \end{cases}$$

$\square$

Example: revList

- Given the following program:

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

- We want to prove the following property:
  - $P(\texttt{xs}) =$ for any $\texttt{ys}$:

$$\texttt{revList}\ (\texttt{xs}{+}{+}\texttt{ys}) = (\texttt{revList ys}){+}{+}(\texttt{revList xs})$$

Induction: revList

- Program:

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

- Base case, $\texttt{xs} = []$:
  - $P([]) =$ for any $\texttt{ys}$,

$$\begin{aligned} \texttt{revList}\ ([]{+}{+}\texttt{ys}) &= (\texttt{revList ys}) \\ &= (\texttt{revList ys}){+}{+}[] \\ &= (\texttt{revList ys}){+}{+}(\texttt{revList}\ []) \end{aligned}$$

Induction: revList

- Assume case, $\mathtt{xs} = \mathtt{zs}$:
  - $P(\mathtt{zs}) =$ for any $\mathtt{ys}$:
    $\mathtt{revList}\ (\mathtt{zs{+}{+}ys}) = (\mathtt{revList\ ys}){+}{+}(\mathtt{revList\ zs})$
- Induction step, $\mathtt{xs} = (\mathtt{z} : \mathtt{zs})$:
  - $P(\mathtt{z} : \mathtt{zs}) =$ for any $\mathtt{ys}$,

$$\begin{aligned}
&\mathtt{revList}\ ((\mathtt{z} : \mathtt{zs}){+}{+}\mathtt{ys}) \\
&= \mathtt{revList}\ (\mathtt{z} :\ (\mathtt{zs}{+}{+}\mathtt{ys})) \\
&= (\mathtt{revList}\ (\mathtt{zs}{+}{+}\mathtt{ys})){+}{+}[\mathtt{z}] \\
&= ((\mathtt{revList\ ys}){+}{+}(\mathtt{revList\ zs})){+}{+}[\mathtt{z}] \\
&= (\mathtt{revList\ ys}){+}{+}((\mathtt{revList\ zs}){+}{+}[\mathtt{z}]) \\
&= (\mathtt{revList\ ys}){+}{+}(\mathtt{revList}\ (\mathtt{z} : \mathtt{zs})) \quad \square
\end{aligned}$$

Example: BoolExpr

- Given the following representation of a Boolean expression:

```
data BoolExpr
     = BoolAnd BoolExpr BoolExpr
     | BoolOr BoolExpr BoolExpr
     | BoolNot BoolExpr
     | BoolTrue
     | BoolFalse
```


Example: BoolExpr

- The following function attempts to simplify a `BoolExpr`:

```
evalBoolExpr :: BoolExpr -> BoolExpr
evalBoolExpr BoolTrue = BoolTrue
evalBoolExpr BoolFalse = BoolFalse

evalBoolExpr (BoolAnd x y)
= (evalBoolExpr x) ‘boolAnd‘ (evalBoolExpr y)

evalBoolExpr (BoolOr x y)
= (evalBoolExpr x) ‘boolOr‘ (evalBoolExpr y)

evalBoolExpr (BoolNot x)
= boolNot (evalBoolExpr x)
```


Example: BoolExpr

- Definition of `boolNot`:

```
-- Pre-condition: input BoolTrue or BoolFalse
boolNot :: BoolExpr -> BoolExpr
boolNot BoolTrue = BoolFalse
boolNot BoolFalse = BoolTrue
boolNot _
  = error ("boolNot: input should be"
    ++ "BoolTrue or BoolFalse")
```

Example: BoolExpr

- Definitions of `boolAnd` and `boolOr`:

```
boolAnd :: BoolExpr -> BoolExpr -> BoolExpr
boolAnd x y
    | isBoolTrue x = y
    | otherwise = BoolFalse

boolOr :: BoolExpr -> BoolExpr -> BoolExpr
boolOr x y
    | isBoolTrue x = BoolTrue
    | otherwise = y

isBoolTrue :: BoolExpr -> Bool
isBoolTrue BoolTrue = True
isBoolTrue _ = False
```


Induction: BoolExpr

- Trying to prove statement:
  - For all $\mathtt{ex}$, $P(\mathtt{ex}) = (\mathtt{evalBoolExpr\ ex})$ evaluates to `BoolTrue` or `BoolFalse`
- Base cases: $\mathtt{ex} = \mathtt{BoolTrue}; \mathtt{ex} = \mathtt{BoolFalse}$:
  - $P(\mathtt{BoolTrue}) = (\mathtt{evalBoolExpr\ BoolTrue}) = \mathtt{BoolTrue}$
  - $P(\mathtt{BoolFalse}) = (\mathtt{evalBoolExpr\ BoolFalse}) = \mathtt{BoolFalse}$

Induction: BoolExpr

- Assume cases, $\mathtt{ex} = \mathtt{kx}, \mathtt{kx1}, \mathtt{kx2}$:
  - e.g. $P(\mathtt{kx}) = (\mathtt{evalBoolExpr\ kx})$ evaluates to `BoolTrue` or `BoolFalse`
- Three inductive steps:

1. Case $\mathtt{ex} = \mathtt{BoolNot\ kx}$

$$\begin{aligned} & P(\mathtt{BoolNot\ kx}) \\ = \ & (\mathtt{evalBoolExpr}\ (\mathtt{BoolNot\ kx})) \\ = \ & \mathtt{boolNot}\ (\mathtt{evalBoolExpr\ kx}) \\ = \ & \begin{cases} \mathtt{BoolFalse} & : \text{if } (\mathtt{evalBoolExpr\ kx}) = \mathtt{BoolTrue} \\ \mathtt{BoolTrue} & : \text{otherwise} \end{cases} \end{aligned}$$

Induction: BoolExpr

- Assume cases, $\mathtt{ex} = \mathtt{kx}, \mathtt{kx1}, \mathtt{kx2}$:
  - e.g. $P(\mathtt{kx1}) = (\mathtt{evalBoolExpr\ kx1})$ evaluates to `BoolTrue` or `BoolFalse`

---

2. Case $\mathtt{ex} = \mathtt{BoolAnd\ kx1\ kx2}$

$$\begin{aligned} & P(\mathtt{BoolAnd\ kx1\ kx2}) \\ = \ & (\mathtt{evalBoolExpr}\ (\mathtt{BoolAnd\ kx1\ kx2})) \\ = \ & (\mathtt{evalBoolExpr\ kx1})\ \text{`}\mathtt{boolAnd}\text{`}\ (\mathtt{evalBoolExpr\ kx2}) \\ = \ & \begin{cases} (\mathtt{evalBoolExpr\ kx2}) & : \text{if } (\mathtt{evalBoolExpr\ kx1}) \\ & \quad = \mathtt{BoolTrue} \\ \mathtt{BoolFalse} & : \text{otherwise} \end{cases} \end{aligned}$$

Induction: BoolExpr

- Assume cases, $\mathtt{ex} = \mathtt{kx}, \mathtt{kx1}, \mathtt{kx2}$:
  - e.g. $P(\mathtt{kx2}) = (\mathtt{evalBoolExpr\ kx2})$ evaluates to `BoolTrue` or `BoolFalse`

---

3. Case $\mathtt{ex} = \mathtt{BoolOr\ kx1\ kx2}$

$$
\begin{aligned}
& P(\mathtt{BoolOr\ kx1\ kx2}) \\
= \; & (\mathtt{evalBoolExpr}\ (\mathtt{BoolOr\ kx1\ kx2})) \\
= \; & (\mathtt{evalBoolExpr\ kx1})\ \text{`}\mathtt{boolOr}\text{`}\ (\mathtt{evalBoolExpr\ kx2}) \\
= \; & \begin{cases} \mathtt{BoolTrue} & : \text{if } (\mathtt{evalBoolExpr\ kx1}) = \mathtt{BoolTrue} \\ (\mathtt{evalBoolExpr\ kx2}) & : \text{otherwise} \end{cases}
\end{aligned}
$$

□

Example: nub

- What happens if you try to prove something that is not true?
- `nub` [from Haskell List library] removes duplicate elements from an arbitrary list

```
nub :: Eq a => [a] -> [a]
nub [] = []
nub (x:xs) = x : filter (x /=) (nub xs)
```

- We are going to attempt to prove:
  - For all lists, $\mathtt{xs}$, $P(\mathtt{xs})$ = for any $\mathtt{ys}$:

$$\mathtt{nub}\ (\mathtt{xs}{+}{+}\mathtt{ys}) = (\mathtt{nub\ xs}){+}{+}(\mathtt{nub\ ys})$$

Induction: nub

- False proposition:
  - For all lists, $\mathtt{xs}$, $P(\mathtt{xs})$ = for any $\mathtt{ys}$:

$$\mathtt{nub}\ (\mathtt{xs}{+}{+}\mathtt{ys}) = (\mathtt{nub\ xs}){+}{+}(\mathtt{nub\ ys})$$

- Base case, $\mathtt{xs} = [\,]$:

$$P([\,]) = \text{for any } \mathtt{ys},$$

$$
\begin{aligned}
& \mathtt{nub}\ ([\,]{+}{+}\mathtt{ys}) \\
= \; & \mathtt{nub\ ys} \\
= \; & [\,]{+}{+}(\mathtt{nub\ ys}) \\
= \; & (\mathtt{nub}\ [\,]){+}{+}(\mathtt{nub\ ys})
\end{aligned}
$$

Induction: nub

- Assume case, $\mathtt{xs} = \mathtt{ks}$:
  - $P(\mathtt{ks})$ = for any $\mathtt{ys}$, $\mathtt{nub}\ (\mathtt{ks}{+}{+}\mathtt{ys}) = (\mathtt{nub\ ks}){+}{+}(\mathtt{nub\ ys})$
- Inductive step, $\mathtt{xs} = (\mathtt{k} : \mathtt{ks})$:
  - $P(\mathtt{k} : \mathtt{ks})$ = for any $\mathtt{ys}$,

$$
\begin{aligned}
& \mathtt{nub}\ ((\mathtt{k} : \mathtt{ks}){+}{+}\mathtt{ys}) \\
= \; & \mathtt{nub}\ (\mathtt{k} : (\mathtt{ks}{+}{+}\mathtt{ys})) \\
= \; & \mathtt{k} : (\mathtt{filter}\ (\mathtt{k}\ /{=})\ (\mathtt{nub}\ (\mathtt{ks}{+}{+}\mathtt{ys}))) \\
= \; & \mathtt{k} : \mathtt{filter}\ (\mathtt{k}\ /{=})\ ((\mathtt{nub\ ks}){+}{+}(\mathtt{nub\ ys})) \\
= \; & (\mathtt{k} : \mathtt{filter}\ (\mathtt{k}\ /{=})\ (\mathtt{nub\ ks})){+}{+}(\mathtt{filter}\ (\mathtt{k}\ /{=})\ (\mathtt{nub\ ys})) \\
= \; & (\mathtt{nub}\ (\mathtt{k} : \mathtt{ks})){+}{+}(\mathtt{filter}\ (\mathtt{k}\ /{=})\ (\mathtt{nub\ ys}))
\end{aligned}
$$

Example: nub

- Review of failed induction:
  - Our proposition was: $P(\texttt{ks}) =$ for any $\texttt{ys}$,
    $\texttt{nub}\ (\texttt{ks++ys}) = (\texttt{nub ks})\texttt{++}(\texttt{nub ys})$
  - If true, we would expect the inductive step to give us: $P(\texttt{k} : \texttt{ks}) =$ for any $\texttt{ys}$,
    $\texttt{nub}\ ((\texttt{k} : \texttt{ks})\texttt{++ys}) = (\texttt{nub}\ (\texttt{k} : \texttt{ks}))\texttt{++}(\texttt{nub ys})$
  - In fact we actually got: $P(\texttt{k} : \texttt{ks}) =$ for any $\texttt{ys}$,
    $\texttt{nub}\ ((\texttt{k} : \texttt{ks})\texttt{++ys}) =$
    $(\texttt{nub}\ (\texttt{k} : \texttt{ks}))\texttt{++}(\texttt{filter}\ (\texttt{k}\ /=)\ (\texttt{nub ys}))$
- Hence the induction failed

Induction: Beware!

- Good news:
  - If you can prove a statement by induction – then it's true!
- Bad news!
  - If an induction proof fails – it's not necessarily false!
- i.e. induction proofs can fail because:
  - the statement is not true
  - induction is not an appropriate proof technique for a given problem

Fermat's Last Theorem

- Fermat stated and didn't prove that:

$$x^n + y^n = z^n$$

  had no positive integer solutions for $n \geq 3$
- Base case: it's been proved that $x^3 + y^3 = z^3$ has no solutions
- Assuming: $x^k + y^k = z^k$ has no solutions for $n \geq 3$
- There is no way of showing that $x^{k+1} + y^{k+1}$ does not have (only) $k + 1$ identical factors, from the assumption for the $n = k$ case

Induction over Data Structures

Given a conjecture $P(\texttt{xs})$ to test:

- For a binary tree:

```
data BTree a
        = BTempty
        | BTnode (BTree a) a (BTree a)
```

  - Base case: test true for $\texttt{xs} = \texttt{BTempty}$
  - Assume true for general cases: $\texttt{xs} = \texttt{t1} :: \texttt{BTree a}$ and $\texttt{xs} = \texttt{t2} :: \texttt{BTree a}$
  - Induction step: prove true for $\texttt{xs} = \texttt{BTnode t1 z t2}$ for any $\texttt{z}$

Induction in General

- In general we can perform induction across data structures (i.e. the same or similar proof works) if:
  1. the data structure has a least element or set of least elements
  2. an ordering exists between the elements of the data structure
- For example for a list:
  - $[\,]$ is the least element
  - $\mathtt{xs} < \mathtt{ys}$ if $\mathtt{length\,xs} < \mathtt{length\,ys}$

Well-founded Induction

- For this induction we need an ordering function $<$ for trees (as we already have for lists)
- $<$ is a well-founded relation on a set/datatype S if there is no infinite decreasing sequence. i.e. $t_1 < t_2 < t_3 < \cdots$ where t_1 is a minimal element
- For trees, $t1, t2 :: \mathtt{BTree\ a}$, $t1 < t2$ if $\mathtt{numBTelem\,t1} < \mathtt{numBTelem\,t2}$

```
numBTelem :: BTelem a -> Int
numBTelem BTempty = 0
numBTelem (BTnode lhs x rhs)
     = 1 + (numBTelem lhs) + (numBTelem rhs)
```


Example: Tree Sort

- We are going to sort a list of integers using the tree data structure:

```
data BTree a
  = BTempty
  | BTnode (BTree a) a (BTree a)
```

- and function, `sortInts`:

```
sortInts :: [Int] -> [Int]
sortInts xs = flattenTree ts where
    ts = foldr insTree BTempty xs
```


Example: Tree Sort

- `flattenTree` creates an inorder list of all elements of t

```
-- pre-condition: input tree is sorted
flattenTree :: BTree a -> [a]
flattenTree BTempty = []
flattenTree (BTnode lhs i rhs)
    = (flattenTree lhs) ++ [i]
      ++ (flattenTree rhs)
```

- inorder: = lhs ++ element ++ rhs
- preorder: = element ++ lhs ++ rhs
- postorder: = lhs ++ rhs ++ element

Example: Tree Sort

- `insTree` inserts an integer into the correct place in a sorted tree

```
-- pre-condition: input tree is pre-sorted,
--    i is arbitrary Int
-- post-condition: output is sorted tree
--    containing  all previous elements and i
insTree :: Int -> BTree Int -> BTree Int
insTree i BTempty = (BTnode BTempty i BTempty)
insTree i (BTnode t1 x t2)
    | i < x     = (BTnode (insTree i t1) x t2)
    | otherwise = (BTnode t1 x (insTree i t2))
```


Example: Tree Sort

- In order to show that sortInts does sort the integers – we need to show:
  - `flattenTree` does produce an inorder traversal of a tree
  - `insTree`
    - inserts the relevent element
    - keeps the tree sorted
    - does not modify any of the pre-existing elements

Induction: flattenTree

- Proposition: $P(\texttt{t}) = (\texttt{flattenTree t})$ creates inorder listing of all elements of `t`
- Base case, $\texttt{t} = \texttt{BTempty}$:

$$P(\texttt{BTempty}) = (\texttt{flattenTree BTempty}) = [\,]$$

- Assume cases, $\texttt{t} = \texttt{t1}$ and `t2`, e.g. : $P(\texttt{t1}) = (\texttt{flattenTree t1})$ creates inorder listing of all elements of `t1`

Induction: flattenTree

- Proposition: $P(\texttt{t}) = (\texttt{flattenTree t})$ creates inorder listing of all elements of `t`
- Inductive step, $\texttt{t} = \texttt{BTnode t1 i t2}$:

$$\begin{aligned} & P(\texttt{BTnode t1 i t2}) \\ = \; & (\texttt{flattenTree}\ (\texttt{BTnode t1 i t2})) \\ = \; & (\texttt{flattenTree t1})\texttt{++}[\texttt{i}]\texttt{++}(\texttt{flattenTree t2}) \end{aligned}$$

Induction: insTree

- We can split the proof of correctness of `insTree` into two inductions:
  1. keeps the tree sorted after the element is inserted
  2. inserts the relevent element and does not modify any of the pre-existing elements

Induction 1: insTree

- A tree $(\texttt{BTnode t1 x t2})$ is sorted if
  - `t1` and `t2` are sorted
  - all elements in `t1` are less than `x`
  - all elements in `t2` are greater than or equal to `x`
- Define induction hypothesis to be:

$$P(\texttt{t}) = \text{for any } \texttt{i}, (\texttt{insTree i t}) \text{ is sorted}$$

Induction 1: insTree

- Base case, $\texttt{t} = \texttt{BTempty}$:
  - $P(\texttt{BTempty}) =$ for any `i`,

$$\texttt{insTree i BTempty} = \texttt{BTnode BTempty i BTempty}$$

    is sorted
- Assume $P(\texttt{t})$ true for cases, $\texttt{BTempty} \leq \texttt{t} < \texttt{BTnode t1 i}' \texttt{ t2}$
  - e.g. $P(\texttt{t1}) =$ for any `i`,

$$(\texttt{insTree i t1}) \text{ is sorted}$$

Induction 1: insTree

- Induction step, case $\texttt{t} = \texttt{BTnode t1 i}' \texttt{ t2}$:
  - $P(\texttt{BTnode t1 i}' \texttt{ t2}) =$ for any `i`,

$$\texttt{insTree i } (\texttt{BTnode t1 i}' \texttt{ t2})$$
$$= \begin{cases} \texttt{BTnode } (\texttt{insTree i t1}) \texttt{ i}' \texttt{ t2} & : \text{if } \texttt{i} < \texttt{i}' \\ \texttt{BTnode t1 i}' \ (\texttt{insTree i t2}) & : \text{otherwise} \end{cases}$$

  - By our assumptions, we know that `t1`, `t2`, $(\texttt{insTree i t1})$, $(\texttt{insTree i t2})$ are sorted

Induction 2: insTree

- $Q(\mathtt{t})$ = there exist some $\mathtt{ms}$, $\mathtt{ns}$ such that:
  - $(\mathtt{ms{+}{+}[i]{+}{+}ns}) = (\mathtt{flattenTree}\ (\mathtt{insTree\ i\ t}))$
  - $(\mathtt{flattenTree\ t}) = (\mathtt{ms{+}{+}ns})$
- Base case, $\mathtt{t} = \mathtt{BTempty}$:
  - $Q(\mathtt{BTempty})$ = there exist some $\mathtt{ms}, \mathtt{ns}$ such that:

$$
\begin{aligned}
&(\mathtt{ms{+}{+}[i]{+}{+}ns})\\
&\quad = (\mathtt{flattenTree}\ (\mathtt{insTree\ i\ BTempty}))\\
&\quad = \mathtt{flattenTree}\ (\mathtt{BTnode\ BTempty\ i\ BTempty})\\
&\quad = (\mathtt{flattenTree\ BTempty}){+}{+}[\mathtt{i}]{+}{+}(\mathtt{flattenTree\ BTempty})\\
&\quad = [\,]{+}{+}[\mathtt{i}]{+}{+}[\,]
\end{aligned}
$$

  - i.e. $\mathtt{ms} = \mathtt{ns} = [\,]$
  - $(\mathtt{flattenTree\ BTempty}) = [\,] = (\mathtt{ms{+}{+}ns})$

Induction 2: insTree

- $Q(\mathtt{t})$ = there exist some $\mathtt{ms}$, $\mathtt{ns}$ such that:
  - $(\mathtt{ms{+}{+}[i]{+}{+}ns}) = (\mathtt{flattenTree}\ (\mathtt{insTree\ i\ t}))$
  - $(\mathtt{flattenTree\ t}) = (\mathtt{ms{+}{+}ns})$

- Assume cases, $\mathtt{t} = \mathtt{t1}, \mathtt{t2}$:
  - $Q(\mathtt{t1})$ = there exist some $\mathtt{ms1}, \mathtt{ns1}$ such that:
    - $(\mathtt{ms1{+}{+}[i]{+}{+}ns1}) = (\mathtt{flattenTree}\ (\mathtt{insTree\ i\ t1}))$
    - $(\mathtt{flattenTree\ t1}) = (\mathtt{ms1{+}{+}ns1})$
  - $Q(\mathtt{t2})$ = there exist some $\mathtt{ms2}, \mathtt{ns2}$ such that:
    - $(\mathtt{ms2{+}{+}[i]{+}{+}ns2}) = (\mathtt{flattenTree}\ (\mathtt{insTree\ i\ t2}))$
    - $(\mathtt{flattenTree\ t2}) = (\mathtt{ms2{+}{+}ns2})$

Induction 2: insTree

- (Part 1) Case $\mathtt{t} = \mathtt{BTnode\ t1\ i'\ t2}$:
  - $Q(\mathtt{BTnode\ t1\ i'\ t2})$ = there exist some $\mathtt{ms}, \mathtt{ns}$ such that:
    - if $i < i'$:

$$
\begin{aligned}
&(\mathtt{ms{+}{+}[i]{+}{+}ns})\\
&\quad = (\mathtt{flattenTree}\ (\mathtt{insTree\ i}\ (\mathtt{BTnode\ t1\ i'\ t2})))\\
&\quad = \mathtt{flattenTree}\ (\mathtt{BTnode}\ (\mathtt{insTree\ i\ t1})\ \mathtt{i'\ t2})\\
&\quad = (\mathtt{flattenTree}\ (\mathtt{insTree\ i\ t1})){+}{+}[\mathtt{i'}]{+}{+}(\mathtt{flattenTree\ t2})
\end{aligned}
$$

    - i.e. $\mathtt{ms} = \mathtt{ms1}$ and $\mathtt{ns} = \mathtt{ns1{+}{+}[i']{+}{+}ms2{+}{+}ns2}$

$$
\begin{aligned}
&\mathtt{flattenTree}\ (\mathtt{BTnode\ t1\ i'\ t2})\\
&\quad = (\mathtt{flattenTree\ t1}){+}{+}[\mathtt{i'}]{+}{+}(\mathtt{flattenTree\ t2})\\
&\quad = \mathtt{ms1{+}{+}ns1{+}{+}[i']{+}{+}ms2{+}{+}ns2}\\
&\quad = \mathtt{ms{+}{+}ns}
\end{aligned}
$$

Induction 2: insTree

- (Part 2) Case $\mathtt{t} = \mathtt{BTnode\ t1\ i'\ t2}$:
  - $Q(\mathtt{BTnode\ t1\ i'\ t2})$ = there exist some $\mathtt{ms}, \mathtt{ns}$ such that:
    - if $i \geq i'$:

$$
\begin{aligned}
&(\mathtt{ms{+}{+}[i]{+}{+}ns})\\
&\quad = (\mathtt{flattenTree}\ (\mathtt{insTree\ i}\ (\mathtt{BTnode\ t1\ i'\ t2})))\\
&\quad = \mathtt{flattenTree}\ (\mathtt{BTnode\ t1\ i'}\ (\mathtt{insTree\ i\ t2}))\\
&\quad = (\mathtt{flattenTree\ t1}){+}{+}[\mathtt{i'}]{+}{+}(\mathtt{flattenTree}\ (\mathtt{insTree\ i\ t2}))
\end{aligned}
$$

    - i.e. $\mathtt{ms} = \mathtt{ms1{+}{+}ns1{+}{+}[i']{+}{+}ms2}$ and $\mathtt{ns} = \mathtt{ns2}$

$$
\begin{aligned}
&\mathtt{flattenTree}\ (\mathtt{BTnode\ t1\ i'\ t2})\\
&\quad = (\mathtt{flattenTree\ t1}){+}{+}[\mathtt{i'}]{+}{+}(\mathtt{flattenTree\ t2})\\
&\quad = \mathtt{ms1{+}{+}ns1{+}{+}[i']{+}{+}ms2{+}{+}ns2}\\
&\quad = \mathtt{ms{+}{+}ns} \quad \square
\end{aligned}
$$