# Functional Programming

## Lecture 8: Reasoning and calculating

Twan van Laarhoven

8 November 2021

# Outline

- Equational reasoning
- Proof by induction
- Program synthesis
- Fusion
- Summary

# Equational reasoning

# Reasoning about programs

- Functional programs are just equations
  - equations as definitions, intended for evaluation
  - but also useful for reasoning: proofs
- Better than testing, because exhaustive
- No side-effects mean that rules of ordinary algebra apply:
  - Substitution of equals for equals
  - If $x = y$ then $y = x$

# Equational reasoning (I)

Given

```
not :: Bool → Bool
not False = True
not True  = False
```

We can prove that: `not (not False) = False`

Proof:

```
  not (not False)
= { definition of not (equation 1) }
  not True
= { definition of not (equation 2) }
  False
```

# Equational reasoning (II)

Prove that: `curry fst x y = const x y`

Proof:

```
  curry fst x y
= { definition of curry }
  fst (x,y)
= { definition of fst }
  x
= { definition of const }
  const x y
```

Where

```
curry f a b = f (a,b)
fst (a,b) = a
const a b = a
```

```haskell
reverse :: [a] → [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]

(++) :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

# Equational reasoning (III)

Prove that: `reverse [x] = [x]`

Proof:

```haskell
  reverse [x]
= { list notation }
  reverse (x:[])
= { definition of reverse (equation 2) }
  reverse [] ++ [x]
= { definition of reverse (equation 1) }
  [] ++ [x]
= { definition of ++ (equation 1) }
  [x]
```

# Proof by induction

# Programming and proving

Programs over recursive types typically require *recursion*

explicit recursion (ad hoc)

```
sum []     = 0
sum (x:xs) = x + sum xs
```

or “canned” recursion

```
sum = foldr (+) 0
```

uses predefined program schemes (higher-order functions)

# Programming and proving

Proofs about recursive programs typically require *induction*

explicit induction (ad hoc)

$$
\begin{aligned}
&\mathsf{proof}\ \mathsf{P}([\,]) &&= \text{—— ——}\\
&\mathsf{proof}\ \mathsf{P}(\mathsf{x}:\mathsf{xs}) &&= \text{——}\ \textit{use}\ P(xs)\ \text{——}
\end{aligned}
$$

or "canned" induction

$$\mathsf{proof}\ \mathsf{P}(\mathsf{xs}) = \text{——}\ \textit{use property of foldr}\ \text{——}$$

uses properties of predefined program schemes (higher-order proofs)

# Proof by induction for lists

Every recursive datatype comes with a *pattern of induction*

- To prove: for any list `xs`, property $P(\texttt{xs})$ holds

# Proof by induction for lists

Every recursive datatype comes with a *pattern of induction*

- To prove: for any list $\mathtt{xs}$, property $\mathsf{P}(\mathtt{xs})$ holds
- Then the proof has two parts:
    1. Prove that $P(\mathtt{[]})$ holds *(base case)*

# Proof by induction for lists

Every recursive datatype comes with a *pattern of induction*

- To prove: for any list `xs`, property P(`xs`) holds
- Then the proof has two parts:
  1. Prove that $P$(`[]`) holds *(base case)*
  2. Prove that $P$(`x:xs`) holds, given that $P$(`xs`) holds *(induction step)*

# Proof by induction for lists

Every recursive datatype comes with a *pattern of induction*

- To prove: for any list `xs`, property P(`xs`) holds
- Then the proof has two parts:
  1. Prove that $P$(`[]`) holds *(base case)*
  2. Prove that $P$(`x:xs`) holds, given that $P$(`xs`) holds *(induction step)*
- Part 2 uses the *induction hypothesis*: that $P$ holds for `xs`

# Proof by induction for lists

Every recursive datatype comes with a *pattern of induction*

- To prove: for any list `xs`, property P(`xs`) holds
- Then the proof has two parts:
  1. Prove that $P$(`[]`) holds *(base case)*
  2. Prove that $P$(`x:xs`) holds, given that $P$(`xs`) holds *(induction step)*
- Part 2 uses the *induction hypothesis*: that $P$ holds for `xs`
- Induction is valid for the same reason that recursive programs are valid: every list is either `[]` or of the form (`x:xs`)

# Example: Lists form a monoid

Recall definition

```
(++)     :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Monoid laws

1. $[\,] \mathbin{+\!\!+} ys = ys$
2. $xs \mathbin{+\!\!+} [\,] = xs$
3. $(xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs = xs \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs)$

Law 1: $[\,] \mathbin{+\!\!+} ys = ys$ holds by definition of $(\mathbin{+\!\!+})$ (first equation)


Radboud University

$$
\begin{array}{lcl}
(+\!\!+) & :: & [a] \to [a] \to [a] \\
[] & +\!\!+\ ys = & ys \\
(x:xs) & +\!\!+\ ys = & x : (xs +\!\!+ ys)
\end{array}
$$

# Example: Lists form a monoid

To prove law 2: $xs +\!\!+ [] = xs$

Use induction over $xs$

- Case $xs = []$ *(base case)*:

$$
\begin{array}{l}
[] +\!\!+ [] \\
= \{\ \text{definition of } +\!\!+\ \} \\
[]
\end{array}
$$

- Case $xs = a:as$ (inductive step), with Induction Hypothesis: $as +\!\!+ [] = []$

$$
\begin{array}{l}
(a:as) +\!\!+ [] \\
= \{\ \text{definition of } +\!\!+\ \} \\
a:(as +\!\!+ []) \\
= \{\ \text{IH}\ \} \\
a:as
\end{array}
$$

$$\begin{array}{l}(+\!\!+) :: [a] \to [a] \to [a] \\ [] \quad +\!\!+ \; ys = ys \\ (x:xs) +\!\!+ \; ys = x : (xs +\!\!+ ys)\end{array}$$

# Example: Lists form a monoid

To prove law 3: $(xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)$

Three variables, which one to choose for applying induction?
Heuristic: $+\!\!+$ does pattern matching on first argument.
So do induction on $xs$.

- Case $xs = []$ *(base case)*:

$$\begin{array}{l}([] +\!\!+ ys) +\!\!+ zs \\ = \{ \text{ definition of } +\!\!+ \} \\ ys +\!\!+ zs \\ = \{ \text{ definition of } +\!\!+ \} \\ [] +\!\!+ (ys +\!\!+ zs)\end{array}$$

$$
\begin{aligned}
(+\!\!+) &:: [a] \rightarrow [a] \rightarrow [a] \\
[\,] \;+\!\!+\; ys &= ys \\
(x:xs) \;+\!\!+\; ys &= x : (xs \;+\!\!+\; ys)
\end{aligned}
$$

# Example: Lists form a monoid

To prove law 3: $(xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)$

- Case $xs = a{:}as$ *(inductive step)*

  Assuming IH: $(as +\!\!+ ys) +\!\!+ zs = as +\!\!+ (ys +\!\!+ zs)$

  $((a : as) +\!\!+ ys) +\!\!+ zs$

  $=$

  $(a : as) +\!\!+ (ys +\!\!+ zs)$

$$
\begin{aligned}
(++) &:: [a] \to [a] \to [a] \\
[] \; ++ \; ys &= ys \\
(x{:}xs) \; ++ \; ys &= x : (xs \; ++ \; ys)
\end{aligned}
$$

# Example: Lists form a monoid

To prove law 3: $(xs ++ ys) ++ zs = xs ++ (ys ++ zs)$

- Case $xs = a{:}as$ *(inductive step)*

  Assuming IH: $(as ++ ys) ++ zs = as ++ (ys ++ zs)$

$$
\begin{aligned}
& ((a : as) ++ ys) ++ zs \\
= & \; \{ \text{ definition of } ++ \; \} \\
& (a : (as ++ ys)) ++ zs \\
= & \; \{ \text{ definition of } ++ \; \} \\
& a : ((as ++ ys) ++ zs) \\
& \\
& (a : as) ++ (ys ++ zs)
\end{aligned}
$$

## Example: Lists form a monoid

$$
\begin{array}{l}
(+\!\!+) :: [a] \to [a] \to [a] \\
[\,] \quad +\!\!+ \; ys = ys \\
(x{:}xs) \; +\!\!+ \; ys = x : (xs \; +\!\!+ \; ys)
\end{array}
$$

To prove law 3: $(xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)$

- Case $xs = a{:}as$ *(inductive step)*

  Assuming IH: $(as +\!\!+ ys) +\!\!+ zs = as +\!\!+ (ys +\!\!+ zs)$

$$
\begin{array}{l}
((a : as) +\!\!+ ys) +\!\!+ zs \\
= \{ \text{ definition of } +\!\!+ \} \\
(a : (as +\!\!+ ys)) +\!\!+ zs \\
= \{ \text{ definition of } +\!\!+ \} \\
a : ((as +\!\!+ ys) +\!\!+ zs) \\
\\
a : (as +\!\!+ (ys +\!\!+ zs)) \\
= \{ \text{ definition of } +\!\!+ \} \\
(a : as) +\!\!+ (ys +\!\!+ zs)
\end{array}
$$

$$
\begin{aligned}
(+\!\!+) &:: [a] \to [a] \to [a] \\
[\,] \mathbin{+\!\!+} ys &= ys \\
(x:xs) \mathbin{+\!\!+} ys &= x : (xs \mathbin{+\!\!+} ys)
\end{aligned}
$$

# Example: Lists form a monoid

To prove law 3: $(xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs = xs \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs)$

- Case $xs = a{:}as$ *(inductive step)*

  Assuming IH: $(as \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs = as \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs)$

$$
\begin{aligned}
&((a : as) \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs \\
=&\ \{\ \text{definition of } +\!\!+\ \} \\
&(a : (as \mathbin{+\!\!+} ys)) \mathbin{+\!\!+} zs \\
=&\ \{\ \text{definition of } +\!\!+\ \} \\
&a : ((as \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs) \\
=&\ \{\ \text{IH}\ \} \\
&a : (as \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs)) \\
=&\ \{\ \text{definition of } +\!\!+\ \} \\
&(a : as) \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs)
\end{aligned}
$$

$$\begin{aligned}
&\texttt{reverse}\ [\,] = [\,]\\
&\texttt{reverse}\ (x:xs) = \texttt{reverse}\ xs \mathbin{+\!\!+} [x]\\
&[\,] \mathbin{+\!\!+} ys = ys\\
&(x:xs) \mathbin{+\!\!+} ys = x:(xs \mathbin{+\!\!+} ys)
\end{aligned}$$

# Example: reverse-append

Prove that for all $xs$, $ys$:

$$\texttt{reverse}\ (xs \mathbin{+\!\!+} ys) = \texttt{reverse}\ ys \mathbin{+\!\!+} \texttt{reverse}\ xs.$$

Proof by induction on $xs$

- Case: $xs = [\,]$ *(base case)*:

$$\begin{aligned}
&\texttt{reverse}\ ([\,] \mathbin{+\!\!+} ys)\\
&= \{\ \texttt{definition of} \mathbin{+\!\!+}\ \}\\
&\texttt{reverse}\ ys\\
&= \{\ \texttt{unitality} \mathbin{+\!\!+}\ \}\\
&\texttt{reverse}\ ys \mathbin{+\!\!+} [\,]\\
&= \{\ \texttt{definition of reverse}\ \}\\
&\texttt{reverse}\ ys \mathbin{+\!\!+} \texttt{reverse}\ [\,]
\end{aligned}$$

$$
\begin{aligned}
&\texttt{reverse}\ [\,] = [\,]\\
&\texttt{reverse}\ (x{:}xs) = \texttt{reverse}\ xs \mathbin{+\!\!+} [x]\\
&[\,] \mathbin{+\!\!+} ys = ys\\
&(x{:}xs) \mathbin{+\!\!+} ys = x{:}(xs \mathbin{+\!\!+} ys)
\end{aligned}
$$

# Example: reverse-append

- Case: $xs = a{:}as$ *(inductive step)*
  using IH: $\texttt{reverse}\ (as \mathbin{+\!\!+} ys) = \texttt{reverse}\ ys \mathbin{+\!\!+} \texttt{reverse}\ as$

$$
\begin{aligned}
&\texttt{reverse}\ ((a{:}as) \mathbin{+\!\!+} ys)\\
&= \{\ \text{definition of } \mathbin{+\!\!+}\ \}\\
&\texttt{reverse}\ (a{:}(as \mathbin{+\!\!+} ys))\\
&= \{\ \text{definition of reverse}\ \}\\
&\texttt{reverse}\ (as \mathbin{+\!\!+} ys) \mathbin{+\!\!+} [a]\\
&= \{\ \text{IH}\ \}\\
&(\texttt{reverse}\ ys \mathbin{+\!\!+} \texttt{reverse}\ as) \mathbin{+\!\!+} [a]\\
&= \{\ \text{associativity of } \mathbin{+\!\!+}\ \}\\
&\texttt{reverse}\ ys \mathbin{+\!\!+} (\texttt{reverse}\ as \mathbin{+\!\!+} [a])\\
&= \{\ \text{definition of reverse}\ \}\\
&\texttt{reverse}\ ys \mathbin{+\!\!+} \texttt{reverse}\ (a{:}as)
\end{aligned}
$$

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

# Equality of functions

Suppose we want to prove: `reverse . reverse = id`

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

# Equality of functions

Suppose we want to prove: `reverse . reverse = id`

There is no variable to do induction on

```
reverse  []     = []
reverse  (x:xs) = reverse xs ++ [x]
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

# Equality of functions

Suppose we want to prove: `reverse . reverse = id`

There is no variable to do induction on

Add missing arguments

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

# Equality of functions

Suppose we want to prove: `reverse . reverse = id`

There is no variable to do induction on

Add missing arguments

Hence we have to prove that `(reverse . reverse) xs = id xs`

which is the same as `reverse (reverse xs) = xs`

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

# Equality of functions

Suppose we want to prove: `reverse . reverse = id`

There is no variable to do induction on

Add missing arguments

Hence we have to prove that `(reverse . reverse) xs = id xs`

which is the same as `reverse (reverse xs) = xs`

Proof by induction on `xs`

- Case `xs = []` *(base case)*:

```
 reverse (reverse [])
= { definition of reverse }
 reverse []
= { definition of reverse }
 []
```

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

# Equality of functions

- Case xs = a : as *(inductive step)*,
  Assuming IH: reverse (reverse as) = as

```
reverse (reverse (a:as))
= { definition of reverse }
reverse (reverse as ++ [a])
= { property reverse—append }
reverse [a] ++ reverse (reverse as)
= { IH }
reverse [a] ++ as
= { property reverse—singleton }
[a] ++ as
= { list notation }
(a:[]) ++ as
= { definition of ++ }
a: ([] ++ as)
= { definition of ++ }
a:as
```

```
reverse  [] = []
reverse (x: xs) = reverse xs ++ [x]
[]      ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

## Program synthesis

`reverse` is slow because ++ traverses its first argument. To eliminate ++ specify

```
reverseCat xs ys = reverse xs ++ ys
```

Use induction to synthesize an efficient implementation of `reverseCat`

- Case xs = [] *(base case)*:

```
reverseCat [] ys
= { specification of reverseCat }
reverse [] ++ ys
= { definition of reverse }
[] ++ ys
= { definition of ++ }
ys
```

```
reverse  []  = []
reverse  (x: xs) = reverse xs ++ [x]
[]       ++ ys = ys
(x:xs)   ++ ys = x:(xs ++ ys)
```

## Program synthesis

- Case xs = a:as *(inductive step)*,
  Assuming IH: for all lists es: `reverseCat as es = reverse as ++ es`

```
reverseCat (a:as) ys
= { specification of reverseCat }
reverse (a:as) ++ ys
= { definition of reverse }
(reverse as ++ [a]) ++ ys
= { associativity of ++ }
reverse as ++ ([a] ++ ys)
= { list notation, definition of ++ }
reverse as ++ (a:ys)
= { IH (right-to-left substituting a:ys for es) }
reverseCat as (a:ys)
```

```
reverse [] = []
reverse (x: xs) = reverse xs ++ [x]
[]       ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

# Program synthesis

We obtained:

```
reverseCat []     ys = ys
reverseCat (x:xs) ys = reverseCat xs (x:ys)
```

Now we can write:

```
reverse xs = reverseCat xs []
```

# Properties of program schemes

# Program schemes and their properties

- Use of program schemes (higher-order functions) improves modularity of *programs*
- Applies not only to programming but also to proving
- Use of properties of program schemes improves modularity of *proofs*, and hence understanding, modification, and reuse
- Example: map preserves identity and composition

```
map id      = id
map (f . g) = map f . map g
```

- Example: polymorphism of concat

```
map f . concat = concat . map (map f)
```

# foldr fusion

$$
\begin{aligned}
&\mathsf{foldr} :: (a \to b \to b) \to b \to [a] \to b \\
&\mathsf{foldr}\ (\otimes)\ z\ [\,] = z \\
&\mathsf{foldr}\ (\otimes)\ z\ (x : xs) = x \otimes \mathsf{foldr}\ z\ xs
\end{aligned}
$$

Example:

$$
\begin{aligned}
\mathsf{list} &= 1 : (2 : (3 : (4 : (5 : [\,])))) \\
\mathsf{foldr} \otimes z\ \mathsf{list} &= 1 \otimes (2 \otimes (3 \otimes (4 \otimes (5 \otimes z))))
\end{aligned}
$$

# foldr fusion

Example:

$$\begin{aligned}
\mathtt{list} &= 1 : (2 : (3 : (4 : (5 : [])))) \\
\mathtt{foldr} \otimes z\ \mathtt{list} &= 1 \otimes (2 \otimes (3 \otimes (4 \otimes (5 \otimes z))))
\end{aligned}$$

Suppose $f\ (x \otimes y) = x \boxplus f\ y$ for all x, y

$$\begin{aligned}
f\ (\mathtt{foldr}\ (\otimes)\ z\ \mathtt{list}) &= f\ (1 \otimes (2 \otimes (3 \otimes (4 \otimes (5 \otimes z))))) \\
&= 1 \boxplus f\ (2 \otimes (3 \otimes (4 \otimes (5 \otimes z)))) \\
&= 1 \boxplus (2 \boxplus f\ (3 \otimes (4 \otimes (5 \otimes z)))) \\
&= 1 \boxplus (2 \boxplus (3 \boxplus f\ (4 \otimes (5 \otimes z)))) \\
&\vdots \\
&= 1 \boxplus (2 \boxplus (3 \boxplus f\ (4 \boxplus (5 \boxplus f\ z)))) \\
&= \mathtt{foldr}\ (\boxplus)\ (f\ z)\ \mathtt{list}
\end{aligned}$$

# foldr fusion law

Captured by the foldr fusion law:

$\mathsf{f}\ (\mathsf{foldr}\ (\otimes)\ \mathsf{z}\ \mathsf{xs}) = \mathsf{foldr}\ (\boxplus)\ (\mathsf{f}\ \mathsf{z})\ \mathsf{xs}$ if for all x y: $\mathsf{f}\ (\mathsf{x} \otimes \mathsf{y}) = \mathsf{x} \boxplus \mathsf{f}\ \mathsf{y}$

Example: Suppose we want to prove that

$$\mathsf{sum}\ \mathsf{xs} + \mathsf{a} = \mathsf{foldr}\ (+)\ 0\ \mathsf{xs} + \mathsf{a} = \mathsf{foldr}\ (+)\ \mathsf{a}\ \mathsf{xs}$$

What are f, $(\otimes)$, $(\boxplus)$?

$$\begin{aligned} \mathsf{f} &= (+\ \mathsf{a}) \\ (\otimes) &= (+) \\ (\boxplus) &= (+) \end{aligned}$$

# Fold fusion example

$\boxed{\texttt{f . foldr g z = foldr h (f z) if } \forall \texttt{ x y: f (g x y) = h x (f y)}}$

We want to prove: `reverse . reverse = id`

First, we write reverse and id as folds

```
reverse = foldr (\a r → r ++ [a]) []
id      = foldr (:) []
```

Filling in the FF-law:

```
reverse . foldr (\a r → r ++ [a]) [] = foldr (:) []
```

Hence

```
f = reverse
g = \a r → r ++ [a]
h = (:)
```

# Fold fusion example

$\boxed{\texttt{f . foldr g z} = \texttt{foldr h (f z)} \text{ if } \forall\, x\, y\text{: } \texttt{f (g x y)} = \texttt{h x (f y)}}$

We want to prove: `reverse . reverse = id`

Filling in the FF-law:

```
reverse . foldr (\a r → r ++ [a]) [] = foldr (:) []
```

Hence

```
f = reverse
g = \a r → r ++ [a]
h = (:)
```

To complete proof it suffices to show that

```
1. []            = f []
2. f (g x y)     = h x (f y)
```

# Fold fusion example

$\boxed{\texttt{f . foldr g z = foldr h (f z) if } \forall \texttt{ x y: f (g x y) = h x (f y)}}$

We want to prove: `reverse . reverse = id`

Filling in the FF-law:

```
reverse . foldr (\a r → r ++ [a]) [] = foldr (:) []
```

Hence

```
f = reverse
g = \a r → r ++ [a]
h = (:)
```

To complete proof it suffices to show that

```
1. []                 = reverse []
2. reverse (y ++ [x]) = (:) x (reverse y)
```

# Fold fusion example

$\boxed{\texttt{f . foldr g z} = \texttt{foldr h (f z)} \text{ if } \forall\, x\, y\text{: } \texttt{f (g x y)} = \texttt{h x (f y)}}$

We want to prove: `reverse . reverse = id`

Filling in the FF-law:

```
reverse . foldr (\a r → r ++ [a]) [] = foldr (:) []
```

Hence

```
f = reverse
g = \a r → r ++ [a]
h = (:)
```

To complete proof it suffices to show that

```
1. []                = reverse []
2. reverse (y ++ [x]) = x : (reverse y)
```

# Fold fusion example

$\boxed{\texttt{f . foldr g z} = \texttt{foldr h (f z)} \text{ if } \forall \texttt{ x y: f (g x y)} = \texttt{h x (f y)}}$

1. [] = reverse [] { definition of reverse }

2. reverse (y ++ [x])
   = { property reverse—append }
   reverse [x] ++ reverse y
   = { property reverse—singleton }
   [x] ++ reverse y
   = { list notation, definition of ++ }
   x : reverse y

# Induction on natural numbers

# Induction on natural numbers

Prove that for all $n \geq 0$, property $P(n)$ holds

Proof by induction on $n$, two cases:

- Base case: prove $P(0)$
- Inductive step: assume that $P(n)$ holds, prove $P(n+1)$


Radboud University

# take n xs ++ drop n xs = xs

```
take :: Int → [a] → [a]
take 0 _      = []
take _ []     = []
take n (x:xs) = x : take (n−1) xs

drop :: Int → [a] → [a]
drop 0 xs     = xs
drop _ []     = []
drop n (x:xs) = drop (n−1) xs
```

Prove that $\forall\ n \geq 0,\ xs :: [a]$: take n xs ++ drop n xs = xs

```
take 0 _       = []
take _ []      = []
take n (x:xs)  = x : take (n-1) xs
drop 0 xs      = xs
drop _ []      = []
drop n (x:xs)  = drop (n-1) xs
```

# take n xs ++ drop n xs = xs

Proof by induction on n

- Base case: n = 0

  to prove ∀ xs `take 0 xs ++ drop 0 xs = xs`

- Inductive case: Assume IH: ∀ xs `take n xs ++ drop n xs = xs`

  to prove ∀ xs `take (n+1) xs ++ drop (n+1) xs = xs`

```
take 0 _        = []
take _ []       = []
take n (x:xs)   = x : take (n-1) xs
drop 0 xs       = xs
drop _ []       = []
drop n (x:xs)   = drop (n-1) xs
```

# take n xs ++ drop n xs = xs

Base case: $n = 0$

```
  take 0 xs ++ drop 0 xs
= { definition of take }
  [] ++ drop 0 xs
= { definition of drop }
  [] ++ xs
= { definition of ++ }
  xs
```

# take n xs ++ drop n xs = xs

```
take 0 _        = []
take _ []       = []
take n (x:xs)   = x : take (n-1) xs
drop 0 xs       = xs
drop _ []       = []
drop n (x:xs)   = drop (n-1) xs
```

Inductive case.

Assume IH: ∀ xs: `take n xs ++ drop n xs = xs`

Prove: ∀ xs: `take (n+1) xs ++ drop (n+1) xs = xs`

We make a case distinction

Case `xs = []`

```
take (n+1) [] ++ drop (n+1) []
= { definition of take, drop }
[] ++ []
= { definition of ++ }
[]
```

Case `xs = a:as`

```
take (n+1) (a:as) ++ drop (n+1) (a:as)
= { definition of take, drop }
(a:take n as) ++ drop n as
= { definition of ++ }
a:(take n as ++ drop n as)
= { IH }
a:as
```

# Take away

# The art of functional verification

- equational reasoning: substitution of equals for equals
- recursion and induction are two sides of the same coin
- explicit (ad-hoc) recursion and induction
  . . . versus canned recursion and the use of laws
- after-the-fact verification
  . . . versus calculating a program from a specification