Functional Programming

Lecture 8: Reasoning about programs

Twan van Laarhoven

7 November 2022

Outline

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

Equational reasoning

Algebra

Recall:

$$\begin{aligned} xy &= yx \\ x + (y + z) &= (x + y) + z \\ x(y + z) &= xy + xz \\ (x + y)z &= xz + yz \end{aligned}$$

Prove that
$(x + a)(x + b) = x^2 + (a + b)x + ab$.

Algebra

$$(x + a)(x + b)$$

Recall:

$$\begin{aligned} xy &= yx \\ x + (y + z) &= (x + y) + z \\ x(y + z) &= xy + xz \\ (x + y)z &= xz + yz \end{aligned}$$

Prove that
$(x + a)(x + b) = x^2 + (a + b)x + ab$.

Algebra

Recall:

$$
\begin{aligned}
xy &= yx \\
x + (y + z) &= (x + y) + z \\
x(y + z) &= xy + xz \\
(x + y)z &= xz + yz
\end{aligned}
$$

Prove that
$(x + a)(x + b) = x^2 + (a + b)x + ab$.

$$
\begin{aligned}
& (x + a)(x + b) \\
= & \quad \{ \text{ left distributivity } \}
\end{aligned}
$$

Algebra

Recall:

$$
\begin{aligned}
xy &= yx \\
x + (y + z) &= (x + y) + z \\
x(y + z) &= xy + xz \\
(x + y)z &= xz + yz
\end{aligned}
$$

Prove that
$(x + a)(x + b) = x^2 + (a + b)x + ab$.

$$
\begin{aligned}
& (x + a)(x + b) \\
= \quad & \{ \text{ left distributivity } \} \\
& (x + a)x + (x + a)b
\end{aligned}
$$

Algebra

Recall:

$$
\begin{aligned}
xy &= yx \\
x + (y + z) &= (x + y) + z \\
x(y + z) &= xy + xz \\
(x + y)z &= xz + yz
\end{aligned}
$$

Prove that
$(x + a)(x + b) = x^2 + (a + b)x + ab$.

$$
\begin{aligned}
& (x + a)(x + b) \\
= & \quad \{ \text{ left distributivity } \} \\
& (x + a)x + (x + a)b \\
= & \quad \{ \text{ right distributivity } \} \\
& xx + ax + xb + ab
\end{aligned}
$$

Algebra

Recall:

$$
\begin{aligned}
xy &= yx \\
x + (y + z) &= (x + y) + z \\
x(y + z) &= xy + xz \\
(x + y)z &= xz + yz
\end{aligned}
$$

Prove that
$(x + a)(x + b) = x^2 + (a + b)x + ab$.

$$
\begin{aligned}
& (x + a)(x + b) \\
= & \quad \{ \text{ left distributivity } \} \\
& (x + a)x + (x + a)b \\
= & \quad \{ \text{ right distributivity } \} \\
& xx + ax + xb + ab \\
= & \quad \{ \text{ squaring } \} \\
& x^2 + ax + xb + ab
\end{aligned}
$$

Algebra

Recall:

$$
\begin{aligned}
xy &= yx \\
x + (y + z) &= (x + y) + z \\
x(y + z) &= xy + xz \\
(x + y)z &= xz + yz
\end{aligned}
$$

Prove that
$(x + a)(x + b) = x^2 + (a + b)x + ab$.

$$
\begin{aligned}
& (x + a)(x + b) \\
= & \quad \{ \text{ left distributivity } \} \\
& (x + a)x + (x + a)b \\
= & \quad \{ \text{ right distributivity } \} \\
& xx + ax + xb + ab \\
= & \quad \{ \text{ squaring } \} \\
& x^2 + ax + xb + ab \\
= & \quad \{ \text{ commutativity } \} \\
& x^2 + ax + bx + ab
\end{aligned}
$$

Algebra

Recall:

$$xy = yx$$
$$x + (y + z) = (x + y) + z$$
$$x(y + z) = xy + xz$$
$$(x + y)z = xz + yz$$

Prove that
$(x + a)(x + b) = x^2 + (a + b)x + ab$.

$$\begin{aligned}
& (x + a)(x + b) \\
= & \quad \{ \text{ left distributivity } \} \\
& (x + a)x + (x + a)b \\
= & \quad \{ \text{ right distributivity } \} \\
& xx + ax + xb + ab \\
= & \quad \{ \text{ squaring } \} \\
& x^2 + ax + xb + ab \\
= & \quad \{ \text{ commutativity } \} \\
& x^2 + ax + bx + ab \\
= & \quad \{ \text{ right distributivity } \} \\
& x^2 + (a + b)x + ab
\end{aligned}$$

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

$$\begin{aligned} &\texttt{not} :: \texttt{Bool} \to \texttt{Bool} \\ &\texttt{not}\ \texttt{False} = \texttt{True} \qquad \text{-- } eq.\ 1 \\ &\texttt{not}\ \texttt{True}\ = \texttt{False} \qquad \text{-- } eq.\ 1 \end{aligned}$$

We can prove that: $\texttt{not}\ (\texttt{not}\ \texttt{False}) = \texttt{False}$

Equational reasoning (I)

Given

```
not :: Bool → Bool
not False = True      -- eq. 1
not True  = False     -- eq. 1
```

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

Proof:

```
not (not False)
```

Equational reasoning (I)

Given

```
not :: Bool → Bool
not False = True        -- eq. 1
not True  = False       -- eq. 1
```

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

Proof:

```
  not (not False)
=    { applying not (equation 1) }
  not True
```

Equational reasoning (I)

Given

```
not :: Bool → Bool
not False = True        -- eq. 1
not True  = False       -- eq. 1
```

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

Proof:

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

Equational reasoning (II)

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

Proof:

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

Where

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

Equational reasoning (III)

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

Proof:

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

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

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

Induction on lists

Programming with recursion for lists

Every recursive datatype comes with a *pattern of recursion*

- Define a function that: for any list `xs`, produces output `f xs`

Programming with recursion for lists

Every recursive datatype comes with a *pattern of recursion*

- Define a function that: for any list `xs`, produces output `f xs`
- The definition has two parts:
  1. Implement `f []` *(base case)*

Programming with recursion for lists

Every recursive datatype comes with a *pattern of recursion*

- Define a function that: for any list `xs`, produces output `f xs`
- The definition has two parts:
  1. Implement `f []` *(base case)*
  2. Implement `f (x:xs)`, perhaps using `f xs` *(recursive case)*

Programming with recursion for lists

Every recursive datatype comes with a *pattern of recursion*

- Define a function that: for any list `xs`, produces output `f xs`
- The definition has two parts:
  1. Implement `f []` *(base case)*
  2. Implement `f (x:xs)`, perhaps using `f xs` *(recursive case)*
- Part 2 uses a *recursive call*: `f xs`

Programming with recursion for lists

Every recursive datatype comes with a *pattern of recursion*

- Define a function that: for any list `xs`, produces output `f xs`
- The definition has two parts:
  1. Implement `f []` *(base case)*
  2. Implement `f (x:xs)`, perhaps using `f xs` *(recursive case)*
- Part 2 uses a *recursive call*: `f xs`
- This is exhaustive: every list is either `[]` or of the form `(x:xs)`

Programming with recursion for lists

Every recursive datatype comes with a *pattern of recursion*

- Define a function that: for any list `xs`, produces output `f xs`
- The definition has two parts:
  1. Implement `f []` *(base case)*
  2. Implement `f (x:xs)`, perhaps using `f xs` *(recursive case)*
- Part 2 uses a *recursive call*: `f xs`
- This is exhaustive: every list is either `[]` or of the form `(x:xs)`
- This terminates for finite lists: the argument gets smaller in each recursive 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

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)*

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 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 case)*
- 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 case)*
- 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. `[] ++ ys = ys` (left-identity)
2. `xs ++ [] = xs` (right-identity)
3. `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)` (associativity)

Example: Lists form a monoid

Recall definition

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

Monoid laws

1. `[] ++ ys = ys` (left-identity)
2. `xs ++ [] = xs` (right-identity)
3. `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)` (associativity)

Law 1: `[] ++ ys = ys` holds by definition of `(++)` (first equation)

Example: Lists form a monoid

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

To prove law 2: `xs ++ [] = xs`

Use induction over `xs`

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

```
  [] ++ []
= { applying ++ }
  []
```

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

```
  (a:as) ++ []
= { applying ++ }
  a:(as ++ [])
= { IH }
  a:as
```

Example: Lists form a monoid

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

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

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

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

$$
\begin{aligned}
&([] \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs \\
&= \{ \text{ applying } \mathbin{+\!\!+} \} \\
&ys \mathbin{+\!\!+} zs \\
&= \{ \text{ applying } \mathbin{+\!\!+} \} \\
&[] \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs)
\end{aligned}
$$

Example: Lists form a monoid

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

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

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

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

  $((a : as) \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs$

  $=$

  $(a : as) \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs)$

Example: Lists form a monoid

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

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

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

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

  $((a : as) \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs$
  $= \{$ applying $+\!\!+ \}$
  $(a : (as \mathbin{+\!\!+} ys)) \mathbin{+\!\!+} zs$
  $= \{$ applying $+\!\!+ \}$
  $a : ((as \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs)$

  $(a : as) \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs)$

Example: Lists form a monoid

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

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

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

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

  $((a : as) \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs$
  $= \{$ applying $+\!\!+ \}$
  $(a : (as \mathbin{+\!\!+} ys)) \mathbin{+\!\!+} zs$
  $= \{$ applying $+\!\!+ \}$
  $a : ((as \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs)$

  $a : (as \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs))$
  $= \{$ applying $+\!\!+ \}$
  $(a : as) \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs)$

Example: Lists form a monoid

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

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

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

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

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

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

Example: reverse-append

Prove that for all xs, ys:

```
reverse (xs ++ ys) = reverse ys ++ reverse xs.
```

Proof by induction on xs

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

```
  reverse ([] ++ ys)
= { applying ++ }
  reverse ys
= { unitality ++ }
  reverse ys ++ []
= { applying reverse }
  reverse ys ++ reverse []
```

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

Example: reverse-append

- Case: `xs = a:as` *(inductive step)*
  using IH: `reverse (as ++ ys) = reverse ys ++ reverse as`

```
reverse ((a:as) ++ ys)
= { applying ++ }
reverse (a:(as ++ ys)
= { applying reverse }
reverse (as ++ ys) ++ [a]
= { IH }
(reverse ys ++ reverse as) ++ [a]
= { associativity of ++ }
reverse ys ++ (reverse as ++ [a])
= { applying reverse }
reverse ys ++ reverse (a:as)
```

Equality of functions

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

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

Equality of functions

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

Suppose we want to prove: `reverse . reverse = id`
There is no variable to do induction on

Equality of functions

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

Suppose we want to prove: `reverse . reverse = id`
There is no variable to do induction on
Add missing arguments

Equality of functions

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

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`

Equality of functions

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

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 [])
= { applying reverse }
reverse []
= { applying reverse }
[]
```

Equality of functions

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

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

```
  reverse (reverse (a:as))
= { applying 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
= { applying ++ }
  a: ([] ++ as)
= { applying ++ }
  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
= { applying reverse }
[] ++ ys
= { applying ++ }
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
= { applying reverse }
(reverse as ++ [a]) ++ ys
= { associativity of ++ }
reverse as ++ ([a] ++ ys)
= { list notation, applying ++ }
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

```
foldr :: (a → b → b) → b → [a] → b
foldr (⊗) z []     = z
foldr (⊗) z (x:xs) = x ⊗ foldr z xs
```

Example:

```
          list = 1 : (2 : (3 : (4 : (5 : []))))
foldr ⊗ z list = 1 ⊗ (2 ⊗ (3 ⊗ (4 ⊗ (5 ⊗ z))))
```

foldr fusion

Example:

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

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

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

foldr fusion law

Captured by the foldr fusion law:
f (foldr (⊗) z xs) = foldr (⊞) (f z) xs if for all x y: f (x ⊗ y) = x ⊞ f y
Example: Suppose we want to prove that

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

What are f, (⊗), (⊞)?

f = (+ a)
(⊗) = (+)
(⊞) = (+)

Fold fusion example

```
f . foldr g z = foldr h (f z) if ∀ 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

```
f . foldr g z = foldr h (f z) if ∀ 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. `[]` = `f []`
2. `f (g x y)` = `h x (f y)`

Fold fusion example

```
f . foldr g z = foldr h (f z) if ∀ 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

```
f . foldr g z = foldr h (f z) if ∀ 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

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

1. $[] = \texttt{reverse } [] \quad \{ \text{ applying reverse } \}$

2. $\texttt{reverse (y ++ [x])}$
   $= \{ \text{ property reverse–append } \}$
   $\texttt{reverse [x] ++ reverse y}$
   $= \{ \text{ property reverse–singleton } \}$
   $\texttt{[x] ++ reverse y}$
   $= \{ \text{ list notation, applying ++ } \}$
   $\texttt{x : reverse y}$

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 $P(n+1)$

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 n xs ++ drop n xs = xs

Base case: n = 0

```
  take 0 xs ++ drop 0 xs
= { applying take }
  [] ++ drop 0 xs
= { applying drop }
  [] ++ xs
= { applying ++ }
  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

```
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) []
= { applying take, drop }
[] ++ []
= { applying ++ }
[]
```

Case `xs = a:as`

```
take (n+1) (a:as) ++ drop (n+1) (a:as)
= { applying take, drop }
(a:take n as) ++ drop n as
= { applying ++ }
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