

Lecture 11. Laws and induction

Functional Programming 2018/19

Doaitse Swierstra, Jurriaan Hage, Alejandro Serrano



Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Goals

- ▶ Reason about Haskell programs
 - ▶ Equational reasoning
 - ▶ Induction on data types

Chapter 16 (up to 16.6) from Hutton's book



Laws



Universiteit Utrecht

[Faculty of **Science**
Information and Computing **Sciences**]

Mathematical laws

- ▶ Mathematical functions do not depend on hidden, changeable values
 - ▶ $2 + 3 = 5$, both in $4 \times (2 + 3)$ and in $(2 + 3)^2$
- ▶ This allows us to more easily prove properties that operators and functions might have
 - ▶ These properties are called **laws**



Examples of laws for integers

+ commutes

$$x + y = y + x$$

× commutes

$$x \times y = y \times x$$

+ is associative

$$x + (y + z) = (x + y) + z$$

× distributes over +

$$x \times (y + z) = x \times y + x \times z$$

0 is the unit of +

$$x + 0 = x = 0 + x$$

1 is the unit of ×

$$x \times 1 = x = 1 \times x$$



Putting laws to good use

- ▶ Mathematical laws can help improve **performance**
 - ▶ That two expressions always have the same value does not mean that computing their value takes the same amount of time or memory
 - ▶ Replace a more expensive version with one that is cheaper to compute
- ▶ We can also prove properties to show that they **correctly** implement what we intended

In short, performance and correctness



Equational reasoning by example

$(a + b)^2$
= -- *definition of square*
 $(a + b) \times (a + b)$
= -- *distributivity*
 $((a + b) \times a) + ((a + b) \times b)$
= -- *commutativity of \times*
 $(a \times (a + b)) + (b \times (a + b))$
= -- *distributivity, twice*
= $(a \times a + a \times b) + (b \times a + b \times b)$
= -- *associativity of $+$*
 $a \times a + (a \times b + b \times a) + b \times b$
= -- *commutativity of \times*
 $a \times a + (a \times b + a \times b) + b \times b$
= -- *definition of square and $(2 \times)$*
 $a^2 + 2 \times a \times b + b^2$



Each theory has its laws

- ▶ We have seen laws that deal with arithmetic operators
- ▶ During courses in logic you have seen similar laws for logic operators

commutativity of \wedge

$$x \wedge y = y \wedge x$$

associativity of \wedge

$$x \wedge (y \wedge z) = (x \wedge y) \wedge z$$

distributivity of \wedge over

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$$

\vee De Morgan's law

$$\neg(x \wedge y) = \neg x \vee \neg y$$

Howard's law

$$(x \wedge y) \rightarrow z = x \rightarrow (y \rightarrow z)$$



A small proof in logic

$$\begin{aligned} & \neg((a \vee b) \vee c) \rightarrow \neg d \\ &= \text{-- De Morgan's law} \\ & (\neg(a \vee b) \wedge \neg c) \rightarrow \neg d \\ &= \text{-- De Morgan's law} \\ & ((\neg a \wedge \neg b) \wedge \neg c) \rightarrow \neg d \\ &= \text{-- Howard's law} \\ & (\neg a \wedge \neg b) \rightarrow (\neg c \rightarrow \neg d) \\ &= \text{-- Howard's law} \\ & \neg a \rightarrow (\neg b \rightarrow (\neg c \rightarrow \neg d)) \end{aligned}$$

- ▶ Proofs feel mechanical
 - ▶ You apply the “rules” implicit in the laws
 - ▶ Possibly even without understanding what \wedge and \vee do
- ▶ Always provide a hint why each equivalence holds!



Back to Haskell

- ▶ Haskell is referentially transparent
 - ▶ Calling a function twice with the same parameter is guaranteed to give the same result
- ▶ This allows us to prove equivalences as above
 - ▶ And use these to improve performance
- ▶ Any definition can be viewed in two ways

`double x = x + x`

1. The *definition* of a function
2. A *property* that can be used when reasoning
 - ▶ Replace `double x` by `x + x` and viceversa, for any `x`



A first example

For all compatible functions f and g , and lists xs

$$(\text{map } f \ . \ \text{map } g) \ xs = \text{map } (f \ . \ g) \ xs$$

This is not a definition, but a property/law

- ▶ The law can be shown to hold for the usual definitions of `map` and `(.)`

The right-hand side is more performant than the left-hand side, in general

- ▶ Two traversals are combined into one



A few important laws

1. Function composition is associative

$$f \cdot (g \cdot h) = (f \cdot g) \cdot h$$

2. `map f` distributes over `(++)`

$$\text{map } f \text{ (xs ++ ys)} = \text{map } f \text{ xs ++ map } f \text{ ys}$$

- ▶ Validates executing a large `map` on different cores
- ▶ There is a generalization to lists of lists

$$\text{map } f \cdot \text{concat} = \text{concap} \cdot \text{map } (\text{map } f)$$

3. `map` distributes over composition

$$\text{map } (f \cdot g) = \text{map } f \cdot \text{map } g$$



A few (more) important laws

4. If op is associative and e is the unit of op , then for finite lists xs

$$\text{foldr } op \ e \ xs = \text{foldl } op \ e \ xs$$

5. Under the same conditions, foldr on a singleton list is the identity

$$\text{foldr } op \ e \ [x] = x$$

These rules apply to very general functions

- The compiler uses these laws heavily to optimize



Relation to imperative languages

The law $\text{map } (f \ . \ g) = \text{map } f \ . \ \text{map } g$ is similar to the merging of subsequent loops

```
foreach (var elt in list) { stats1 }  
foreach (var elt in list) { stats2 }  
=  
foreach (var elt in list) { stats1 ; stats2 }
```

But due to side-effects in these languages, you have to be **really** careful when to apply them

- What could prevent us from merging the loops?



Why prove the laws?

- ▶ A proof guarantees that your optimization is justified
 - ▶ Otherwise you may accidentally change the behavior
- ▶ Proving is one additional way of increasing your confidence in the optimization that you perform
 - ▶ Others are testing, intuition, explanations...
- ▶ Of course, proofs can be wrong too
 - ▶ Proofs *can* be mechanically checked



Proving is like programming

1. Theorem = functionality of specification
2. Proof = implementation
3. Lemmas = library functions, local definitions
4. Proof strategies = paradigms, design patterns
 - ▶ **Equational reasoning**, i.e., by a chain of equalities
 - ▶ **Proof by induction**
 - ▶ Proof by contradiction: assuming the opposite, show that leads to contradiction
 - ▶ Breaking down equalities: $x = y$ iff $x \leq y$ and $y \leq x$
 - ▶ Combinatorial proofs

Like programming, proving takes *practice*



Equational reasoning



foldr over a singleton list

If e is the unit element of f , then $\text{foldr } f \ e \ [x] = x$

```
foldr f e [x]
= -- rewrite list notation
foldr f e (x : [])
= -- definition of foldr, case cons
f x (foldr f e [])
= -- definition of foldr, case empty
f x e
= -- e is neutral for f
x
```



Function composition is associative

For all functions f, g and h , $f \circ (g \circ h) = (f \circ g) \circ h$

Proof: consider any x

$$\begin{aligned} & (f \circ (g \circ h)) \, x \\ &= \text{-- definition of } (.) \\ & f \, ((g \circ h) \, x) \\ &= \text{-- definition of } (.) \\ & f \, (g \, (h \, x)) \\ &= \text{-- definition of } (.) \\ & (f \circ g) \, (h \, x) \\ &= \text{-- definition of } (.) \\ & ((f \circ g) \circ h) \, x \end{aligned}$$



Proving functions equal

- ▶ We prove functions f and g equal by proving that for all input x , $f\ x = g\ x$
 - ▶ They give the same results for the same inputs
 - ▶ Provided that they don't have side effects!
- ▶ They need *not* be the same function, as long as they behave in the same way
 - ▶ We call this **extensional** equality
- ▶ It is essential to make *no* assumptions about x
 - ▶ Otherwise, the proof does not work *for all* x



Two column style proofs

Reasoning from two ends is typically easier

- ▶ Rewrite the expression until you reach the same point
- ▶ Equalities can be read “backwards”

For all functions f , g and h , $f \circ (g \circ h) = (f \circ g) \circ h$

Proof: consider any x

$$\begin{aligned} & (f \circ (g \circ h)) \ x \\ &= \{- \text{defn. of } (\circ) -\} \\ & f \ ((g \circ h) \ x) \\ &= \{- \text{defn. of } (\circ) -\} \\ & f \ (g \ (h \ x)) \end{aligned}$$

$$\begin{aligned} & ((f \circ g) \circ h) \ x \\ &= \{- \text{defn. of } (\circ) -\} \\ & (f \circ g) \ (h \ x) \\ &= \{- \text{defn. of } (\circ) -\} \\ & f \ (g \ (h \ x)) \end{aligned}$$



map after (:)

For all type compatible values x and functions f ,

$$\text{map } f \ . \ (x \ :) = (f \ x \ :) \ . \ \text{map } f$$



map after (:)

For all type compatible values x and functions f ,

$$\text{map } f \text{ . } (x :) = (f x :) \text{ . map } f$$

Proof: consider any list xs

$$\begin{aligned} & (\text{map } f \text{ . } (x :)) \text{ } xs \\ &= \{- \text{ defn of } (.) \text{ -}\} \\ & \text{map } f ((x :) \text{ } xs) \\ &= \{- \text{ section notation -}\} \\ & \text{map } f (x : xs) \\ &= \{- \text{ defn. of map -}\} \\ & f x : \text{map } f \text{ } xs \end{aligned}$$

$$\begin{aligned} & ((f x :) \text{ . map } f) \text{ } xs \\ &= \{- \text{ defn of } (.) \text{ -}\} \\ & (f x :) (\text{map } f \text{ } xs) \\ &= \{- \text{ section notation -}\} \\ & f x : \text{map } f \text{ } xs \end{aligned}$$



not is an involution

The functions `not . not` and `id` are equal

Proof: consider any Boolean value `x`

► Case `x = False`

```
(not . not) False
= {- defn of (.) -}
not (not False)
= {- defn of not -}
not True
= {- defn of not -}
False
```

```
id False
= {- defn. of id -}
False
```

► Case `x = True`

```
(not . not) True
= {- as above -}
True
```

```
id True
= {- defn. of id -}
True
```



Case distinction

- ▶ To prove a property *for all* x , sometimes we need to distinguish the possible shapes that x may take
 - ▶ We need to be exhaustive to cover *all* cases
- ▶ For example,
 - ▶ A Boolean may be either True or False
 - ▶ A Maybe a value could be Nothing or Just x for some x
 - ▶ Given a data type of the form

```
data Shape = Circle    Point Float
           | Rectangle Point Float Float
           | Triangle  Point Point Point
```

you need to consider three different cases



Booleans and ($\&\&$) form a monoid

1. True is a neutral element: for any Boolean x ,
 $\text{True} \ \&\& \ x = x$
 $x \ \&\& \ \text{True} = x$
2. ($\&\&$) is associative: for any Booleans x , y , and z ,
 $x \ \&\& \ (y \ \&\& \ z) = (x \ \&\& \ y) \ \&\& \ z$



Maybe a forms a monoid

Consider the following operation:

`Just x <|> _ = Just x`

`Nothing <|> y = y`

1. `Nothing` is a neutral element: for any $x :: \text{Maybe } a$,

`Nothing <|> x = x`

`x <|> Nothing = x`

2. `(<|>)` is associative



Induction on data types



The case for lists

- ▶ Every (finite) list is built by finitely many $(:)$ 'es applied to a final $[]$
$$x : (y : (z : \dots (w : [])))$$
 - ▶ Don't bother about (finite) for now
- ▶ What if ...?
 - ▶ we prove a property P for $[]$
 - ▶ given any list xs , we can prove P holds for any list $x:xs$
- ▶ The *(structural) induction principle for (finite) lists* says that the result holds **for all** finite lists



The case for numbers and trees

- ▶ Every finite natural number can be seen as applying the successor function finitely many times to 0

4 = Succ (Succ (Succ (Succ Zero)))

- ▶ What if...?
 - ▶ we prove a property P for 0
 - ▶ given a number n , we can prove P for $\text{succ } n = n + 1$
- ▶ Every (finite) binary tree is built by finitely many Nodes ultimately applied to Leaf
 - ▶ What if...?
 - ▶ we prove a property P for Leaf
 - ▶ given any two trees l and r and a value x , we can prove P for Node $l \ x \ r$



Structural induction

A strategy for proving properties of structured data

1. State the law
 - a. If we speak about functions, introduce input variables
2. Enumerate the cases for one of the variables
 - ▶ Usually, one per constructor in the data type
3. Prove the base cases by equational reasoning
4. Prove the recursive cases
 - a. State the *induction hypotheses* (IH)
 - b. Use equational reasoning, applying IH when needed



Curry-Howard correspondence

The similarity with the recipe for recursion is **not** accidental

- ▶ We can use it to prove properties about programs *within* the code
 - ▶ Languages with theorem proving like Agda, Idris, or Coq
 - ▶ Plug-ins for Haskell such as LiquidHaskell
- ▶ Victor will tell you more about this on **25 October**



Structural induction for lists

1. State the law
 - a. If we speak about functions, introduce input variables
 - b. If needed, choose a variable to perform induction on
2. Prove the case `[]` by equational reasoning
3. State the induction hypothesis for `xs`
4. Prove the case `x:xs`, assuming that the IH holds



map f distributes over (++)

For all lists `xs` and `ys`

`map f (xs ++ ys) = map f xs ++ map f ys`

Proof: by induction on `xs`

► Case `xs = []`

`map f ([] ++ ys)`
= {- defn. of (++) -}
`map f ys`

`map f [] ++ map f ys`
= {- defn. of map -}
`[] ++ map f ys`
= {- defn of (++) -}
`map f ys`



map f distributes over (++)

► Case $xs = z:zs$

► IH: $\text{map } f (zs ++ ys) = \text{map } f zs ++ \text{map } f ys$

```
map f ((z:zs) ++ ys)
= {- defn. of (++) -}
map f (z : (zs ++ ys))
= {- defn of map -}
f z : map f (zs ++ ys)
```

```
map f (z:zs) ++ map f ys
= {- defn. of map -}
(f z : map f zs) ++ map f ys
= {- defn of (++) -}
f z : (map f zs ++ map f ys)
= {- IH -}
f z : map f (zs ++ ys)
```



map distributes over composition

For all compatible functions f and g ,

$$\text{map } (f \ . \ g) = \text{map } f \ . \ \text{map } g$$

Proof: by extensionality, we need to prove that for all xs

$$\text{map } (f \ . \ g) \ xs = (\text{map } f \ . \ \text{map } g) \ xs$$



map distributes over composition

For all compatible functions f and g ,

$$\text{map } (f \ . \ g) = \text{map } f \ . \ \text{map } g$$

Proof: by extensionality, we need to prove that for all xs

$$\text{map } (f \ . \ g) \ xs = (\text{map } f \ . \ \text{map } g) \ xs$$

We proceed by induction on xs

► Case $xs = []$

$$\begin{aligned} \text{map } (f \ . \ g) \ [] &= \{- \text{defn. of map} -\} \quad (\text{map } f \ . \ \text{map } g) \ [] \\ &= \{- \text{defn of } (.) -\} \quad \text{map } f \ (\text{map } g \ []) \\ &= \{- \text{defn. of map, twice} -\} \quad [] \end{aligned}$$



map distributes over composition

► Case $xs = z:zs$

► IH: $\text{map } (f \ . \ g) \ zs = (\text{map } f \ . \ \text{map } g) \ zs$

```
map (f.g) (z:zs)
= {- defn. of map -}
(f.g) z : map (f.g) zs
= {- defn of (.) -}
f (g z) : map (f.g) zs
```

```
(map f . map g) (z:zs)
= {- defn. of (.) -}
map f (map g (z:zs))
= {- defn. of map -}
map f (g z : map g zs)
= {- defn. of map -}
f (g z) : map f (map g zs)
= {- IH -}
f (g z) : map (f.g) zs
```



reverse is an involution

The functions `reverse . reverse` and `id` are equal

Proof: by extensionality we need to prove that for all `xs`

$$\begin{aligned} &(\text{reverse} \ . \ \text{reverse}) \ \text{xs} \\ &= \text{reverse} \ \text{reverse} \ \text{xs} \qquad = \ \text{id} \ \text{xs} \end{aligned}$$


reverse is an involution

The functions `reverse . reverse` and `id` are equal

Proof: by extensionality we need to prove that for all `xs`

```
(reverse . reverse) xs  
= reverse reverse xs      = id xs
```

We proceed by induction on `xs`

► Case `xs = []`

```
reverse (reverse [])      id []  
= {- defn. of reverse -}  = {- defn. of id -}  
reverse []                []  
= {- defn. of reverse -}  
[]
```



reverse is an involution

► Case $xs = z:zs$

► IH: $\text{reverse} (\text{reverse } zs) = \text{id } zs = zs$

$\text{reverse} (\text{reverse } (z:zs))$	$\text{id } (z:zs)$
$= \{- \text{defn. of reverse} -\}$	$= \{- \text{defn of id} -\}$
$\text{reverse} (\text{reverse } zs ++ [z])$	$z:zs$

We are stuck!



Lemmas

To keep going we defer some parts as *lemmas*

- ▶ Similar to local definitions in code
- ▶ Lemmas have to be proven separately

In our case, we need the following lemmas

```
-- Distributivity of (++) over reverse
reverse (xs ++ ys) = reverse ys ++ reverse xs
-- Reverse on singleton lists
reverse [x]         = [x]
```

Finding the right lemmas involves lots of practice



reverse is an involution

```
reverse (reverse (z:zs))  
= {- defn. of reverse -}  
reverse (reverse zs ++ [z])  
= {- distributivity -}  
reverse [z] ++ reverse (reverse zs)  
= {- reverse on singleton -}  
[z] ++ reverse (reverse zs)  
= {- IH -}  
[z] ++ zs  
= {- defn of (++) -}  
z : zs
```

```
id (z : zs)  
= {- defn of id -}  
z : zs
```

We still need to prove the lemmas separately



reverse is an involution

Lemma: `reverse (xs++ys) = reverse ys ++ reverse xs`

Proof: by induction on `xs` ...

Lemma: `reverse [x] = [x]`

Proof:

```
reverse [x]
= {- list notation -}
reverse (x : [])
= {- defn. of reverse -}
reverse [] ++ [x]
= {- defn. of reverse -}
[] ++ [x]
= {- defn. of (++) -}
[x]
```



Mathematical induction

- ▶ To prove that a statement P holds for all $n \in \mathbb{N}$
 - ▶ Prove that it holds for 0
 - ▶ Prove that it holds for $n + 1$ assuming that it holds for n
- ▶ This strategy is equivalent to structural induction on

`data Nat = Zero | Succ Nat`

This encoding is called *Peano numbers*

Note: there are stronger forms of induction for natural numbers, but we restrict ourselves to the simpler one



Arithmetic using Peano numbers

Addition and multiplication are defined by recursion

```
add  :: Nat -> Nat -> Nat
add  Zero      m = m
--       $0 + m = m$ 
add  (Succ n) m = Succ (n + m)
--       $(n + 1) + m = (n + m) + 1$ 

mult :: Nat -> Nat -> Nat
mult Zero      m = Zero
--       $0 \times m = 0$ 
mult (Succ n) m = add (mult n m) m
--       $(n + 1) \times m = (n \times m) + m$ 
```



0 is right identity for addition

For all natural n , $\text{add } n \text{ Zero} = n$

Proof: by induction on n

► Case $n = \text{Zero}$

$\text{add } \text{Zero } \text{Zero}$
 $= \{- \text{ defn. of add } -\}$
 Zero

► Case $n = \text{Succ } p$

► IH: $\text{add } p \text{ Zero} = p$
 $\text{add } (\text{Succ } p) \text{ Zero}$
 $= \{- \text{ defn. of add } -\}$
 $\text{Succ } (\text{add } p \text{ Zero})$
 $= \{- \text{ IH } -\}$
 $\text{Succ } p$



Some functions over binary trees

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

size t counts the number of nodes

```
size Leaf = 0
```

```
size (Node l _ r) = 1 + size l + size r
```

mirror t obtains the “rotated” image of a tree

```
mirror Leaf = Leaf
```

```
mirror (Node l x r) = Node (mirror r) x (mirror l)
```



mirror preserves the size

For all trees t , $\text{size}(\text{mirror } t) = \text{size } t$



mirror preserves the size

For all trees t , $\text{size} (\text{mirror } t) = \text{size } t$

Proof: by induction on t

► Case $t = \text{Leaf}$
 $\text{size} (\text{mirror } \text{Leaf})$
 $= \{- \text{ defn. of mirror } -\}$
 $\text{size } \text{Leaf}$



mirror preserves the size

- ▶ Case $t = \text{Node } l \times r$
 - ▶ We get one induction hypothesis per recursive position
 - ▶ IH1: $\text{size } (\text{mirror } l) = \text{size } l$
 - ▶ IH2: $\text{size } (\text{mirror } r) = \text{size } r$

```
size (mirror (Node l x r))  
= {- defn. of mirror -}  
size (Node (mirror r) x (mirror l))  
= {- defn. of size -}  
1 + size (mirror r) + size (mirror l)  
= {- IH1 and IH2 -}  
1 + size r + size l  
= {- commutativity of addition -}  
1 + size l + size r  
= {- defn. of size -}  
size (Node l x r)
```



0 is an absorbing element for product

For all natural n , $\text{mult } n \text{ Zero} = \text{Zero}$



Some advice

- ▶ Proving takes practice, just like programming
 - ▶ So **practice**
 - ▶ Both the book and the lecture notes contain many more examples of inductive proofs
- ▶ Inductive proofs are **definitely** part of the final exam
 - ▶ Could be about lists, natural numbers, trees, or some other recursively defined data type

