Lecture 11. Laws and induction

Functional Programming

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

What does it mean for programs to be equal/equivalent?

Goals

- Equational reasoning: proving program equalities
- Reasoning principles at various types:
  - inductive proofs at algebraic data types;
  - extensional equality at function types.

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

Laws

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	$x + y = y + x$
$\times$ commutes	$x \times y = y \times x$
$+$ is associative	$x + (y + z) = (x + y) + z$
$\times$ 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 $\times$	$x \times 1 = x = 1 \times x$

Putting laws to good use

Why care about program equivalences?

Putting laws to good use

Why care about program equivalences?

- 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

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Putting laws to good use

Why care about program equivalences?

- 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

Putting laws to good use

Why care about program equivalences?

- 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) × (a + b)
= -- distributivity
((a + b) × a) + ((a + b) × b)
= -- commutativity of ×
(a × (a + b)) + (b × (a + b))
= -- distributivity, twice
= (a × a + a × b) + (b × a + b × b)
= -- associativity of +
a × a + (a × b + b × a) + b × b
= -- commutativity of ×
a × a + (a × b + a × b) + b × b
= -- definition of square and (2 ×)
a² + 2 × a × b + b²
```

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

```
¬((a \/ b) \/ c) → ¬d
= -- De Morgan's law
(¬(a \/ b) /\ ¬c) → ¬d
= -- De Morgan's law
((¬a /\ ¬b) /\ ¬c) → ¬d
= -- Howard's law
(¬a /\ ¬b) → (¬c → ¬d)
= -- Howard's law
¬a → (¬b → (¬c → ¬d))
```

- 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

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

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`

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

```haskell
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`

- NB: by contrast, <- “assignments” in `do`-blocks are not referentially transparent!

A first example

For all compatible functions `f` and `g`, and lists `xs`

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

A first example

For all compatible functions `f` and `g`, and lists `xs`

```
(map f . map g) xs = 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 that the left-hand side, in general

- Two traversals are combined into one

Relation to imperative languages

The law `map (f . g) = map f . 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 }
```

Relation to imperative languages

The law `map (f . g) = map f . 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?

A few important laws

1. Function composition is associative

   ```
   f . (g . h) = (f . g) . h
   ```


Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

A few important laws

1. Function composition is associative

```
f . (g . h) = (f . g) . h
```

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

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

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

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

A few important laws

1. Function composition is associative

   ```
   f . (g . h) = (f . g) . h
   ```

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

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

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

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

3. `map` distributes over composition

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

A few (more) important laws

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

   ```
   foldr op e xs = foldl op e xs
   ```

A few (more) important laws

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

   ```
   foldr op e xs = foldl op e xs
   ```

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

   ```
   foldr op e [x] = x
   ```

These rules apply to very general functions

- The compiler uses these laws heavily to optimize

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. Proposition = functionality of specification
2. Proof = implementation
3. Lemmas = library functions, local definitions

Proving is like programming

1. Proposition = 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 contraposition: prove `p` implies `q` by showing `not q` implies `not p`
   - 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 `op`, then `foldr op e [x] = x`

```
foldr op e [x]
= ...
```


Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

`foldr` over a singleton list

If `e` is the unit element of `op`, then `foldr op e [x] = x`

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

`foldl` over a singleton list

If `e` is the unit element of `op`, then `foldl op e [x] = x`

```
foldl op e [x]
= ...
```

Try it yourself!

`foldl` over a singleton list

If `e` is the unit element of `op`, then `foldl op e [x] = x`

```
foldl op e [x]
= -- rewrite list syntactic sugar
foldl op e (x:[])
= -- definition foldl
foldl op (op e x) []
= -- definition foldl
op e x
= -- e is neutral for op
x
```

Function composition is associative

For all functions `f`, `g` and `h`, `f . (g . h) = (f . g) . h`

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Function composition is associative

For all functions `f`, `g` and `h`, `f . (g . h) = (f . g) . h`

Proof: consider any `x`

```
(f . (g . h)) x
= -- definition of (.)
f ((g . h) x)
= -- definition of (.)
f (g (h x))
= -- definition of (.)
(f . g) (h x)
= -- definition of (.)
((f . g) . h) x
```


Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

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 . (g . h) = (f . g) . h`

Proof: consider any `x`

```
(f . (g . h)) x
= {- defn. of (.) -}
f ((g . h) x)
= {- defn. of (.) -}
f (g (h x))
```

```
((f . g) . h) x
= {- defn. of (.) -}
(f . g) (h x)
= {- defn. of (.) -}
f (g (h x))
```

`map` after `(:)`

For all type compatible values `x` and functions `f`,

```
map f . (x :) = (f x :) . map f
```

`map` after `(:)`

For all type compatible values `x` and functions `f`,

```
map f . (x :) = (f x :) . map f
```

Proof: consider any list `xs`

```
(map f . (x :)) xs
= {- defn of (.) -}
map f ((x :) xs)
= {- section notation -}
map f (x : xs)
= {- defn. of map -}
f x : map f xs
```

```
((f x :) . map f) xs
= {- defn of (.) -}
(f x :) (map f xs)
= {- section notation -}
f x : map f xs
```

`not` is an involution

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

Let's try!

`not` is an involution

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

Proof: consider any Boolean value `x`

- Case `x = False`

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

`not` is an involution

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

Proof: consider any Boolean value `x`

- Case `x = False`

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

- Case `x = True`

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

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

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

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
- Let's try an example!

Homework: Booleans and `(&&)` form a monoid

1. `True` is a neutral element: for any Boolean `x`,

   ```
   True && x = x
   x && True = x
   ```

2. `(&&)` is associative: for any Booleans `x`, `y`, and `z`,

   ```
   x && (y && z) = (x && y) && z
   ```

Homework: `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 :: Maybe a`,
   ```
   Nothing <|> x = x
   x <|> Nothing = x
   ```
2. `(<|>)` is associative

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Induction on data types

The case for lists

- Every (finite) list is built by finitely many `(:)`'es appplied to a final `[]`

  `x : (y : (z : ... (w : [])))`

  - Don't bother about (finite) for now
- What if ...?
  - we prove a property P for []
  - given any list `xs` satisfying P, we can prove P holds for `x:xs`
- The (structural) induction principle for (finite) lists says that the result then 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` satisfying P, we can prove P for `succ n = n + 1`

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` satisfying P, we can prove P for `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` satisfying P and a value `x`, we can prove P for `Node l x r`

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Structural induction

A strategy for proving properties of strucured 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

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
```

`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: `map f (zs ++ ys) = map f zs ++ 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`,

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

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

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

`map` distributes over composition

For all compatible functions `f` and `g`,

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

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

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

We proceed by induction on `xs`

- Case `xs = []`

```
map (f . g) []                (map f . map g) []
= {- defn. of map -}          = {- defn of (.) -}
[]                            map f (map g [])
                              = {- defn. of map, twice -}
                              []
```


Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

`map` distributes over composition

- Case `xs = z:zs`
  - IH: `map (f . g) zs = (map f . 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`

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

`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: `reverse (reverse zs) = id zs = zs`

`reverse (reverse (z:zs))`	`id (z:zs)`
`= {- defn. of reverse -}`	`= {- defn of id -}`
`reverse (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                                 id (z : zs)
= {- defn of (++) -}                      = {- defn of id -}
z : zs                                    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 × m = 0
mult (Succ n) m = add (mult n m) m
--   (n + 1) × m = (n × m) + m
```

0 is right identity for addition

For all natural `n`, `add n Zero = n`

Proof: by induction on `n`

- Case `n = Zero`

```
add Zero Zero
= {- defn. of add -}
Zero
```

- Case `n = Succ p`
  - IH: `add p Zero = p`

```
add (Succ p) Zero
= {- defn. of add -}
Succ (add p Zero)
= {- IH -}
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`, `size (mirror t) = size t`

`mirror` preserves the size

For all trees `t`, `size (mirror t) = size t`

Proof: by induction on `t`

- Case `t = Leaf`

  ```
  size (mirror Leaf)
  = {- defn. of mirror -}
  size Leaf
  ```

`mirror` preserves the size

- Case `t = Node l x r`
  - We get one induction hypothesis per recursive position
  - IH1: `size (mirror l) = size l`
  - IH2: `size (mirror r) = 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`, `mult n Zero = Zero`

Summary

- Proving program equivalences is useful for
  - establishing correctness;
  - finding opportunities for improving performance;
- We prove equivalences using
  - definitions and laws;
  - extensional equality at function types;
  - case distinction and induction on algebraic data types;

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