Course year 2015 - 2016
Equational reasoning and induction

Doaitse Swierstra, Jurriaan Hage

Utrecht University
1. The 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 function might have. These properties are called laws

► Typical examples 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 sometimes 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
- The law allows us to replace a more expensive version with one that is cheaper to compute
- We can also prove properties of expressions to show that they correspond to what we intended to implement
An equational proof of $(a + b)^2 = a^2 + 2ab + b^2$ 

\[
(a+b)^2 \\
= \quad \text{(definition square)} \\
(a+b) \times (a+b) \\
= \quad \text{(distributivity)} \\
((a+b) \times a) + (a+b) \times b \\
= \quad \text{(commutativity × (twice))} \\
(a \times (a+b)) + (b \times (a+b)) \\
= \quad \text{(distributivity (twice))} \\
(a \times a + a \times b) + (b \times a + b \times b) \\
= \quad \text{(associativity +)} \\
a \times a + (a \times b + b \times a) + b \times b \\
= \quad \text{(definition square (twice))} \\
a^2 + (a \times b + b \times a) + b^2 \\
= \quad \text{(commutativity ×)} \\
a^2 + (a \times b + a \times b) + b^2 \\
= \quad \text{(definition ‘(2×)’) } \\
a^2 + 2 \times a \times b + b^2 
\]
Each theory has its laws

- We have seen laws that deal with arithmetic operators
- During courses on logic you see similar laws for logic operators:
  - commutativity $\land$ \[ x \land y = y \land x \]
  - associativity $\land$ \[ x \land (y \land z) = (x \land y) \land z \]
  - distributivity $\land$ over $\lor$ \[ x \land (y \lor z) = (x \land y) \lor (x \land z) \]
  - De Morgan’s law \[ \neg(x \land y) = \neg x \lor \neg y \]
  - Howard’s law \[ (x \land y) \rightarrow z = x \rightarrow (y \rightarrow z) \]
A small proof in logic

\[ \neg((a \lor b) \lor c) \rightarrow \neg d = \neg a \rightarrow (\neg b \rightarrow (\neg c \rightarrow \neg d)) \] can be proven as follows:

\[ \neg((a \lor b) \lor c) \rightarrow \neg d \]

\[ = \text{ (De Morgan’s law)} \]

\[ (\neg(a \lor b) \land \neg c) \rightarrow \neg d \]

\[ = \text{ (De Morgan’s law)} \]

\[ (((\neg a \land \neg b) \land \neg c) \rightarrow \neg d) \]

\[ = \text{ (Howard’s law)} \]

\[ (\neg a \land \neg b) \rightarrow (\neg c \rightarrow \neg d) \]

\[ = \text{ (Howard’s law)} \]

\[ \neg a \rightarrow (\neg b \rightarrow (\neg c \rightarrow \neg d)) \]

Proofs feel mechanical: you apply the “rules” implicit in the laws, possibly even without understanding what \( \lor \) and \( \land \) do.

Always provide a hint why each equivalence holds!
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 by replacing equals with equals.
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) that can be shown to hold for the usual definitions of $\text{map}$ and $(.)$.

Typically, you need to unfold definitions like that of $\text{map}$ so have them at the ready (in the exam the definitions will be provided)
Putting the law to use

Consider

\[(\text{altsum \ . \ zipWith (*) \ ry \ . \ map \ det \ . \ map \ Mat \ . \ gaps \ . \ transpose}) \ rys\]

We can apply the law \((\text{map } f \ . \ \text{map } g) \ xs = \text{map } (f \ . \ g) \ xs\) to obtain

\[(\text{altsum \ . \ zipWith (*) \ ry \ . \ map \ (det \ . \ Mat) \ . \ gaps \ . \ transpose}) \ rys\]

The latter will typically execute faster: two traversals of a list are combined into one
Putting the law to use

Consider

\[(\text{altsum} \cdot \text{zipWith} (\ast) \text{ry} \cdot \text{map det} \cdot \text{map Mat} \cdot \text{gaps} \cdot \text{transpose}) \text{rys}\]

We can apply the law \((\text{map } f \cdot \text{map } g) \, xs = \text{map } (f \cdot g) \, xs\) to obtain

\[(\text{altsum} \cdot \text{zipWith} (\ast) \text{ry} \cdot \text{map } (\text{det} \cdot \text{Mat}) \cdot \text{gaps} \cdot \text{transpose}) \text{rys}\]

The latter will typically execute faster: two traversals of a list are combined into one

Unless the compiler can actually perform the transformation for you
A few important laws

The following laws are used regularly:

- function composition is associative:

\[ f \cdot (g \cdot h) = (f \cdot g) \cdot h \]
The following laws are used regularly:

- function composition is associative:
  \[ f \circ (g \circ h) = (f \circ g) \circ h \]

- \textit{map} \( f \) distributes over \( ++ \):
  \[ \text{map} \ f \ (xs ++ ys) = \text{map} \ f \ xs ++ \text{map} \ f \ ys \]

Validates executing a large \textit{map} on different cores.
A few important laws

The following laws are used regularly:

- function composition is associative:

\[ f \cdot (g \cdot h) = (f \cdot g) \cdot h \]

- \textit{map } \textit{f} distributes over \textit{++}:

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

Validates executing a large map on different cores.

- generalising to a list of lists (instead of just two):

\[ \text{map } f \ . \ \text{concat} = \text{concat} \ . \ \text{map } (\text{map } f) \]
and some more...

- *map* distributes over composition:

  \[
  \text{map} (f \cdot g) = \text{map } f \cdot \text{map } g
  \]
and some more...

- *map* distributes over composition:
  
  \[ \text{map} (f \cdot g) = \text{map} f \cdot \text{map} g \]

- If *op* is associative (so \( x \, \text{op} \, (y \, \text{op} \, z) = (x \, \text{op} \, y) \, \text{op} \, z \)), and \( e \) is the unit of \( \text{op} \), then for finite lists \( xs \):
  
  \[ \text{foldr} \, (\text{op}) \, e \, xs = \text{foldl} \, (\text{op}) \, e \, xs \]
and some more...

- `map` distributes over composition:
  \[ map (f \cdot g) = map f \cdot map g \]

- If `op` is associative (so \( x \ op (y \ op z) = (x \ op y) \ op z \)), and \( e \) is the unit of `op`, then for finite lists `xs`:
  \[ foldr (op) \ e \ xs = foldl (op) \ e \ xs \]

- Under the same conditions `foldr` on a singleton list is the identity function:
  \[ foldr (op) \ e \ [x] = x \]
Relation to imperative languages

- Optimisations are also possible in imperative languages
- The law $map (f \cdot g) = map f \cdot map g$ is the merging of subsequent loops

```plaintext
foreach elt1 from list1 {
  stats1
}
foreach elt2 from list1 {
  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?
Relation to imperative languages

- Optimisations are also possible in imperative languages
- The law $\text{map } (f \circ g) = \text{map } f \circ \text{map } g$ is the merging of subsequent loops

```
foreach elt1 from list1 {
    stats1
}
foreach elt2 from list1 {
    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?
  - $elt1$ and $elt2$ could be the same name
  - Side effects (e.g. on other variables)
2. Proving the laws
Why prove laws?

▶ A proof guarantees that your optimisation is justified
▶ Without it you may accidentally change the output of your program
▶ Of course, proofs can be wrong too
▶ Proving is one additional way of increasing your confidence in the optimisation that you perform
  ▶ Others are testing, intuiting, explaining (in comments)...
▶ Proofs can be mechanically checked.
Proving is like programming

- (named) theorem = functionality or specification
- proof = implementation
- (named/numbered) lemmas = library functions, local definitions,
- proof strategies = programming paradigms, design patterns
- Some examples of proof strategies:
  - algebraic or equational, i.e., reasoning by a chain of equalities
  - proof by induction (typically over inductively defined datatypes, like natural numbers, trees and lists)
  - proof by contradiction: assuming the opposite, show that that leads to contradiction
  - breaking down equalities: $x = y$ iff $x \leq y$ and $y \leq x$
  - combinatorial proofs: often with large case distinctions
- Like programming proving takes practice
A first proof

Law foldr over a single element list

If e is the unit element (aka neutral element or identity) of f, then foldr f e [x] = x.
Law foldr over a single element list

If e is the unit element (aka neutral element or identity) of f, then $\text{foldr} \ f \ e \ [x] = x$.

Proof:

$\text{foldr} \ f \ e \ [x]$

$= \ (\text{rewrite list notation})$

$\text{foldr} \ f \ e \ (x:[])$

$= \ (\text{unfold def. foldr})$

$f \ x \ (\text{foldr} \ f \ e \ [])$

$= \ (\text{def. foldr case empty list})$

$f \ x \ e$

$= \ (e \text{ is neutral for } f)$

$x$
Another example of equational reasoning

**Law** function composition is associative

For all functions $f$, $g$, $h$,

$$f \cdot (g \cdot h) = (f \cdot g) \cdot h$$
Another example of equational reasoning

Law: function composition is associative

For all functions \( f, g, h \),
\[ f \cdot (g \cdot h) = (f \cdot g) \cdot h \]

Proof: consider any \( x \):
\[
(f \cdot (g \cdot h)) x = (\text{def. (~)}) \\
= f ((g \cdot h) x) = (\text{def. (~)}) \\
= f (g (h x)) = (\text{def. (~)}) \\
= (f \cdot g) (h x) = (\text{def. (~)}) \\
= ((f \cdot g) \cdot h) x
\]
We prove functions $f$ and $g$ equal by proving that for all $x$: $f(x) = g(x)$.

In other words: they give the same results for the same inputs (and they can’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 assumption about $x$, otherwise it may not work for all $x$. 
Two column style proofs

Apply to $x$ on both sides and show that they rewrite to the same expression:

<table>
<thead>
<tr>
<th></th>
<th>$f \cdot (g \cdot h)$</th>
<th>$(f \cdot g) \cdot h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$(f \cdot (g \cdot h)) \ x$</td>
<td>$((f \cdot g) \cdot h) \ x$</td>
</tr>
<tr>
<td></td>
<td>$= (\text{def. (.))}$</td>
<td>$= (\text{def. (.))}$</td>
</tr>
<tr>
<td></td>
<td>$f ((g \cdot h) \ x)$</td>
<td>$(f \cdot g) (h \ x)$</td>
</tr>
<tr>
<td></td>
<td>$= (\text{def. (.))}$</td>
<td>$= (\text{def. (.))}$</td>
</tr>
<tr>
<td></td>
<td>$f (g (h \ x))$</td>
<td>$f (g (h \ x))$</td>
</tr>
</tbody>
</table>

Reasoning from two ends is typically easier.
Another two column style proof

Law  map after cons

*For all type compatible values $x$ and functions $f$:

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

<table>
<thead>
<tr>
<th>$\text{map } f \ . \ (x:) \</th>
<th>\ ((f \ x):) \ . \ \text{map } f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{xs}$</td>
<td>$\text{xs}$</td>
</tr>
<tr>
<td>$(\text{map } f \ . \ (x:) ) \ \text{xs}$</td>
<td>$( ( (f \ x):) \ . \ \text{map } f ) \ \text{xs}$</td>
</tr>
<tr>
<td>$= (\text{def. } (\cdot))$</td>
<td>$= (\text{def. } (\cdot))$</td>
</tr>
<tr>
<td>$\text{map } f \ ((x:) \ \text{xs})$</td>
<td>$((f \ x):) \ (\text{map } f \ \text{xs})$</td>
</tr>
<tr>
<td>$= (\text{section notation})$</td>
<td>$= (\text{section notation})$</td>
</tr>
<tr>
<td>$\text{map } f \ (x: \text{xs})$</td>
<td>$f \ x : \ \text{map } f \ \text{xs}$</td>
</tr>
<tr>
<td>$= (\text{def. } \text{map})$</td>
<td></td>
</tr>
<tr>
<td>$f \ x : \ \text{map } f \ \text{xs}$</td>
<td></td>
</tr>
</tbody>
</table>
3. Proof by structural induction
The case for lists

- Every finite list is built by finitely many (:)’es applied to a single []
- What if
  - we prove a property $P$ for []
  - given any list $xs$, we can prove $P$ holds for any list $(x : xs)$?
- Then the (structural) induction principle for (finite) lists says that the result holds for all finite lists
- A simple example for $P$: “$xs$ has length at least 0”
Structural induction

A strategy for proving properties of structured data

Examples

- natural numbers
- lists
- trees

Characteristic is that a tree can only consist of subtrees that are strictly smaller, and lists only of such sublists

The relation with recursion that tends to work well only in such cases is not accidental

- You typically need induction to prove recursive functions correct
Induction principle for a datatype definition

```haskell
data Tree2 = Leafi Int | Leafb Bool | Split Tree2 Tree2
```

- Many Haskell datatypes come with an induction principle.
- How does it work for `Tree2`:
  - The base cases for `Leafi` and `Leafb` have no subtrees.
  - The inductive case for `Split` does have them.
- What is the corresponding induction principle here:
  - Base case 1: property $P$ holds for `Leafi nr` with $nr$ any `Int`.
  - Base case 2: property $P$ holds for `Leafb bl` with $bl$ any `Bool`.
  - Inductive case: property $P$ holds for `Split t1 t2` under the condition that $P$ holds for both $t1$ and $t2$.
  - Together they imply that $P$ holds for all values of type `Tree2`.
- The justification is that you can unravel by means of the inductive case until you arrive at the base case(s).
The relation with mathematical induction

- Mathematical induction is a special case of structural induction
- It works only for natural numbers:

```haskell
data Nat = Zero | Successor Nat
```

- The induction principle here is: given a property $P$,
  - if $P$ holds for $\text{Zero}$ and
  - given that $P$ holds for any natural number $n$, it also holds for $\text{Successor } n$,
  - then $P$ has been proven to hold for all values of type $\text{Nat}$

- This encoding of naturals as a Haskell datatype is called the Peano encoding

- We ignore more general and other forms of induction (Noetherian induction, co-induction)
Warning: this proof is not correct

Theorem: in any collection of horses, all horses are of the same colour
Proof: by induction
Base case (collection of size 1): easy
Inductive case (collection of size $n + 1$): put all $n + 1$ horses in a row. Consider the first $n$ horses: by induction they must all be of the same colour. Then look only at the last $n$ horses: they must also have the same colour. So the first horse, the middle horses, and the last horse must all have the same colour.

What is wrong here?
4. Inductive proofs for Haskell code
A first proof by induction (on lists)

<table>
<thead>
<tr>
<th>Law</th>
<th>map after ++</th>
</tr>
</thead>
<tbody>
<tr>
<td>IH xs</td>
<td>map f (xs++ys)</td>
</tr>
<tr>
<td>[]</td>
<td>map f ([]++ys) = (def. ++) map f ys</td>
</tr>
<tr>
<td>x:xs</td>
<td>map f ((x:xs)+y) = (def. ++) map f (x:(xs++ys)) = (def. map) f x : map f (xs++ys)</td>
</tr>
<tr>
<td>Law</td>
<td>map after function composition</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>map ((f \cdot g))</td>
</tr>
<tr>
<td>IH xs</td>
<td>map ((f \cdot g)) xs</td>
</tr>
<tr>
<td></td>
<td>= (def. (().))</td>
</tr>
<tr>
<td></td>
<td>map (f) (map (g) xs)</td>
</tr>
<tr>
<td>[]</td>
<td>map ((f \cdot g)) []</td>
</tr>
<tr>
<td></td>
<td>= (def. map)</td>
</tr>
<tr>
<td></td>
<td>[]</td>
</tr>
<tr>
<td>x:xs</td>
<td>map ((f \cdot g)) (x:xs)</td>
</tr>
<tr>
<td></td>
<td>= (def. map)</td>
</tr>
<tr>
<td></td>
<td>(f.g) x : map ((f \cdot g)) xs</td>
</tr>
<tr>
<td></td>
<td>= (def. (().))</td>
</tr>
<tr>
<td></td>
<td>f(g x) : map ((f \cdot g)) xs</td>
</tr>
<tr>
<td></td>
<td>= (IH xs)</td>
</tr>
<tr>
<td></td>
<td>f(g x) : map ((f \cdot g)) xs</td>
</tr>
</tbody>
</table>
Closing remarks

- Proving takes practice, just like programming
- So practice
- The lecture notes contain many more examples of inductive proofs
- Inductive proofs are definitely part of the final exam
  - Could be about lists, about natural numbers, or about some other recursively defined datatype
- And inductive proofs themselves include algebraic reasoning