Equational Reasoning

COMP2209 - Programming III

Dr Julian Rathke
High School Algebra

• We’ve all learned in school that we can manipulate polynomial equations in order to help solve them.

• For example, the “completing the squares” method of solving quadratics does just this.

• We know that e.g. that \((x + a)(x + a) = x^2 + 2ax + a^2\)

• But what we often don’t think about when we make these manipulations is what underlying equations allow us to do so.

\[
\begin{align*}
xy &= yx \\
x + (y + z) &= (x + y) + z \\
x(y + z) &= xy + yz \\
(x + y)z &=xz + yz
\end{align*}
\]

• In fact, we call this algebra, but an algebra is a general term for a set of things, operations on those things and equations between them.
An algebra of programs

• So can we think of programs as an algebra?
• Yes, take the set of all programs that you can legally form according to the grammar of your language
• Take the operations to be the programming language constructs
• Now consider what equations exist between terms.
  • This is the hard bit!
• In Haskell, everything is defined equationally!
  • e.g. double x = x + x
• And because of the way that core Haskell is built, no references, no I/O etc, these equations serve as the equations of our algebra of programs.
• We can reason with them and manipulate programs as we do with polynomials in high school.
So can we prove something now?

- So how about we prove that `even (double x) = True` for any `x`?
- Well that’s actually a little trickier than it looks.
- We need to look at the equations for `double` and `even`:
  - `double x = x + x`
  - `even x = x ‘mod’ 2 == 0`
- Which means we need to look at the equations for `+`, `mod` and `==` also.
- Plus we need to consider how to do this for any `x`
- Let’s start off more simply by proving things about structured types where we know the equations on the structured values.
Reasoning with Bool

• The data type Bool is easy to reason with because it has only two elements.

• To prove not (not b) = b for any boolean value b we simply consider the two cases one by one.

• The definition of not is

\[
\text{not} :: \text{Bool} \rightarrow \text{Bool} \\
\text{not True} = \text{False} \\
\text{not False} = \text{True}
\]

\[
\text{not (not True)} = \text{not (False)} = \text{True} \\
\text{not (not False)} = \text{not (True)} = \text{False}
\]
The problem with cases

- We must be wary when reasoning with cases though as Haskell’s semantics for defining functions are such that the order in which we write down the cases is significant.

\[
\text{isZero :: Int } \to \text{ Bool}
\]
\[
isZero 0 = \text{True}
\]
\[
isZero n = \text{False}
\]

- If we looked at these equations in isolation we might (incorrectly) conclude that \(\text{isZero 0} = \text{False}\).

- We need to interpret cases by conjoining the negation of the previous cases guards to them.

\[
\text{isZero :: Int } \to \text{ Bool}
\]
\[
isZero 0 | \text{True} = \text{True}
\]
\[
isZero n | n \neq 0 = \text{False}
\]

Non-overlapping cases are suitable for reasoning with
A simple List example

• Let’s look at a simple example of reasoning over the List type.
• Remember the function reverse?

\[
\text{reverse} :: [a] \rightarrow [a] \\
\text{reverse} \; [] = [] \\
\text{reverse} \; (x:xs) = \text{reverse} \; xs \; ++ \; [x]
\]

• Let’s prove that \( \text{reverse} \; xs = xs \) for any singleton list \( xs \).

\[
\begin{align*}
\text{reverse} \; [x] & \\
& = \{ \text{syntactic sugar} \} \\
& \text{reverse} \; (x:\;[]) \\\n& = \{ \text{defn of reverse} \} \\
& \text{reverse} \; [] \; ++ \; [x] \\
& = \{ \text{defn of ++} \} \\
& [x] \\
& \iff \text{reverse} \; [x] = [x]
\end{align*}
\]
Really?

• I said above that we had proved reverse \( xs = xs \) for any singleton list.

• I actually proved reverse \([x] = [x]\) for some variable \(x\).

• The logical principle of *generalisation* says that this is fine though if there are no assumptions made about the variable \(x\).
  • Because \(x\) can be instantiated by any value, and the equation is true for \(x\), then the equation is true for any value.

• Can we extend this idea to prove something more complicated on lists?

• How about \(\text{reverse ( reverse } xs ) = xs\) for any list \(xs\)?

• We can, but first we’ll need to take a little diversion down induction alley.
Proof by induction

• We have learned already about proof by induction over natural numbers:
  • To prove $P(n)$ for all $n$, we prove $P(0)$ and we then prove for any $k$ that $P(k+1)$ holds under the assumption that $P(k)$ holds.
• What is not obvious from this is that this principle uses the fact that natural numbers are a Structured Data Type.
• Let’s define them in Haskell

```haskell
data Nat = Zero | Succ Nat
```

• The values of this type are

$0, 1, 2, 3, \ldots$
Induction on type Nat

- Let’s restate the induction principle for values of the data type Nat:
  - To prove $P(n)$ holds for all values of type Nat, we prove:
    - $P(\text{Zero})$
    - $P(\text{Succ } x)$ under the assumption that $P(x)$
  - The semantics of algebraic data types are such that this principle is valid - this covers all values of type Nat.
- Example, define the add function

```haskell
add :: Nat -> Nat -> Nat
add Zero m = m
add (Succ n) m = Succ (add n m)
```

Let’s prove that add is associative.
Associativity of add

We need to show, for all \(x, y, z\),

\[
\text{add } x \ (\text{add } y \ z) = \text{add} \ (\text{add } x \ y) \ z
\]

We will use induction on the \(x\) variable and choose any \(y, z\)

**Base Case (Zero)**

\[
\text{add Zero } m = m
\]

\[
\text{add } (\text{Succ } n) \ m = \text{Succ} \ (\text{add } n \ m)
\]

**Inductive Case (Succ x)**

\[
\text{add } (\text{Succ } x) \ (\text{add } y \ z)
\]

\[
= \ {\text{defn of add}}
\]

\[
\text{Succ } (\text{add } x \ (\text{add } y \ z))
\]

\[
= \ {\text{inductive hypothesis}}
\]

\[
\text{Succ } (\text{add } (\text{add } x \ y) \ z)
\]

\[
= \ {\text{defn of add ‘backwards’}}
\]

\[
\text{add } (\text{Succ } (\text{add } x \ y) \ z)
\]

\[
= \ {\text{defn of add ‘backwards’}}
\]

\[
\text{add } (\ (\text{add } (\text{Succ } x) \ y) \ z)
\]
Induction on Structure

• There is nothing special about the Nat data type that allows induction
• We can use proof by induction on any structured data type
• Roughly stated, the principle goes like this
• To prove $P(x)$ for all values in type $T$ it is sufficient to prove
  • $P(C)$ for all constructors $C$ in $T$ that have no recursive arguments
  • $P(F x)$ assuming that $P(x)$ holds for all constructors $F$ in $T$ that have a single recursive argument
  • $P(G x y)$ assuming that $P(x)$ and $P(y)$ hold for all constructors $G$ in $T$ that have two recursive arguments
  • ...

$$P(x) \text{ holds for all } x \text{ if for all } C, F \text{ in } T \quad P(C) \text{ and } \quad P(x_i) \text{ for each recursive } x_i \quad \text{implies } P(F x_1 x_2 \ldots x_N)$$
Induction on Lists

- This is a very useful principle because it allows us to prove properties over any of the algebraic data types we define.

- Let’s consider Lists (data \([ a ] = \[ \] \mid (:) \ a \ [ a ]\))

  The List data type has one constructor \([\]\) and one constructor (:) whose second argument is recursive

- To prove a property P(xs) for all lists we must prove
  - P(\[\]) and
  - P(x : xs) under the assumption that P(xs)

- It is time to go back to our proof of

  reverse(reverse xs) = xs for all lists xs

That was a long diversion wasn’t it?
Proof that reverse is involutive

We need to show, for all xs

reverse ( reverse xs ) = xs

We will use induction on the list xs

**Base Case ([ ])**

reverse ( reverse [ ] )
= { defn of reverse }
reverse [ ]
= { defn of reverse}
[ ]

**Inductive Case (x:xs)**

reverse ( reverse (x:xs) )
= { defn of reverse }
reverse ( reverse xs ++ [x] )
= { hmmm question } 
reverse [x] ++ reverse (reverse xs)
= { reverse of singleton }
[x] ++ reverse (reverse xs)
= { inductive hypothesis}
[x] ++ xs
= (x:[]) ++ xs
= x : ( [] ++ xs)
= x : xs
all by defn of ++
The missing bit

• We need to resolve that missing step

\[
\text{reverse ( reverse xs ++ [x] )}
= \{ \text{hmmm ?????} \}
\text{reverse [x] ++ reverse (reverse xs)}
\]

• Let’s simplify it by replacing “reverse xs” with any old list.

\[
\text{reverse ( ys ++ [x] )}
= \{ \text{??????} \}
\text{reverse [x] ++ reverse (ys)}
\]

• And this suggests the following property

\[
\text{reverse ( ys ++ xs )}
= \{ \text{distributivity of reverse and ++} \}
\text{reverse xs ++ reverse (ys)}
\]

We’ll need to prove this!
Prove distributivity of reverse and ++

We need to show, for all $xs$, $ys$

We will use induction on the list $xs$ and choose any $ys$

**Base Case ($[]$)**

$reverse :: [a] \rightarrow [a]$

$reverse [] = []$

$reverse (x:xs) = reverse xs ++ [x]$

$((++)) \ [\] \ ys = ys$

$((++)) (x:xs) \ ys = x : ((++)) xs \ ys$

**Inductive Case (x:xs)**

$reverse ((x:xs) ++ ys)$

$= \{ \text{defn of ++ } \}$

$reverse (x : (xs ++ ys))$

$= \{ \text{defn of reverse } \}$

$reverse (xs++ys) ++ [x]$

$= \{ \text{inductive hypothesis } \}$

$(reverse ys ++ reverse xs) ++ [x]$

$= \{ \text{associativity of ++ } \}$

$reverse ys ++ (reverse xs ++ [x])$

$= \{ \text{defn of reverse backwards } \}$

$reverse ys ++ reverse (x:xs)$

We’ll need to prove this!
Prove that map on lists is a functor

Recall the functor laws from a previous lecture?

**Functor Law 1**: \( \text{map id} = \text{id} \)

**Functor Law 2**: \( \text{map (g . h)} = \text{map g . map h} \)

Let’s show that these hold for map on lists

\[
\begin{align*}
\text{map } f \; [] &= [] \\
\text{map } f \; (x : \text{xs}) &= f \; x : \text{map } f \; \text{xs}
\end{align*}
\]

We need to use another reasoning principle: **extensionality**

Two functions are equal if they are equal whenever they are applied to the same arguments

For example, to prove the first functor law we need to prove

\[
\text{map id } \text{xs} = \text{id } \text{xs} \quad \text{for any } \text{xs}
\]
Proof of Functor Laws for Map

Law 1  \(\text{map id } xs = \text{id } xs\) for any \(xs\)

We use induction on the list \(xs\)

Base Case ([ ])

\[
\text{map id } [] = \{ \text{defn of map } \} \\
[] = \{ \text{defn of id backwards } \} \\
\text{id } []
\]

Inductive Case (x:xs)

\[
\text{map id } (x:x:xs) = \{ \text{defn of map } \} \\
\text{id } x : \text{map id } xs \\
= \{ \text{defn of id } \} \\
x : \text{map id } xs \\
= \{ \text{inductive hypothesis } \} \\
x : \text{id } xs \\
= \{ \text{defn of id } \} \\
x : xs \\
= \{ \text{defn of id backwards } \} \\
\text{id } (x:x:xs)
\]
Proof of Functor Laws for Map

Law 2 \[\text{map} \ (g \ . \ h) \ xs = (\text{map} \ g \ . \ \text{map} \ h) \ xs \text{ for any } xs\]

Firstly, we can use the definition of \((\cdot)\) to rewrite the equation as

\[\text{map} \ (g \ . \ h) \ xs = \text{map} \ g \ (\ \text{map} \ h \ xs) \text{ for any } xs\]

We prove this by induction over \(xs\)

**Base Case ([ ])**

\[
\text{map} \ (g \ . \ h) \ [] \\
= \quad \{ \text{defn of map} \} \\
\quad [] \\
= \quad \{ \text{defn of map backwards} \} \\
\text{map} \ g \ [] \\
= \quad \{ \text{defn of map backwards} \} \\
\text{map} \ g \ (\text{map} \ h \ [])
\]

**Inductive Case (x:xs)**

\[
\text{map} \ (g \ . \ h) \ (x:xs) \\
= \quad \{ \text{defn of map} \} \\
(g \ . \ h) \ x : \text{map} \ (g \ . \ h) \ xs \\
= \quad \{ \text{defn of (\cdot)} \} \\
g \ (h \ x) : \text{map} \ (g \ . \ h) \ xs \\
= \quad \{ \text{inductive hypothesis} \} \\
g \ (h \ x) : \text{map} \ g \ (\text{map} \ h \ xs) \\
= \quad \{ \text{defn of map backwards} \} \\
\text{map} \ g \ (\ h \ x : \text{map} \ h \ xs \ ) \\
= \quad \{ \text{defn of id backwards} \} \\
\text{map} \ g \ (\text{map} \ h \ (x:xs))
\]
Example Induction using Trees

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

Let’s prove

```
leaves t \leq 2^{\text{height } t} \text{ for all trees } t
```

```
height :: Tree \to \text{Int}
height (Leaf a) = 1
height (Node a l r) = 1 + \text{max} (\text{height } l) (\text{height } r)
```

```
leaves :: Tree \to \text{Int}
leaves (Leaf a) = 1
leaves (Node a l r) = (leaves l) + (leaves r)
```
Example Induction using Trees

We use induction on the tree $t$.

**Base Case (Leaf $a$)**

leaves (Leaf $a$)  
= 1  
< 2  
= $2^1$  
= $2^{\text{height (Leaf } a)}$

**Inductive Case (Node $a \ l \ r$)**

leaves (Node $a \ l \ r$)  
= leaves $l$ + leaves $r$  
< $2^{\text{height } l}$ + $2^{\text{height } r}$  
< $2 \cdot 2^{\max (\text{height } l) \ (\text{height } r)}$  
< $2^{1 + \max (\text{height } l) \ (\text{height } r)}$  
< $2^{\text{height (Node } a \ l \ r)}$

We're using lots of arithmetic properties in this proof. Strictly speaking these should be proved.
Help!

- These proofs written by hand seem unreasonably complicated.
- Especially as in the previous example where there are lots of minor properties of arithmetic etc required.
- For this reason, for reasoning proper we would use automation.
- There are loads of proof tools and Theorem Proving tools out in the wild that allow you to do these manipulations and will allow for automatic proof of minor properties.
- See for example, Isabelle, HOL, Coq, Agda, LEGO, NuPRL, PVS
- Agda is an interesting one to look at as it is based on Haskell.
YOUR QUESTIONS

Next Lecture:
Evaluation Order and Laziness