Equational reasoning

Then we can prove that
\[(x + a)(x + b) = x^2 + (a + b)x + ab\]
by using the earlier laws
\[(x + a)(x + b) = \]
\[xx + ax + xb + ab = \]
\[x^2 + ax + xb + ab = \]
\[x^2 + a(b + b) + ab = \]
\[x^2 + (a + b)x + ab \]

Please note that although
\[x(a + b) = xa + xb\]
The Lhs requires two arithmetic operations, while the rhs requires three.
That's why it is important.
Consider

double :: Int -> Int
double x = x + x

A function definition

But also

A property of a function!

So whenever you have `double x` you can write `x + x`.

But also whenever you have `x + x` you can write `double x`.

Applying and unapplying a function.

But be careful!

Consider

isZero :: Int -> Bool
isZero 0 = True
isZero n = False
Equational reasoning about Haskell

But be careful!
Consider

```haskell
isZero :: Int -> Bool
isZero 0 = True
isZero n = False
```

The first equation: bidirectional. The second: not so much! Why?

Because the order of expressions is significant: `isZero n` is replaced by `False` ONLY WHEN \( n \neq 0 \).

This effectively means:

```haskell
isZero :: Int -> Bool
isZero 0       = True
isZero n | n /= 0 = False
```

The guard ensures explicit presence of the condition.

This effectively means:

```haskell
isZero :: Int -> Bool
isZero 0       = True
isZero n | n /= 0 = False
```

The guard ensures explicit presence of the condition.

It also makes the equations \textit{independent of the order}!

Patterns independent of the order of checking are called \textit{non-overlapping}.

A good practice: use always non-overlapping patterns whenever possible.
Simple examples

A common example:

reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]

Using this definition we can show that reverse [x] = [x] for any value of x.

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

So changing reverse [x] to [x] does not change the meaning of a program, but changes its efficiency!

Another example:

not :: Bool -> Bool
not False = True
not True = False

Pattern matching in the definition forces case analysis on arguments. E.g. for not (not b) = b we need to separately consider False:

not (not False) =
not True = False

and then (similarly) True.
Induction on numbers

The simplest example of a recursive type:

data Nat = Zero | Succ Nat

meaning the only values are

Zero
Succ Zero
Succ (Succ Zero)
Succ (Succ (Succ Zero))
...

We will NOT consider infinite case, where you add
inf = Succ inf,
just finite natural numbers.

Proving a property \( p \) that holds for all elements of a recursive type (e.g. natural numbers above):

1. \( p \) Zero
2. If \( p \) \( n \) then necessarily \( p \) (Succ \( n \))

Mathematical induction.

Consider:

\[
\text{add} :: \text{Nat} \to \text{Nat} \to \text{Nat}
\]

add Zero m = m
add (Succ n) m = Succ (add n m)

Prove (by induction) that adding a Zero does not change a value.
Consider:

\[
\text{add} :: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \\
\text{add Zero m} = m \\
\text{add (Succ n) m} = \text{Succ (add n m)}
\]

Prove (by induction) that adding a Zero does not change a value.

Case 1: add Zero m = m

directly from the definition

Case 2: add n Zero = n

\[\text{Induction on numbers}\]

Induction applies to other enumerable types isomorphic with natural numbers, e.g. Haskell integers:

\[
\text{replicate} :: \text{Integer} \rightarrow a \rightarrow [a] \\
\text{replicate 0} _ = [] \\
\text{replicate n} x = x : \text{replicate (n-1) x}
\]

Property to show:
\[\text{length (replicate n x)} = n \text{ for all } n \geq 0.\]
**Induction on numbers**

Base case:

\[
\text{length } (\text{replicate } 0 \ x) = \\
\text{length } [] = \\
0
\]

Induction step:

\[
\text{length } (\text{replicate } (n+1) \ x) = \\
\text{length } (x : \text{replicate } n \ x) = \\
1 + \text{length } (\text{replicate } n \ x) = \\
1 + n = \\
n + 1
\]

QED

Note the active use of the induction hypothesis!

---

**Induction on lists**

Consider:

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

Let us prove:

\[
\text{reverse } (\text{reverse } xs) = xs
\]
Induction on lists

Base case:
\[
\text{reverse (reverse [])} = \\
\text{reverse []} = \\
[]
\]

Inductive case:
\[
\text{reverse (reverse (x:xs))} = \\
\text{reverse (reverse xs ++ [x])} = \\
\text{reverse [x] ++ reverse (reverse xs)} = \\
[x] ++ reverse (reverse xs)) = \\
x : xs
\]

We have used a lemma: the distributivity of reverse over append:
\[
\text{reverse (xs ++ ys) = reverse ys ++ reverse xs}
\]

Jacek Malec, http://rss.cs.lth.se

Induction on lists

Base case (because ++ is defined by pattern matching over the first argument):
\[
\text{reverse ([] ++ ys)} = \\
\text{reverse ys} = \\
\text{reverse ys ++ []} = \\
\text{reverse ys ++ reverse []}
\]

Inductive case:
\[
\text{reverse ((x:xs) ++ ys)} = \\
\text{reverse (x : (xs ++ ys))} = \\
\text{reverse (xs ++ ys) ++ [x]} = \\
(reverse ys ++ reverse xs) ++ [x] = \\
\text{reverse ys ++ (reverse xs ++ [x])} = \\
\text{reverse ys ++ reverse (x:xs)}
\]

QED

Jacek Malec, http://rss.cs.lth.se
Induction on lists

Remember functor laws:
  \( \text{fmap id} = \text{id} \)
  \( \text{fmap} \ (g \ . \ h) = \text{fmap} \ g \ . \ \text{fmap} \ h \)

We can verify them using induction over lists (or, more generally, over recursive data structures, or functor types), where \( \text{fmap} \) is meaningful.

We use
  \( \text{fmap} :: (a \to b) \to [a] \to [b] \)
  \( \text{fmap} \ [] = [] \)
  \( \text{fmap} \ g \ (x:xs) = g \ x : \text{fmap} \ g \ xs \)

Whiteboard: show the first law.

Exercise: prove the second law.

Making append vanish

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

Complexity?
Making append vanish

reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]

Complexity?
++ linear with respect to the first argument, thus
reverse is quadratic wrt to the length of its argument.

How to improve it?

The trick: define a more general function reverse' combining the
behaviour of reverse and ++, so that always

reverse' xs ys = reverse xs ++ ys

Then reverse would just become

reverse xs = reverse' xs []

Constructing reverse'

Let's verify the equation by induction on xs.
Base case:
reverse' [] ys =
reverse [] ++ ys =
[] ++ ys =
ys

Inductive case:
reverse' (x:xs) ys =
reverse (x:xs) ++ ys =
(reverse xs ++ [x]) ++ ys =
reverse xs ++ ([x] ++ ys) =
reverse' xs ([x] ++ ys) =
reverse' xs (x:ys)
Constructing reverse'

From the construction we can conclude that

```haskell
reverse' :: [a] -> [a] -> [a]
reverse' [] ys = ys
reverse' (x:xs) ys = reverse' xs (x:ys)
```

suffices to show by induction that

```haskell
reverse' xs ys = reverse xs ++ ys
```

As the definition does not use `reverse`, we can redefine it as

```haskell
reverse :: [a] -> [a]
reverse xs = reverse' xs []
```

Complexity? Linear!

Moa Johansson @ Chalmers.