Equational reasoning

We can prove that

\[(x + a)(x + b) = x^2 + (a + b)x + ab\]

by using the earlier laws:

\[(x + a)(x + b) = \]
\[xx + ax + xb + ab = \]
\[x^2 + ax + xb + ab = \]
\[x^2 + ax + bx + ab = \]
\[x^2 + (a + b)x + ab\]

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

\[ \text{double} :: \text{Int} \rightarrow \text{Int} \]
\[ \text{double } x = x + x \]

A function \textit{definition}

But also

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

But be careful!
Consider

A property of a function!

So whenever you have \text{double } x you can write \( x + x \).

But also

whenever you have \( x + x \) you can write \text{double } x.

Applying and \textit{unapplying} a function.
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 ≠ 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 independent of the order!

Patterns independent of the order of checking are called non-overlapping.

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

```haskell
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):

- $p$ Zero
- If $p$ $n$ then necessarily $p$ (Succ $n$)

Mathematical induction.

---

Consider:

```
add :: Nat -> Nat -> 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:

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

Prove (by induction) that adding a Zero does not change a value.
Case 1: \(\text{add Zero m} = \text{m}\)
   directly from the definition
Case 2: \(\text{add n Zero} = \text{n}\)

Case 2: \(\text{add n Zero} = \text{n}\)

**base case:**

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

**inductive step:**

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

QED. \(\Box\) vsv.

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

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

Property to show:

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

Base case:
length (replicate 0 x) =
length [] =
0

Induction step:
length (replicate (n+1) x) =
length (x : replicate n x) =
1 + length (replicate n x) =
1 + n =
n + 1
QED
Note the active use of the induction hypothesis!

Induction on lists

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

Let us prove:
reverse (reverse xs) = xs
**Induction on lists**

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

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

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

---

**Induction on lists**

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

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

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

---

**Induction on lists**

**Base case:**
- \(\text{reverse ([ ] ++ ys)} =\)
- \(\text{reverse ys} =\)
- \(\text{reverse ys ++ [ ]} =\)
- \(\text{reverse ys ++ reverse [ ]}\)

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

QED
Verification

Induction on lists

Remember functor laws:

\[ \text{fmap id} = \text{id} \]
\[ \text{fmap (g \ . \ h)} = \text{fmap g \ . \ fmap h} \]

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

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

Whiteboard: show the first law.

Exercise: prove the second law.

Making append vanish

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

Complexity?
Making append vanish

\[
\begin{align*}
\text{reverse} :: [a] & \rightarrow [a] \\
\text{reverse} \; [] &= [] \\
\text{reverse} \; (x:xs) &= \text{reverse} \; xs \; ++ \; [x]
\end{align*}
\]

Complexity?

(++): linear with respect to the first argument, thus \text{reverse} is quadratic wrt the length of its argument.

How to improve it?

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

\[
\text{reverse'} \; xs \; ys = \text{reverse} \; xs \; ++ \; ys
\]

Then reverse would just become

\[
\text{reverse} \; xs = \text{reverse'} \; xs \; []
\]
**Constructing reverse'**

From the construction we can conclude that

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

suffices to show by induction that

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

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

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

Complexity? Linear!

---

**Induction on tree-like types**

```
data Tree = Leaf Int | Node Tree Tree

flatten :: Tree -> [Int]
flatten (Leaf n) = [n]
flatten (Node l r) = flatten l ++ flatten r
```

Append makes it inefficient. Let's then do the trick again.

```
flatten' t ns = flatten t ++ ns
```

Now induction must work on branches instead of successors.
Verification
Constructing flatten'

Base case:

\[
\text{flatten'} (\text{Leaf } n) \text{ ns} = \\
\text{flatten} (\text{Leaf } n) ++ \text{ ns} = \\
[n] ++ \text{ ns} = \\
\text{n : ns}
\]

Inductive case:

\[
\text{flatten'} (\text{Node } l \text{ r}) \text{ ns} = \\
(\text{flatten } l ++ \text{ flatten } r) ++ \text{ ns} = \\
\text{flatten } l ++ (\text{flatten } r ++ \text{ ns}) = \\
\text{flatten'} l (\text{flatten } r ++ \text{ ns}) = \\
\text{flatten'} l (\text{flatten'} r \text{ ns})
\]

So the definition:

\[
\text{flatten'} :: \text{Tree} \rightarrow \text{[Int]} \rightarrow \text{[Int]} \\
\text{flatten'} (\text{Leaf } n) \text{ ns} = \text{n : ns} \\
\text{flatten'} (\text{Node } l \text{ r}) \text{ ns} = \text{flatten'} l (\text{flatten'} r \text{ ns})
\]
satisfies the specification we had for \text{flatten'}.

Finally we can define

\[
\text{flatten} :: \text{Tree} \rightarrow \text{[Int]} \\
\text{flatten } t = \text{flatten'} t []
\]
Again: much more efficient.

Verification

HipSpec: automating proofs

Moa Johansson @ Chalmers.