EDAN40: Functional Programming
On Program Verification

Jacek Malec
Dept. of Computer Science, Lund University, Sweden
May 18th, 2017

Equational reasoning

Then we can prove that

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

by using the earlier laws

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

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

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.
But be careful!
Consider

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

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

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

This effectively means:

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

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.
The simplest example of a recursive type:

\[
data \text{Nat} = \text{Zero} \mid \text{Succ Nat}
\]

meaning the only values are

- \text{Zero}
- \text{Succ Zero}
- \text{Succ (Succ Zero)}
- \text{Succ (Succ (Succ Zero))}
- ...

We will NOT consider infinite case, where you add

\[
\text{inf} = \text{Succ inf}
\]

just \text{finite} natural numbers.

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

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

Mathematical induction.

Consider:

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

\[
\begin{align*}
\text{add Zero m} &= m \\
\text{add (Succ n) m} &= \text{Succ (add n m)}
\end{align*}
\]

Prove (by induction) that adding a \text{Zero} does not change a value.
**Verification**

**Induction on numbers**

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.

Case 1: add Zero m = m
directly from the definition

Case 2: add n Zero = n

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

**12(23)**

---

**Induction on numbers**

**Case 2:** add n Zero = n

**base case:**

add Zero Zero = Zero

**inductive step:**

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

QED. □ vsv.

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

**13(23)**

---

**Induction on numbers**

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

replicate :: Integer -> a -> [a]
replicate 0 _ = []
replicate n x = x : replicate (n-1) x

**Property to show:**

length (replicate n x) = n for all n ≥ 0.

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

**14(23)**
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 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:
\[
\text{reverse} :: [a] \rightarrow [a] \\
\text{reverse} [] = [] \\
\text{reverse} (x:xs) = \text{reverse} xs ++ [x]
\]

Let us prove:
\[
\text{reverse} (\text{reverse} xs) = xs
\]
Induction on lists

Base case:
\[ \text{reverse} \left( \text{reverse} \, \emptyset \right) = \]
\[ \text{reverse} \, \emptyset = \emptyset \]

Inductive case:
\[ \text{reverse} \left( \text{reverse} \, (x:xs) \right) = \]
\[ \text{reverse} \, (x:xs) = \]
\[ x : xs \]

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

Induction on lists

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

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

QED
Induction on lists

Remember functor laws:

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

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

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

Whiteboard: show the first law.

Exercise: prove the second law.

---

Making append vanish

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

Jacek Malec, http://rss.cs.lth.se
20(23)

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)

Jacek Malec, http://rss.cs.lth.se
21(23)
From the construction we can conclude that

\[
\text{reverse'} :: [a] \rightarrow [a] \rightarrow [a]
\]

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

\[
\text{reverse'} \; (x:xs) \; ys = \text{reverse'} \; xs \; (x:ys)
\]

suffices to show by induction that

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

As the definition does not use \texttt{reverse}, we can redefine it as

\[
\text{reverse} :: [a] \rightarrow [a]
\]

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

Complexity? Linear!