EDAN40: Functional Programming
On Program Verification

Jacek Malec
Dept. of Computer Science, Lund University, Sweden

May 15th, 2023

Equational reasoning

$$xy = yx$$

$$x + (y + z) = (x + y) + z$$

$$x(y + z) = xy + xz$$

$$(x + y)z = xz + yz$$

Equational reasoning

Then we can prove that

$$(x + a)(x + b) = x^2 + (a + b)x + ab$$

by using the earlier laws

$$(x + a)(x + b) =$$

$$xx + ax + xb + ab =$$

$$x^2 + ax + xb + ab =$$

$$x^2 + ax + bx + ab =$$

$$x^2 + (a + b)x + ab$$

Equational reasoning

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.

Equational reasoning about Haskell

Consider

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

A function *definition*

Equational reasoning about Haskell

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`.

Equational reasoning about Haskell

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.

Equational reasoning about Haskell

But be careful!
Consider

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

Equational reasoning about Haskell

But be careful!
Consider

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

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

Equational reasoning about Haskell

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 $n \neq 0$.

Equational reasoning about Haskell

This effectively means:

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

The guard ensures explicit presence of the condition.

Equational reasoning about Haskell

This effectively means:

```
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.

Simple examples

A common example:

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

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!

Simple examples

Another example:

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

Simple examples

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))
...
```

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.

Induction on 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.

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.

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`

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.

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
```

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 $n \geq 0$.

Induction on numbers

Base case:

```
length (replicate 0 x) =
length [] =
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:

```
reverse (reverse []) =
reverse [] =
[]
```

Induction on lists

Base case:

```
reverse (reverse []) =
reverse [] =
[]
```

Inductive case:

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

Induction on lists

Base case:

```
reverse (reverse []) =
reverse [] =
[]
```

Inductive case:

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

We have used a *lemma*: the distributivity of `reverse` over `append`:

```
reverse (xs ++ ys) = reverse ys ++ reverse xs
```

Induction on lists

Base case (because ++ is defined by pattern matching over the first argument):

```
reverse ([] ++ ys) =
reverse ys =
reverse ys ++ [] =
reverse ys ++ reverse []
```

Induction on lists

Base case (because ++ is defined by pattern matching over the first argument):

```
reverse ([] ++ ys) =
reverse ys =
reverse ys ++ [] =
reverse ys ++ reverse []
```

Inductive case:

```
reverse ((x:xs) ++ ys) =
reverse (x : (xs ++ ys)) =
reverse (xs ++ ys) ++ [x] =
(reverse ys ++ reverse xs) ++ [x] =
reverse ys ++ (reverse xs ++ [x]) =
reverse ys ++ reverse (x:xs)
```

QED

Induction on lists

Remember functor laws:

```
fmap id = id
fmap (g . h) = fmap g . fmap h
```

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

Induction on lists

Remember functor laws:

```
fmap id = id
fmap (g . h) = fmap g . fmap h
```

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

We use

```
fmap :: (a -> b) -> [a] -> [b]
fmap g [] = []
fmap g (x:xs) = g x : fmap g xs
```

Whiteboard: show the first law.

Induction on lists

Remember functor laws:

```
fmap id = id
fmap (g . h) = fmap g . fmap h
```

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

We use

```
fmap :: (a -> b) -> [a] -> [b]
fmap g [] = []
fmap g (x:xs) = g x : 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?

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:

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

```
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.

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.

Constructing flatten’

Base case:

```
flatten’ (Leaf n) ns =
flatten (Leaf n) ++ ns =
[n] ++ ns =
n : ns
```

Constructing flatten’

Base case:

```
flatten’ (Leaf n) ns =
flatten (Leaf n) ++ ns =
[n] ++ ns =
n : ns
```

Inductive case:

```
flatten’ (Node l r) ns =
(flatten l ++ flatten r) ++ ns =
flatten l ++ (flatten r ++ ns) =
flatten’ l (flatten r ++ ns) =
flatten’ l (flatten’ r ns)
```

Constructing flatten’

So the definition:

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

satisfies the specification we had for `flatten’`.

Constructing flatten’

So the definition:

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

satisfies the specification we had for `flatten'`.
Finally we can define

```
flatten :: Tree -> [Int]
flatten t = flatten' t []
```

Again: much more efficient.

HipSpec: automating proofs

Moa Johansson @ Chalmers.