Lazy evaluation

Slides based on: Graham Hutton. Programming in Haskell chapter 12

Evaluation

```
inc :: Int -> Int
inc n = n + 1
```

```
inc (2 * 3)
= { applying * }
inc 6
= { applying inc }
6 + 1
= { applying + }
7
```

```
inc (2 * 3)
= { applying inc }
(2 * 3) + 1
= { applying * }
6 + 1
= { applying + }
7
```

Order of evaluation does not matter for terminating programs!

Imperative version

```
n + (n := 1)
= { applying n }
0 + (n := 1)
= { applying := }
0 + 1
= { applying + }
1
```

```
n + (n := 1)
= { applying := }
n + 1
= { applying n }
1 + 1
= { applying + }
2
```

Redex

(reducible expression)

```
mult :: (Int, Int ) → Int
mult (x , y) = x * y
```

```
mult (1 + 2, 2 + 3)
```

1 2 3

Innermost Evaluation

```
mult (1 + 2,2 + 3)
= { applying the first + }
mult (3,2 + 3)
= { applying + }
mult (3, 5)
= { applying mult }
3 * 5
= { applying * }
15
```

Outermost Evaluation

```
mult (1 + 2,2 + 3)
= { applying mult }
(1 + 2) * (2 + 3)
= { applying the first + }
3 * (2 + 3)
= { applying + }
3 * 5
= { applying * }
15
```

Back to the basics

```
mult :: Int → Int → Int
mult x = λy → x * y
```

```
mult (1 + 2) (2 + 3)
= { applying the first + }
mult 3 (2+3)
= { applying mult }
(λy → 3 * y) (2+3)
= { applying + }
(λy → 3 * y) 5
= { applying λy → 3 * y }
3 * 5
= { applying * }
15
```

Reduction under lambda

```
(λx → 1 + 2) 0
= { applying λx → 1 + 2 }
1 + 2
= { applying + }
3
```

"Using innermost and outermost evaluation, but not under lambdas, is referred to as call-by-value and call-by-name evaluation, respectively"

Termination properties

```haskell
inf :: Int
inf = 1+inf
```

Evaluation

```
inf
= { applying inf }
1 + inf
= { applying inf }
1 + (1+inf )
= { applying inf }
1 + (1 + (1 + inf ))
= { applying inf }
...
```

Termination properties

```
fst (0, inf )
= { applying inf }
fst (0,1 + inf )
= { applying inf }
fst (0,1 + (1+inf ))
= { applying inf }
fst (0,1 + (1+(1+inf )))
= { applying inf }
```

call-by-value

```
fst (0, inf )
= { applying fst }
0
```

call-by-name

"call-by-name evaluation is preferable to call-by-value for the purpose of ensuring that evaluation terminates as often as possible"

Number of reductions

```
square :: Int → Int
square n = n * n
```

call-by-value

```
square (1 + 2)
= { applying + }
square 3
= { applying square }
3 * 3
= { applying * }
9
```

call-by-name

```
square (1 + 2)
= { applying square }
(1 + 2) * (1 + 2)
= { applying the first + }
3 * (1 + 2)
= { applying + }
3 * 3
= { applying * }
9
```

"Arguments are evaluated precisely once using call-by-value evaluation, but may be evaluated many times using call-by-name"

Sharing

```
square (1 + 2)
= { applying square }

p1 * p2  1 + 2
= { applying + }

p1 * p2      3
= { applying * }
```

Definition Lazy Evaluation

"The use of call-by-name evaluation in conjunction with sharing is called **lazy evaluation**"

Infinite Structures

```
ones :: [Int ]
ones = 1:ones
```

```
head ones
= { applying ones }
head (1 : ones)
= { applying head }
1
```

Under lazy evaluation ones is **not** an infinite list as such, but rather **a potentially infinite list**, which is only evaluated as much as required by the context.

Calculating Prime Numbers

2	3	4	5	6	7	8	9	10	11	12	13	14	15	…
	3		5		7		9		11		13		15	…
			5		7				11		13			…
					7				11		13			…
									11		13			…
											13			…

In Haskell

```
primes :: [Int ]
primes = sieve [2 ..]


sieve :: [Int ] → [Int ]
sieve (p : xs) = p : sieve [x | x ← xs, x ‘mod‘ p /= 0]
```

Strict ! Evaluation

```
square $! (1 + 2)
= { applying + }
square $! 3
= { applying $! }
square 3
= { applying square }
3 * 3
= { applying * }
9
```

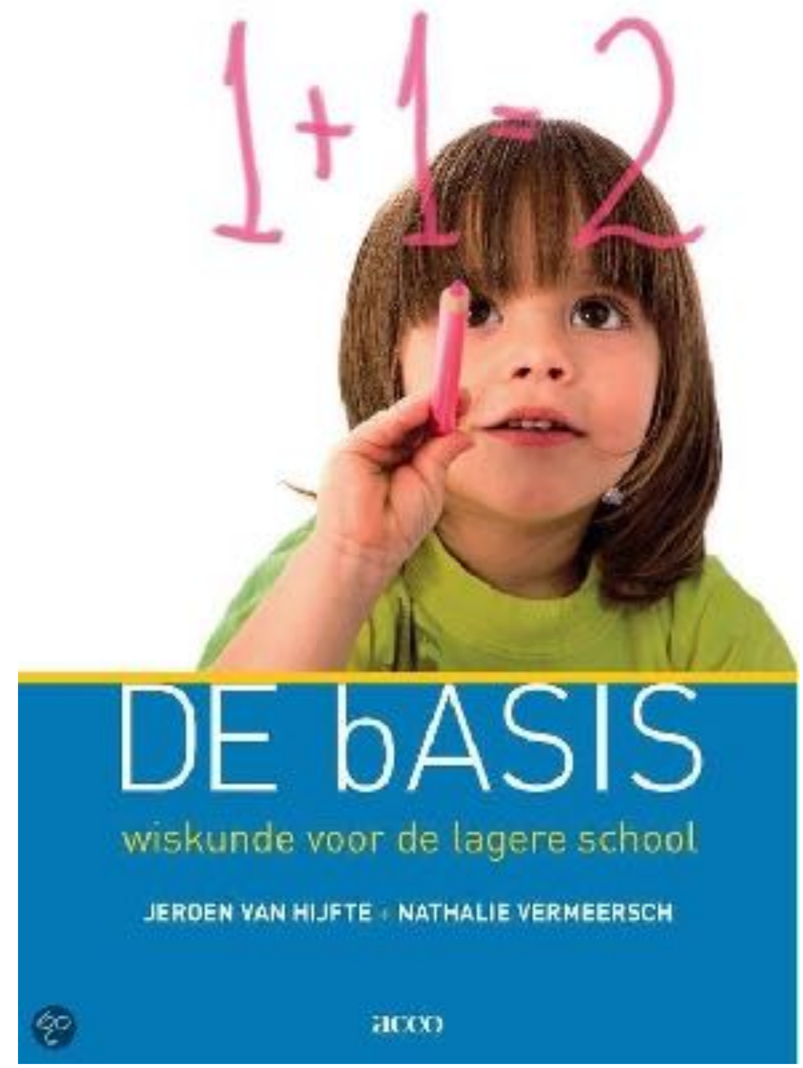
Equational Reasoning

Christophe Scholliers

Slides adopted from Prof. Dr. Wolfgang de Meuter

Lagere School

a	$(a+2)\cdot(a+3)$	a^2+5a+6
1	$(1+2)\cdot(1+3) = 3\cdot 4 = 12$	$1+5+6 = 12$
2	$(2+2)\cdot(2+3) = 4\cdot 5 = 20$	$4+10+6 = 20$
3	$(3+2)\cdot(3+3) = 5\cdot 6 = 30$	$9+15+6 = 30$
4	$(4+2)\cdot(4+3) = 6\cdot 7 = 42$	$16+20+6 = 42$
5	$(5+2)\cdot(5+3) = 7\cdot 8 = 56$	$25+25+6 = 56$
6	$(6+2)\cdot(6+3) = 8\cdot 9 = 72$	$36+30+6 = 72$
7	$(7+2)\cdot(7+3) = 9\cdot 10 = 90$	$49+35+6 = 90$
8	$(8+2)\cdot(8+3) = 10\cdot 11 = 110$	$64+40+6 = 110$
9	$(9+2)\cdot(9+3) = 11\cdot 12 = 132$	$81+45+6 = 132$



Equals = Equals

Expand $(x+y)(z+q)$: $xz+xq+yz+yq$

Steps

$(x+y)(z+q)$

Distribute parentheses using: $(a+b)(c+d)=ac+ad+bc+bd$

$=xz+xq+yz+yq$

Equals !

```
       x*y  =  y*x
x+(y+z)     =  (x+y)+z
x*(y+z)     =  x*y+x*z
(x+y)*z     =  x*z+y*z
```

Functional Programming

```
double x = x + x

double e <->  e+e
```

```
isZero 0 = True
isZero n = False
```

“Better” definition

```
isZero 0 = True
isZero n = False  n!=0
```

Order independent

Neutral element of reverse

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

Proof
reverse (x:[])


…

[x]
```

Neutral element of reverse

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

Proof
reverse (x:[])
= {unfold reverse/2}
(reverse []) ++ [x]
= {reverse [] }
[] ++ [x]
= {definition of ++ }
[x]
```

Induction

```haskell
data Nat = Zero | Succ Nat

Zero
Succ Zero
Succ (Succ Zero)
```

Induction over Nat

```
P(Zero)
P(n)  => P(Succ n)
---------------------
∀(n) . P(n)
```

Example

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

```
add n Zero = n   ?
```

Step 1

```
P(n) := add n Zero = n
```

Base case

```
P(Zero) :=
add Zero Zero
=
Zero
```

Inductive step

```
IH   := add n Zero        = n
Goal := add (Succ n) Zero = (Succ n)
```

```
add (Succ n) Zero
= {add 2}
Succ (add n Zero)
= {IH}
Succ n
```

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

Exercise

```
add x (add y z) = add (add x y) z
```

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

Base case

```
add Zero (add y z)
=
add y z

add (add Zero y) z
=
add y z
```

```
IH :=  add n (add y z) = add (add n y) z

Goal := add (Succ n) (add y z) = add (add (Succ n) y) z
```

```
add (Succ n) (add y z) =
= {Def}
Succ (add n (add y z)
= {IH}
Succ (add (add n y) z)



add (add (Succ n) y) z
= {Def}
add Succ (add n y) z
= {Def}
Succ (add (add n y) z)
```

Replicate

```
replicate 0 x       = []
replicate (n+1) x = x : replicate n x
```

```
length (replicate n x)  = n
```

Base case

```
length (replicate 0 x)
= {def replicate}
length []
= {def length}
0
```

Inductive step

```
length(replicate (n+1) x)
= {def replicate}
length (x:replicate n x)
= {def length}
1+length(replicate n x)
= {IH }
1+n
```

Induction over lists

```
P([])
P(xs) => P(x:xs)
---------------------
∀(ys) P(ys)
```

Reverse$_2$

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

```haskell
reverse (reverse (xs)) = xs
```

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

Lemma

Lemma

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

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

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

Lemma

Lemma

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

```
reverse ((y:ys)++[x])
= {def ++ 2}
reverse (y:(ys++[x]))
= {def reverse 2}
reverse (ys++[x]) ++ [y]
= {ind hyp}
(x:reverse ys) ++ [y]
= {def ++ 2}
x:(reverse ys ++ [y])
= {def reverse 2}
x:reverse(y:ys)
```

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

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

Reverse proof

```
reverse (reverse (xs)) = xs
```

```
reverse (reverse [])
= reverse [] {def reverse 1}
= [] {def reverse 1}
```

P([])

```
reverse (reverse (x:xs))
= reverse (reverse xs++[x]) {def reverse 2}
= x:reverse(reverse xs)     {lemma}
= x:xs {ind hyp}
```

P(xs) →P (x:xs)

Deriving implementations

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

reverse’ xs ys = reverse xs ++ ys

reverse xs = reverse’ xs []

Deriving implementations

reverse' xs ys = reverse xs ++ ys

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

Base case:

```
reverse' [ ] ys
= { specification of reverse' }
reverse [ ] ++ ys
= { applying reverse }
[ ] ++ ys
= { applying ++ }
ys
```

reverse’ xs ys = reverse xs ++ ys

Inductive case:

```
 reverse’ (x : xs) ys
= { specification of reverse’ }
 reverse (x : xs) ++ ys
= { applying reverse }
(reverse xs ++ [x]) ++ ys
= { associativity of ++ }
reverse xs ++ ([x] ++ ys)
= { induction hypothesis }
reverse’ xs ([x] ++ ys)
= { applying ++ }
reverse’ xs (x : ys)
```

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

Look mum no append !

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

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

Tree Recursion

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

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

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

Inductive case:

```
flatten' (Node l r) ns
= { specification of flatten' }
(flatten l ++ flatten r) ++ ns
= { associativity of ++ }
flatten l ++ (flatten r ++ ns)
= { induction hypothesis for l }
flatten' l (flatten r ++ ns)
= { induction hypothesis for r }
flatten' l (flatten' r ns)
```

Base case:

```
flatten' (Leaf n) ns
= { specification of flatten' }
flatten (Leaf n) ++ ns
= { applying flatten }
[n] ++ ns
= { applying ++ }
n : ns
```

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

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

Flatten

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

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

Conclusion

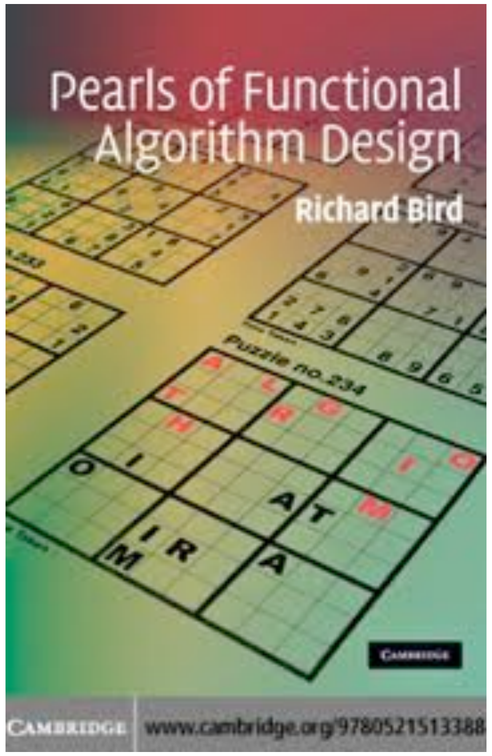
- *Equational reasoning can be an elegant way to prove properties of a program.*

- *Equational reasoning can be used to establish a relation between an "obviously correct" Haskell program (a inefficient specification) and an efficient Haskell program.*

- *Equational reasoning is usually quite lengthy.*

- ***Careful with special cases (laziness): undefined values + infinite values***

Further Reading




FUNCTIONAL PEARL

A Pointless Derivation of Radix Sort