CS 457/557: Functional Languages

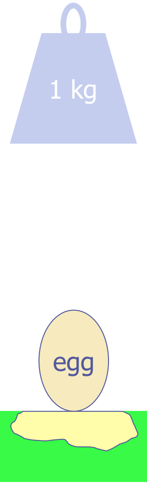
Equational Reasoning: Algebra of Programming

Mark P Jones
Portland State University

What Makes a Good Program?

- Performance?
- Code size?
- Maintainability?
- Above all else, correctness!
- But what does that mean? How can it be established?

Testing:

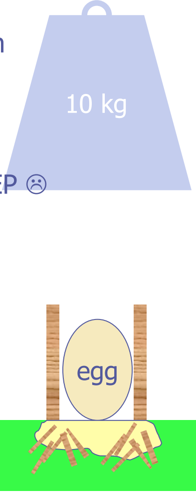


- Tests confirm expectations about the way things work
- If you drop a weight …
- … onto an egg …
- … Scrambled Egg!

Testing:

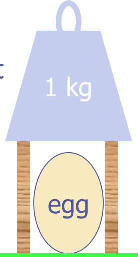
- Suppose it's our job to **protect** eggs from falling weights …
- We might design an EP (**E**gg **P**rotector™) to accomplish this …
- Then we test again …
- Hooray! The egg is safe! ☺

Generalizing from Tests:



- "The EP will protect an egg from a falling weight"
- Scrambled egg, and a crushed EP ☹ How embarrassing …
- It can be **dangerous** to generalize from the results of testing!

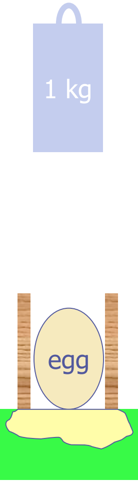
Refining the **claim**:



- Think back to our test:
- "The EP will protect an egg from a falling weight of at most 1kg"
- This isn't such a general statement …
- … but it describes the EP's properties more accurately

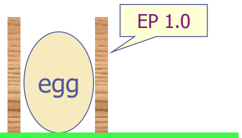
More Tests:

- "The EP will protect an egg from a falling weight of at most 1kg"
- Oops, another embarrassing oversight!



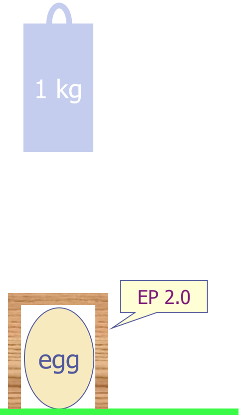
Refining the **EP Design**:

- "The EP will protect an egg from a falling weight of at most 1kg"



Refining the **EP Design**:

- "The EP 2.0 will protect an egg from a falling weight of at most 1kg"
- We had to change the design of the EP …
- But our egg is safe again!

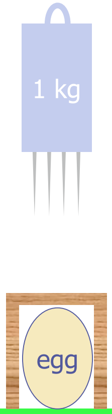


Or is it?

- We'd like the EP to protect **any** egg …

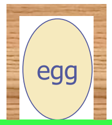
Or is it?

- We'd like the EP to protect **any** egg …
- … from **any** weight …



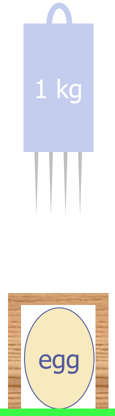
General Observations:

- Testing helps us to find (and then avoid):
  - bugs in the things that we build
  - bugs in the claims that we make about them
- Testing and Development working together …
- But …



Testing has Limits:

- "testing can be used to show the presence of bugs, but never to show their absence" [Edsger Dijkstra, 1969]
- To be **absolutely certain** that the EP 2.0 will protect **any** egg from **any** weight under 1kg, we will need to **prove it**.



Equational Reasoning:

- Functional Languages are Good for Equational Reasoning (Gofer!)
- Much of what follows is inspired by the work of Richard Bird
- **Goal**: to prove laws of the form $e_1=e_2$ relating program fragments e_1 and e_2
- **Goal**: to calculate/synthesize efficient definitions of functions from clear, high-level specifications

---

Laws of Numbers:

If n is a natural number, then either:
n = 0; or
n = 1 + m for some (smaller) natural m

Functions on natural numbers:

$$0 + n = n$$
$$(1+m) + n = 1 + (m + n)$$

Does this look at all familiar?

+ is associative:

$\forall n.\ \forall p.\ \forall q.\ (n + p) + q = n + (p + q)$

If n = 0, then

$(n + p) + q$
$= (0 + p) + q$ (because n = 0)
$= p + q$ (definition of +)
$= 0 + (p + q)$ (definition of +)

+ is associative:

$\forall n.\ \forall p.\ \forall q.\ (n + p) + q = n + (p + q)$

If n = (1+m), then

$(n + p) + q$
$= ((1 + m) + p) + q$ (because n=1+m)
$= (1 + (m + p)) + q$ (definition of +)
$= 1 + ((m + p) + q)$ (definition of +)
$= 1 + (m + (p + q))$ (induction)
$= (1 + m) + (p + q)$ (definition of +)
$= n + (p + q)$ (definition of +)

+ is associative:

We've shown:
- The property holds for n = 0
- If the property holds for n = m, then it holds for n = (1 + m)
- So it holds for n = 1
- And for n = 2
- And for n = 3
- …

In fact, we've shown that it holds for all n:
$\forall n.\ \forall p.\ \forall q.\ (n + p) + q = n + (p + q)$

Laws of Numbers:

If n is a natural number, then either:
n = Zero; or
n = Succ m for some (smaller) natural m

data Nat = Zero | Succ Nat

Functions on natural numbers:
add Zero n = n
add (Succ m) n = Succ (add m n)

add is associative:

∀n. ∀p. ∀q. add (add n p) q = add n (add p q)

If n = Zero, then
add (add n p) q
= add (add Zero p) q (because n = Zero)
= add p q (definition of add)
= add Zero (add p q) (definition of add)

add is associative:

∀n. ∀p. ∀q. add (add n p) q = add n (add p q)

If n = Succ m, then
add (add n p) q
= add (add (Succ m) p) q (because n=1+m)
= add (Succ (add m p)) q (definition of +)
= Succ (add (add m p) q) (definition of +)
= Succ (add m (add p q)) (induction)
= add (Succ m) (add p q) (definition of +)
= add n (add p q) (definition of +)

add is associative:

We've shown:

- The property holds for n = Zero
- If the property holds for n = m, then it holds for n = Succ m
- So it holds for n = Succ Zero
- And for n = Succ (Succ Zero)
- And for n = Succ (Succ (Succ Zero))
- …

In fact, we've shown that it holds for all n:
∀n. ∀p. ∀q. add (add n p) q = add n (add p q)

Laws in Haskell:

We can apply these same ideas to many other Haskell datatypes, not just numbers

Algebra for programs:

- Break into cases (no junk, no confusion)
- Induction (recursion)
- Equational reasoning

Where do Laws come From?

Laws typically arise in one of three ways:

- From function definitions (with care)
  (x:xs) ++ ys = x : (xs ++ ys)
- From previously established laws
  map f . map g = map (f . g)
- From specifications of new functions
  sumSquares n = sum (map square [1..n])

Referential Transparency:

- The ability to replace equals with equals
  - If $e_1=e_2$, then $\ldots e_1 \ldots = \ldots e_2 \ldots$
- The inability to observe sharing
  - **let** x = e **in** (x,x) = (e, e)
  - **let** x = print 1 **in** (x,x) = (print 1, print 1)

Tools:

- Extensionality:
  - f = g ⇔ ∀x. f x = g x
- Simple substitution/instantiation:
  - From (f . g) x = f (g x), we can infer that ((1+) . (2*)) n = 1 + 2*n

… continued:

- Case analysis:
  - If xs :: [a], then xs = [], or xs = (y:ys) for some y and ys, or xs = ⊥
  - If b :: Bool, then b=False, b=True, or b=⊥
- Induction:
  - If property P(xs) holds for xs = [] and for xs = ⊥, and for (y:ys) whenever it holds for ys, then P(xs) holds for <u>all</u> lists xs.

Introducing Bottom, ⊥:

- We treat every type in Haskell as having a special element called bottom, written ⊥
- ⊥ represents the value produced by expressions that fail to terminate properly
  - Non-termination
  - Error (e.g., missing pattern matching case)
  - Explicit call of error "… message …"
- Called "bottom" because it has the least amount of information of any value

Strictness:

- We say that a function is <u>strict</u> if it is guaranteed to evaluate its argument.
- Another way to say this: f is strict if, and only if f ⊥ = ⊥
- Examples:
  - (1+) and not are both strict
  - (&&) and (||) are strict in their left arguments, but not in their right
  - map is strict in its list argument (but not the function)

Example:

- Suppose we specify:

```
f :: [Int] -> [Int]
f = map (1+)
```

- Now we can calculate:

```
  f []
= { by definition of f }
  map (1+) []
= { by definition of map}
  []
```

… continued:

- We can also calculate:

f (x:xs)
= { by definition of f }
map (1+) (x:xs)
= { by definition of map }
(1+x) : map (1+) xs
= { by definition of f }
(1 + x) : f xs

- Thus we have derived:

```
f         :: [Int] -> [Int]
f []      = []
f (x:xs)  = (1+x) : f xs
```

Associativity of (++):

Claim: xs++(ys++zs) = (xs++ys)++zs, for all xs, ys, and zs

Proof by induction on xs:

Base case: xs = []
[] ++ (ys ++ zs)
= { by definition of ++ }
ys ++ zs
= { by definition of ++ }
([] ++ ys) ++ zs

… continued:

Base case: xs = ⊥
lhs: ⊥ ++ (ys ++ zs)
= { ++ is strict in its first argument }
⊥

rhs: (⊥ ++ ys) ++ zs
= { ++ is strict in its first argument }
⊥ ++ zs
= { ++ is strict in its first argument }
⊥

… continued:

Inductive case: (x:xs)
(x:xs) ++ (ys ++ zs)
= { by definition of ++ }
x : (xs ++ (ys ++ zs))
= { by induction }
x : ((xs ++ ys) ++ zs)
= { by definition of ++ }
(x: (xs ++ ys)) ++ zs
= { by definition of ++ }
((x:xs) ++ ys) ++ zs

Fold Right:

A function from the prelude:
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (⊕) e $[x_0,x_1,x_2]$ = $x_0 \oplus (x_1 \oplus (x_2 \oplus e))$

Examples:
and = foldr (&&) True
concat = foldr (++) []

Definition:

```
foldr f e []      = e
foldr f e (x:xs)  = f x (foldr f e xs)
```

Fold Left:

A function from the prelude:
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl (⊕) e $[x_0,x_1,x_2]$ = $((e \oplus x_0) \oplus x_1) \oplus x_2$

Examples:
sum = foldl (+) 0
product = foldl (*) 1

Definition:

```
foldl f e []      = e
foldl f e (x:xs)  = foldl f (f e x) xs
```

Scan Left:

A function from the prelude:

scanl :: (a -> b -> a) -> a -> [b] -> [a]

scanl (⊕) e $[x_0,x_1,x_2]$

= [e, $e \oplus x_0$, $(e \oplus x_0) \oplus x_1$, $((e \oplus x_0) \oplus x_1) \oplus x_2$]

Specification:

```
scanl f e      = map (foldl f e) . inits

inits []       = [[]]
inits (x:xs)   = [] : map (x:) (inits xs)
```

Calculating scanl:

It is easy to derive scanl f e [] = [e]

For non empty lists:

```
scanl f e (x:xs)
= map (foldl f e) (inits (x:xs))
= map (foldl f e) ([] : map (x:) (inits xs))
= foldl f e [] : map (foldl f e) (map (x:) (inits xs))
= foldl f e [] : map (foldl f e . (x:)) (inits xs)
= e : map (foldl f (f e x)) (inits xs)
= e : scanl f (f e x) xs
```

Comparison:

- Specification:
  scanl f e = map (foldl f e) . inits
- Definition:
  scanl f e [] = [e]
  scanl f e (x:xs) = e : scanl f (f e x) xs
- The specification requires $O(n^2)$ applications of f on a list of length n while the definition uses only n applications for a list of the same length.
- But, in terms of the results that we obtain, we know that the two versions are equal!

Scan Right:

A dual of scanl:

```
scanr       :: (a -> b -> b) -> b -> [a] -> [b]
scanr f e    = map (foldr f e) . tails
```

scanr (⊕) e $[x_0, x_1, x_2]$

= $[x_0 \oplus (x_1 \oplus (x_2 \oplus e)), x_1 \oplus (x_2 \oplus e), x_2 \oplus e, e]$

More efficient version:

```
scanr f e []       = [e]
scanr f e (x:xs)   = f x (head ys) : ys
      where ys = scanr f e xs
```

Maximum Segment Sum:

- Given a sequence of numbers, find the subsegment whose sum is largest:
  - Example: maximal subsegment sum for the list [-1, 2, -3, 5, -2, 1, 3, -2, -2, -3, 6] is 7 (for the segment [5, -2, 1, 3])
- Simple solution:

```
mss :: [Int] -> Int
mss = maximum . map sum . segs
 where segs = concat . map inits . tails
```

- Not a great performer … $O(n^3)$

Calculate!

```
   mss
= {definition of mss}
   maximum . map sum . segs
```

Calculate!

mss

= {definition of segs}

maximum . map sum . concat . map inits . tails

Calculate!

mss

= {using map f . concat = concat . map (map f) }

maximum . concat . map (map sum) . map inits . tails

(map f . concat) [xs_1, xs_2, xs_3]
= map f (xs_1 ++ xs_2 ++ xs_3)
= map f xs_1 ++ map f xs_2 ++ map f xs_3

(concat . map (map f)) [xs_1, xs_2, xs_3]
= concat [map f xs_1, map f xs_2, map f xs_3]
= map f xs_1 ++ map f xs_2 ++ map f xs_3

Calculate!

mss

= { using map f . map g = map (f . g) }

maximum . concat . map (map sum . inits) . tails

(map f . map g) [x_1, x_2, x_3]
= map f [g x_1, g x_2, g x_3]
= [f (g x_1), f (g x_2), f (g x_3)]

map (f . g) [x_1, x_2, x_3]
= [(f . g) x_1, (f . g) x_2, (f . g) x_3]
= [f (g x_1), f (g x_2), f (g x_3)]

Calculate!

mss

= { the "bookkeeping law" }

maximum . map maximum . map (map sum . inits) . tails

maximum . concat
= maximum . map maximum

General form:
foldr f a . concat = foldr f a . map (foldr f a)
if f is associative with unit a

Calculate!

mss

= { Definition of scanl }

maximum . map maximum . map (scanl (+) 0) . tails

Definition:
scanl f e = map (foldl f e) . inits

Calculate!

mss

= {using map f . map g = map (f . g) }

maximum . map (maximum . scanl (+) 0) . tails

map f . map g = map (f . g)
(again ...)

Calculate!

mss

= { fold-scan fusion }

maximum . map (foldr f 0) . tails
  where f x y = max 0 (x + y)

We can prove that:
  maximum . scanl (+) 0 = foldr f 0
(A special case of a general property called “Fold-scan fusion”)

Calculate!

mss

= { definition of scanr }

maximum . scanr f 0
  where f x y = max 0 (x + y)

Definition of scanr:
  scanr f e = map (foldr f e) . tails

A simple, linear time algorithm,
courtesy of equational reasoning!

Calculate!

Remember:
  scanr (⊕) e $[x_0, x_1, x_2]$
  = $[x_0 \oplus (x_1 \oplus (x_2 \oplus e)),$
  $x_1 \oplus (x_2 \oplus e),$
  $x_2 \oplus e,$
  $e]$

mss

= { definition of scanr }

maximum . scanr f 0
  where f x y = max 0 (x + y)

```
mss xs = loop 0 0 (reverse xs)
 where
  loop m v []       = m
  loop m v (x:xs) = let y = max 0 (x+v)
                           in loop (max m y) y xs
```

This version of the definition is
not very intuitive … but we know
by construction that it is correct!

A Quick Check:

Just to be sure, let’s load these definitions i ugs
and quickly check to see if they are

Main> quickChec s)
OK, pa

To Be Continued …

Hmm, now that looks like another useful tool,
doesn’t it …

Summary:

- The ability to reason about code is essential if you care about its behavior (for example, in safety or security critical applications)
- Compilers rely on equivalences between program fragments to justify/validate some optimizations
- Functional Languages are Good for Equational Reasoning
- Referential transparency/lack of side effects makes reasoning more tractable
- It helps to build up a collection of laws and results that you can draw on in program verification or synthesis!