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 (Egg Protector™) 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 ...

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 +)

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 +)

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:
\[ \forall n. \forall p. \forall q. \text{add (add n p) q} = \text{add n (add p q)} \]

If n = Zero, then
  add (add n p) q
  = add (add Zero p) q  \text{(because n = Zero)}
  = add p q  \text{(definition of add)}
  = add Zero (add p q)  \text{(definition of add)}

add is associative:
\[ \forall n. \forall p. \forall q. \text{add (add n p) q} = \text{add n (add p q)} \]

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

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 $...e_1... = ...e_2...$

- 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:
  - $f = g \iff \forall x. f x = g x$

- Simple substitution/instantiation:
  - From $(f \cdot g) x = f (g x)$, we can infer that $((1+) \cdot (2^*)) n = 1 + 2^n$

... continued:

- Case analysis:
  - If $xs :: [a]$, then $xs = []$, or $xs = (y:ys)$ for some $y$ and $ys$, or $xs = \perp$
  - If $b :: Bool$, then $b=False$, $b=True$, or $b=\perp$

- Induction:
  - If property $P(xs)$ holds for $xs = []$ and for $xs = \perp$, and for $(y:ys)$ whenever it holds for $ys$, then $P(xs)$ holds for all lists $xs$.

Introducing Bottom, $\perp$:

- We treat every type in Haskell as having a special element called bottom, written $\perp$

- $\perp$ 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 strict if it is guaranteed to evaluate its argument.

- Another way to say this: $f$ is strict if, and only if $f \perp = \perp$

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

- Now we can calculate:
  
  ```haskell
  f []
  = \{ by definition of f \}
    map (1+) []
  = \{ by definition of map \}
    []
  ```
... continued:

- We can also calculate:
  \[ f(x:xs) \]
  = \{ by definition of \( f \) \}
  \[ \text{map}(1+)(x:xs) \]
  = \{ by definition of \text{map} \}
  \[ (1+x) : \text{map}(1+)xs \]
  = \{ by definition of \( f \) \}
  \[ (1+x) : fxs \]
- Thus we have derived:
  \( f \) :: [Int] -> [Int]
  \( f[] \) = []
  \( f(x:xs) \) = (1+x) : f xs

... continued:

**Base case:** \( xs = \bot \)

\( \text{lhs:} \) \( \bot \) ++ (ys ++ zs)

= \{ ++ is strict in its first argument \}

\( \bot \)

\( \text{rhs:} \) (\( \bot \) ++ ys) ++ zs

= \{ ++ is strict in its first argument \}

\( \bot \) ++ zs

= \{ ++ is strict in its first argument \}

\( \bot \)

... 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:

\( \text{foldr} :: (a -> b -> b) -> b -> [a] -> b \)

\( \text{foldr} (\oplus) \) \( e \times x_{y_1}x_jx_2 \) = \( x_y \oplus (x_1 \oplus (x_2 \oplus e)) \)

Examples:

- \( \text{and} = \text{foldr} (\&\&) \) True
- \( \text{concat} = \text{foldr} (++) [] \)

Definition:

\( \text{foldr} f e [] = e \)

\( \text{foldr} f e (x:xs) = f x (\text{foldr} f e xs) \)

Fold Left:

A function from the prelude:

\( \text{foldl} :: (a -> b -> a) -> a -> [b] -> a \)

\( \text{foldl} (\oplus) \) \( e \times x_{y_1}x_jx_2 \) = \((e \oplus x_y) \oplus x_j \) \( \oplus x_2 \)

Examples:

- \( \text{sum} = \text{foldl} (+) 0 \)
- \( \text{product} = \text{foldl} (*) 1 \)

Definition:

\( \text{foldl} f e [] = e \)

\( \text{foldl} f e (x:xs) = \text{foldl} f (f e x) xs \)
**Scan Left:**

A function from the prelude:

\[
\text{scanl} :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow [a]
\]

\[
\text{scanl} \ (\oplus) \ e \ [x_0, x_1, x_2] = [e, e \oplus x_0, (e \oplus x_0) \oplus x_1, ((e \oplus x_0) \oplus x_1) \oplus x_2]
\]

Specification:

\[
\text{scanl} \ f \ e = \text{map} \ (\text{foldl} \ f \ e) \ . \ \text{inits}
\]

\[
\text{inits} \ [\] = [\[]
\]

\[
\text{inits} \ (x:xs) = [\] : \text{map} \ (x:) \ (\text{inits} \ xs)
\]

**Calculating scanl:**

It is easy to derive \(\text{scanl} \ f \ e \ [\] = [e]\)

For non empty lists:

\[
\text{scanl} \ f \ e \ (x:xs) = \text{map} \ (\text{foldl} \ f \ e) \ (\text{inits} \ (x:xs))
\]

\[
\text{scanl} \ f \ e \ (x:xs) = \text{foldl} \ f \ e \ [\] : \text{map} \ (\text{foldl} \ f \ e) \ (\text{inits} \ xs)
\]

\[
\text{scanl} \ f \ e \ (x:xs) = \text{foldl} \ f \ e \ [\] : \text{map} \ (\text{foldl} \ f \ e \ . \ (x:)) \ (\text{inits} \ xs)
\]

\[
\text{scanl} \ f \ e \ (x:xs) = e : \text{map} \ (\text{foldl} \ f \ (f \ e \ x)) \ (\text{inits} \ xs)
\]

\[
\text{scanl} \ f \ e \ (x:xs) = e : \text{scanl} \ f \ (f \ e \ x) \ xs
\]

**Comparison:**

- Specification:

\[
\text{scanl} \ f \ e = \text{map} \ (\text{foldl} \ f \ e) \ . \ \text{inits}
\]

- Definition:

\[
\text{scanl} \ f \ e \ [\] = [e]
\]

\[
\text{scanl} \ f \ e \ (x:xs) = e : \text{scanl} \ f \ (f \ e \ x) \ xs
\]

- 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 \(\text{scanl}\):

\[
\text{scanr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow [b]
\]

\[
\text{scanr} \ (\oplus) \ e \ [x_0, x_1, x_2] = [x_0 \oplus (x_1 \oplus (x_2 \oplus e)), x_1 \oplus (x_2 \oplus e), x_2 \oplus e, e]
\]

More efficient version:

\[
\text{scanr} \ f \ e \ [\] = [e]
\]

\[
\text{scanr} \ f \ e \ (x:xs) = f \ x \ (\text{head} \ ys) : ys
\]

\[
\text{where} \ ys = \text{scanr} \ f \ e \ xs
\]

**Maximum Segment Sum:**

- Given a sequence of numbers, find the subsegment whose sum is largest:

  - Example: maximal subsegment sum for the list \([-1, 2, -3, 5, -2, 1, 3, -2, -3, 6]\) is 7 (for the segment \([5, -2, 1, 3]\))

- Simple solution:

\[
\text{mss} :: [\text{Int}] \rightarrow \text{Int}
\]

\[
\text{mss} = \text{maximum} \ . \ \text{map} \ \text{sum} \ . \ \text{segs}
\]

\[
\text{where} \ \text{segs} = \text{concat} \ . \ \text{map} \ \text{inits} \ . \ \text{tails}
\]

- Not a great performer ... \(O(n^3)\)

**Calculate!**

\[
\text{mss}
\]

\[
= \langle \text{definition of mss} \rangle
\]

\[
= \text{maximum} \ . \ \text{map} \ \text{sum} \ . \ \text{segs}
\]

\[
\text{where} \ \text{segs} = \text{concat} \ . \ \text{map} \ \text{inits} \ . \ \text{tails}
\]
Calculate!

\[ mss = \{ \text{definition of \textit{segs}} \} \]

maximum . map sum . concat . map inits . tails

Calculate!

\[ mss = \{ \text{using } \text{map } f \cdot \text{concat} = \text{concat} \cdot \text{map } (\text{map } f) \} \]

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

\[
(map f \cdot \text{concat}) [x_{s_1}, x_{s_2}, x_{s_3}]
= \text{map } f (x_{s_1} ++ x_{s_2} ++ x_{s_3})
= \text{map } f x_{s_1} ++ \text{map } f x_{s_2} ++ \text{map } f x_{s_3}
\]

Calculate!

\[ mss = \{ \text{the "bookkeeping law"} } \]

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

General form:
\[ \text{foldr } f a \cdot \text{concat} = \text{foldr } f a \cdot \text{map } (\text{foldr } f a) \]
if \( f \) is associative with unit \( a \)

Calculate!

\[ mss = \{ \text{Definition of \textit{scanl}} \} \]

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

Definition:
\[ \text{scanl } f e = \text{map } (\text{foldl } f e) \cdot \text{inits} \]

Calculate!

\[ mss = \{ \text{using } \text{map } f \cdot \text{map } g = \text{map } (f \cdot g) \} \]

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

map \( f \cdot \text{map } g = \text{map } (f \cdot g) \)
(again ...)
Calculate!

\[ mss = \{ \text{fold-scan fusion} \} \]
\[
\text{maximum . map (foldr f 0) . tails}
\]
where \( f \, x \, y = \max 0 \, (x + y) \)

We can prove that:
\[ \text{maximum . scanl (+) 0 = foldr f 0} \]
(A special case of a general property called "Fold-scan fusion")

Calculate!

\[ mss = \{ \text{definition of scanr} \} \]
\[
\text{maximum . scanr f 0}
\]
where \( f \, x \, y = \max 0 \, (x + y) \)

Definition of scanr:
\[ \text{scanr f e = map (foldr f e) . tails} \]
A simple, linear time algorithm, courtesy of equational reasoning!

Calculate!

\[ mss = \{ \text{definition of scanr} \} \]
\[
\text{maximum . scanr f 0}
\]
where \( f \, x \, y = \max 0 \, (x + y) \)

Remember:
\[ \text{scanr (\oplus) e [x_0, x_1, x_2]} = [x_0 \oplus (x_1 \oplus (x_2 \oplus e))], \]
\[ x_1 \oplus (x_2 \oplus e), \]
\[ x_2 \oplus e, \]
\[ e \]

\[ mss \, xs = \text{loop 0 0 (reverse xs)} \]
where
\[
\text{loop m v []} = m
\]
\[
\text{loop m v (x:xs)} = \begin{cases} 
    y = \max 0 \, (x+v) & \text{in loop (max m y) y xs}
\end{cases}
\]

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 into Hugs and quickly check to see if they are equal …

Main> quickCheck (\xs -> mss xs == mss' xs)
OK, passed 100 tests.
Main>

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

To Be Continued …

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!