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 (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 ...

- ... 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:

\[ n = 0; \text{ or} \]
\[ n = 1 + m \text{ for some (smaller) natural } m \]

Functions on natural numbers:

\[ 0 + n = n \]
\[ (1+m) + n = 1 + (m + n) \]

Does this look at all familiar?
+ is associative:

∀n. ∀p. ∀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

\[
(n + p) + q \\
= (((1 + m) + p) + q) \\
= (1 + (m + p)) + q \\
= 1 + ((m + p) + q) \\
= 1 + (m + (p + q)) \\
= (1 + m) + (p + q) \\
= n + (p + q)
\]

(because \( n=1+m \))

(definition of +)

(definition of +)

(induction)

(definition of +)

(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:

$n = \text{Zero};$ or

$n = \text{Succ } m \text{ for some (smaller) natural } m$

data Nat = Zero | Succ Nat

Functions on natural numbers:

$\text{add Zero } n = n$

$\text{add (Succ } m\text{) } n = \text{Succ (add m n)}$
add is associative:

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

If \( n = \text{Zero} \), then
\[
\begin{align*}
\text{add} (\text{add} n p) q \\
= \text{add} (\text{add} \text{Zero} p) q \quad (\text{because } n = \text{Zero}) \\
= \text{add} p q \quad (\text{definition of add}) \\
= \text{add} \text{Zero} (\text{add} p q) \quad (\text{definition of add})
\end{align*}
\]
add is associative:

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

If \( n = \text{Succ} m \), then

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

(because \( n=1+m \))
(definition of +)
(definition of +)
(induction)
(definition of +)
(definition of +)
add is associative:

We’ve shown:
- The property holds for \( n = \text{Zero} \)
- If the property holds for \( n = m \), then it holds for \( n = \text{Succ } m \)
- So it holds for \( n = \text{Succ } \text{Zero} \)
- And for \( n = \text{Succ } (\text{Succ } \text{Zero}) \)
- And for \( n = \text{Succ } (\text{Succ } (\text{Succ } \text{Zero})) \)
- ...

In fact, we’ve shown that it holds for all \( n \):
\[
\forall n. \forall p. \forall q. \text{add } (\text{add } n \ p) \ q = \text{add } n \ (\text{add } p \ q)
\]
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
  \[\text{map } f \cdot \text{map } g = \text{map } (f \cdot g)\]

- From specifications of new functions
  \[\text{sumSquares } n = \text{sum } (\text{map } \text{square } [1..n])\]
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:**
  - $f = g \iff \forall x. f\ x = g\ x$

- **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 $xs = \bot$
- If $b :: \text{Bool}$, then $b=\text{False}$, $b=\text{True}$, or $b=\bot$

**Induction:**

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

- We treat every type in Haskell as having a special element called bottom, written \( \bot \)

- \( \bot \) 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 \( f \downarrow = \downarrow \)

Examples:

- \((1+)\) and \text{not} are both strict
- \((\&\&\) and \((||)\) are strict in their left arguments, but not in their right
- \text{map} is strict in its list argument (but not the function)
Example:

- Suppose we specify:
  \[ f :: [\text{Int}] \to [\text{Int}] \]
  \[ f = \text{map} \ (1+) \]

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

We can also calculate:

\[
\begin{align*}
  f \ (x:xs) \\
  &\quad = \{ \text{by definition of } f \} \\
  &\quad \text{map} \ (1+) \ (x:xs) \\
  &\quad = \{ \text{by definition of } \text{map} \} \\
  &\quad (1+x) : \text{map} \ (1+) \ xs \\
  &\quad = \{ \text{by definition of } f \} \\
  &\quad (1+x) : f \ xs
\end{align*}
\]

Thus we have derived:

\[
\begin{align*}
  f &:: \text{[Int]} \rightarrow \text{[Int]} \\
  f \ [] &\quad = [] \\
  f \ (x:xs) &\quad = (1+x) : f \ xs
\end{align*}
\]
Associativity of (++):

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

Proof by induction on \( xs \):

**Base case:** \( xs = [] \)

\[
\begin{align*}
[] & \quad ++ \quad (ys \quad ++ \quad zs) \\
= & \quad \{ \text{by definition of } ++ \} \\
ys & \quad ++ \quad zs \\
= & \quad \{ \text{by definition of } ++ \} \\
([] \quad ++ \quad ys) & \quad ++ \quad zs
\end{align*}
\]
... continued:

**Base case:** $xs = \bot$

**lhs:** $\bot \mathbin{++} (ys \mathbin{++} zs)$

$$= \{ \mathbin{++} \text{ is strict in its first argument } \} \bot$$

**rhs:** $(\bot \mathbin{++} ys) \mathbin{++} zs$

$$= \{ \mathbin{++} \text{ is strict in its first argument } \} \bot \mathbin{++} zs$$

$$= \{ \mathbin{++} \text{ is strict in its first argument } \} \bot$$
... 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:

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (⊕) e [x₀, x₁, x₂] = x₀ ⊕ (x₁ ⊕ (x₂ ⊕ 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:

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

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

Examples:

\[
\text{sum} \ = \ \text{foldl} \ (+) \ 0
\]

\[
\text{product} \ = \ \text{foldl} \ (\times) \ 1
\]

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:

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

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

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:

\[
\begin{align*}
\text{scanl} \ f \ e \ (x:xs) &= \text{map} \ (\text{foldl} \ f \ e) \ (\text{inits} \ (x:xs)) \\
&= \text{map} \ (\text{foldl} \ f \ e) \ (\square : \text{map} \ (x:) \ (\text{inits} \ xs)) \\
&= \text{foldl} \ f \ e \ [] : \text{map} \ (\text{foldl} \ f \ e) \ (\text{map} \ (x:) \ (\text{inits} \ xs)) \\
&= \text{foldl} \ f \ e \ [] : \text{map} \ (\text{foldl} \ f \ e \ . \ (x:)) \ (\text{inits} \ xs) \\
&= e : \text{map} \ (\text{foldl} \ f \ (f \ e \ x)) \ (\text{inits} \ xs) \\
&= e : \text{scanl} \ f \ (f \ e \ x) \ xs
\end{align*}
\]
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) \ \text{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 \texttt{scanl}:

\[
\begin{align*}
\texttt{scanr} & : (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow [b] \\
\texttt{scanr} \ f \ e & = \text{map} \ (\text{foldr} \ f \ e) \ . \ \text{tails}
\end{align*}
\]

\[
\begin{align*}
\texttt{scanr} \ (\oplus) \ e \ [x_0, x_1, x_2] & = [x_0 \oplus (x_1 \oplus (x_2 \oplus e)), x_1 \oplus (x_2 \oplus e), x_2 \oplus e, e]
\end{align*}
\]

More efficient version:

\[
\begin{align*}
\texttt{scanr} \ f \ e \ [] & = [e] \\
\texttt{scanr} \ f \ e \ (x:xs) & = f \ x \ (\text{head} \ ys) : ys \\
\text{where} \ ys & = \text{scanr} \ f \ e \ xs
\end{align*}
\]
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:
  ```haskell
  mss :: [Int] -> Int
  mss = maximum . map sum . segs
  where segs = concat . map inits . tails
  ```

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

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

\[ \text{maximum . map sum . segs} \]
Calculate!

\[
\text{mss} = \{\text{definition of } \text{segs}\}
\]
\[
\text{maximum} \ . \ \text{map} \ \text{sum} \ . \ \text{concat} \ . \ \text{map} \ \text{inits} \ . \ \text{tails}
\]
Calculate!

\[
mss = \{ \text{using } \text{map } f . \text{concat} = \text{concat} . \text{map (map } f) \} \\
\text{maximum . concat . map (map sum) . map inits . tails}
\]

\[
\text{(map } f . \text{concat) [xs}_1, \text{xs}_2, \text{xs}_3] = \text{map } f (\text{xs}_1 ++ \text{xs}_2 ++ \text{xs}_3) = \text{map } f \text{xs}_1 ++ \text{map } f \text{xs}_2 ++ \text{map } f \text{xs}_3}
\]

\[
\text{(concat . map (map } f)) [\text{xs}_1, \text{xs}_2, \text{xs}_3] = \text{concat [map } f \text{xs}_1, \text{map } f \text{xs}_2, \text{map } f \text{xs}_3] = \text{map } f \text{xs}_1 ++ \text{map } f \text{xs}_2 ++ \text{map } f \text{xs}_3
\]
Calculate!

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

\[
\text{maximum} \cdot \text{concat} \cdot \text{map } (\text{map } \text{sum} \cdot \text{inits}) \cdot \text{tails}
\]

\[
(\text{map } f \cdot \text{map } g) [x_1, x_2, x_3]
\]
\[
= \text{map } f [g x_1, g x_2, g x_3]
\]
\[
= [f (g x_1), f (g x_2), f (g x_3)]
\]

\[
\text{map } (f \cdot g) [x_1, x_2, x_3]
\]
\[
= [ (f \cdot g) x_1, (f \cdot g) x_2, (f \cdot g) x_3]
\]
\[
= [ f (g x_1), f (g x_2), f (g x_3)]
\]
Calculate!

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

\[
\text{maximum . map maximum . map (map sum . inits) . tails}
\]

\[
\text{maximum . concat}
\]

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

**General form:**

\[
\text{foldr f a . concat} = \text{foldr f a . map (foldr f a)}
\]

if \( f \) is associative with unit \( a \)
Calculate!

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

\[
\text{maximum} \cdot \text{map} \text{ maximum} \cdot \text{map} (\text{scanl} (+) 0) \cdot \text{tails}
\]

**Definition:**

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

\[
\text{mss} = \{ \text{using } \text{map } f \cdot \text{map } g = \text{map } (f \cdot g) \} \\
\text{maximum} \cdot \text{map} \ (\text{maximum} \cdot \text{scanl} \ (+) \ 0) \cdot \text{tails}
\]

\[
\text{map } f \cdot \text{map } g = \text{map } (f \cdot g) \\
(\text{again }\ldots)
\]
Calculate!

\[
\text{mss} = \{ \text{fold-scan fusion} \}
\]

\[
\text{maximum} \cdot \text{map} (\text{foldr} \ f \ 0) \cdot \text{tails}
\]

where \( f \ x \ y = \max 0 (x + y) \)

We can prove that:

\[
\text{maximum} \cdot \text{scanl} (+) 0 = \text{foldr} \ f \ 0
\]

(A special case of a general property called “Fold-scan fusion”)
Calculate!

\[ \text{mss} = \{ \text{definition of scanr} \} \]

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

Definition of scanr:

\[ \text{scanr } f \, e = \text{map (foldr } f \, e) \, . \, \text{tails} \]

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

\[
\text{mss} = \{ \text{definition of } \text{scanr} \} \,
\]

\[
\text{maximum} \cdot \text{scanr} \ f \ 0 \quad \text{where } f \ x \ y = \max 0 (x + y)
\]

\[
\text{mss} \ \text{xs} = \text{loop} \ 0 \ 0 \ (\text{reverse} \ \text{xs}) \quad \text{where}
\]

\[
\begin{align*}
\text{loop} \ m \ v \ [] &= m \\
\text{loop} \ m \ v \ (x:xs) &= \text{let } y = \max 0 (x+v) \\
&\quad \text{in } \text{loop} \ (\max m y) \ y \ xs
\end{align*}
\]

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.

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!