Program Calculation

Jeremy Gibbons

IIJ, Tokyo, February 2015

1. Manifesto

- programs are *mathematical objects*
- so we should be able to *calculate* with them:

  prove properties, transform, derive
- this requires *clean semantics* and *concise notation*
- I will argue for functional programming—in particular, *lazy*

2. Insertion sort, imperatively

$\{\ N \geqslant 0\ \}$

$n := 0\ ;$

while $n \neq N$ **do**

  $m := n\ ;\ x := a[m]\ ;$

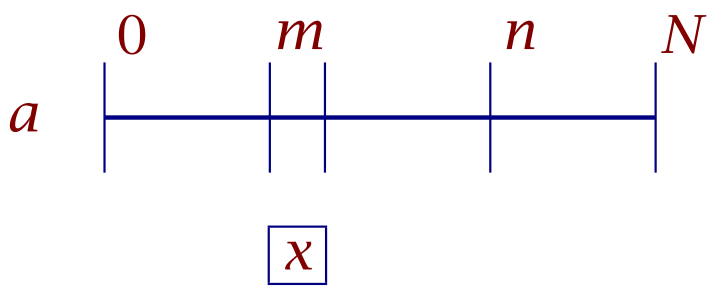
  while $m \neq 0 \wedge x \leqslant a[m-1]$ **do** $a[m] := a[m-1]\ ;\ m := m-1$ **end** ;

  $a[m] := x\ ;$

  $n := n+1$

end

$\{\ a[0..N)$ sorted $\}$



2.1. Outer invariant

$\{\ N \geqslant 0\ \}$

$n := 0\,;$

$\{\ \text{inv: } 0 \leqslant n \leqslant N \wedge a[0..n) \text{ sorted}\ \}$

while $n \neq N$ **do**

  $m := n\,;\ x := a[m]\,;$

  while $m \neq 0 \wedge x \leqslant a[m-1]$ **do** $a[m] := a[m-1]\,;\ m := m-1$ **end** ;

  $a[m] := x\,;$

  $n := n+1$

end

$\{\ a[0..N) \text{ sorted}\ \}$

2.2. Outer loop body

$\{\ N \geqslant 0\ \}$

$n := 0$;

$\{\ \text{inv: } 0 \leqslant n \leqslant N \wedge a[0..n) \text{ sorted}\ \}$

while $n \neq N$ **do**

  $\{\ 0 \leqslant n < N \wedge a[0..n) \text{ sorted}\ \}$

  $m := n$; $x := a[m]$;

  while $m \neq 0 \wedge x \leqslant a[m-1]$ **do** $a[m] := a[m-1]$; $m := m-1$ **end** ;

  $a[m] := x$;

  $n := n+1$

  $\{\ 0 \leqslant n \leqslant N \wedge a[0..n) \text{ sorted}\ \}$

end

$\{\ a[0..N) \text{ sorted}\ \}$

2.3. Restoring outer invariant

{ $N \geqslant 0$ }
$n := 0$;
{ inv: $0 \leqslant n \leqslant N \wedge a[0..n)$ sorted }
while $n \neq N$ **do**
  { $0 \leqslant n < N \wedge a[0..n)$ sorted }
  $m := n$; $x := a[m]$;
  while $m \neq 0 \wedge x \leqslant a[m-1]$ **do** $a[m] := a[m-1]$; $m := m-1$ **end** ;
  $a[m] := x$;
  { $0 \leqslant n+1 \leqslant N \wedge a[0..n+1)$ sorted }
  $n := n+1$
  { $0 \leqslant n \leqslant N \wedge a[0..n)$ sorted }
end
{ $a[0..N)$ sorted }

2.4. Inner loop

$\{\ 0 \leqslant n < N \wedge a[0..n) \text{ sorted } \}$

$m := n \,;\, x := a[m] \,;$

$\textbf{while}\ \ m \neq 0 \wedge x \leqslant a[m-1]\ \ \textbf{do}\ \ a[m] := a[m-1] \,;\, m := m-1\ \ \textbf{end}\,;$

$a[m] := x \,;$

$\{\ 0 \leqslant n+1 \leqslant N \wedge a[0..n+1) \text{ sorted } \}$

2.4. Inner loop

$\{\ 0 \leqslant n < N \land a[0..n)\ \text{sorted}\ \}$

$m := n\,;\ x := a[m]\,;$

while $m \neq 0 \land x \leqslant a[m-1]$ **do**

$\quad a[m] := a[m-1]\,;$

$\quad m := m-1$

end ;

$a[m] := x\,;$

$\{\ 0 \leqslant n+1 \leqslant N \land a[0..n+1)\ \text{sorted}\ \}$

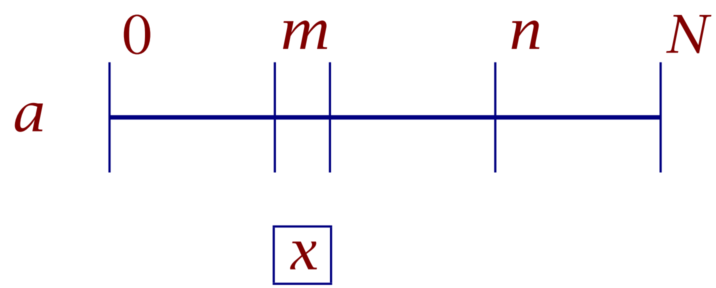
2.5. Inner invariant

$\{\ 0 \leqslant n < N \land a[0..n) \text{ sorted } \}$

$m := n\ ;\ x := a[m]\ ;$

$\{\ \text{inv: } 0 \leqslant m \leqslant n < N \land a[0..m) +\!\!+ a[m+1..n) \text{ sorted} \land x \preccurlyeq a[m+1..n)\ \}$

while $m \neq 0 \land x \leqslant a[m-1]$ **do**



$a[m] := a[m-1]\ ;$

$m := m-1$

end ;

$\{\ 0 \leqslant m \leqslant n < N \land a[0..m) +\!\!+ a[m+1..n) \text{ sorted} \land a[0..m) \preccurlyeq x \preccurlyeq a[m+1..n)\ \}$

$a[m] := x\ ;$

$\{\ 0 \leqslant n+1 \leqslant N \land a[0..n+1) \text{ sorted } \}$

2.6. Inner loop body

$\{\ 0 \leqslant n < N \wedge a[0..n) \text{ sorted } \}$

$m := n\,;\, x := a[m]\,;$

$\{\ \text{inv: } 0 \leqslant m \leqslant n < N \wedge a[0..m) \mathbin{+\!\!+} a[m{+}1..n) \text{ sorted} \wedge x \preccurlyeq a[m{+}1..n) \ \}$

while $m \neq 0 \wedge x \leqslant a[m{-}1]$ **do**

$\quad\{\ 0 < m \leqslant n < N \wedge a[0..m) \mathbin{+\!\!+} a[m{+}1..n) \text{ sorted} \wedge x \preccurlyeq a[m{+}1..n) \wedge x \leqslant a[m{-}1] \ \}$

$\quad a[m] := a[m{-}1]\,;$

$\quad m := m{-}1$

$\quad\{\ 0 \leqslant m \leqslant n < N \wedge a[0..m) \mathbin{+\!\!+} a[m{+}1..n) \text{ sorted} \wedge x \preccurlyeq a[m{+}1..n) \ \}$

end ;

$\{\ 0 \leqslant m \leqslant n < N \wedge a[0..m) \mathbin{+\!\!+} a[m{+}1..n) \text{ sorted} \wedge a[0..m) \preccurlyeq x \preccurlyeq a[m{+}1..n) \ \}$

$a[m] := x\,;$

$\{\ 0 \leqslant n{+}1 \leqslant N \wedge a[0..n{+}1) \text{ sorted } \}$

2.6. Inner loop body

$\{\ 0<m\leqslant n<N \wedge a[0..m){+\!\!+}a[m+1..n) \text{ sorted} \wedge x\preccurlyeq a[m+1..n) \wedge x\leqslant a[m-1]\ \ \}$

$a[m] := a[m-1]$;

$m := m-1$

$\{\ 0\leqslant m\leqslant n<N \wedge a[0..m){+\!\!+}a[m+1..n) \text{ sorted} \wedge x\preccurlyeq a[m+1..n)\ \ \}$

2.6. Inner loop body

$\{\ 0<m\leqslant n<N \wedge a[0..m)+\!\!+a[m+1..n) \text{ sorted} \wedge x\preccurlyeq a[m+1..n) \wedge x\leqslant a[m-1]\ \}$

$a[m] := a[m-1]\ ;$

$m := m-1$

$\{\ 0\leqslant m\leqslant n<N \wedge a[0..m)+\!\!+a[m+1..n) \text{ sorted} \wedge x\preccurlyeq a[m+1..n)\ \}$

2.7. Restoring inner invariant

$\{\ 0<m\leqslant n<N \wedge a[0..m)+\!\!+a[m+1..n)\ \text{sorted} \wedge x\preccurlyeq a[m+1..n) \wedge x\leqslant a[m-1]\ \}$

$a[m] := a[m-1]\ ;$

$\{\ 0\leqslant m-1\leqslant n<N \wedge a[0..m-1)+\!\!+a[m..n)\ \text{sorted} \wedge x\preccurlyeq a[m..n)\ \}$

$m := m-1$

$\{\ 0\leqslant m\leqslant n<N \wedge a[0..m)+\!\!+a[m+1..n)\ \text{sorted} \wedge x\preccurlyeq a[m+1..n)\ \}$

2.7. Restoring inner invariant

$\{\ 0<m\leqslant n<N \wedge a[0..m)+\!\!+a[m+1..n) \text{ sorted} \wedge x\preccurlyeq a[m+1..n) \wedge x\leqslant a[m-1]\ \}$

$a[m] := a[m-1]\,;$

$\{\ 0<m\leqslant n<N \wedge a[0..m-1)+\!\!+a[m+1..n) \text{ sorted} \wedge x\preccurlyeq a[m+1..n)$
$\qquad \wedge a[0..m-1)\preccurlyeq a[m]\preccurlyeq a[m+1..n) \wedge x\leqslant a[m]\ \}$

$\{\ 0\leqslant m-1\leqslant n<N \wedge a[0..m-1)+\!\!+a[m..n) \text{ sorted} \wedge x\preccurlyeq a[m..n)\ \}$

$m := m-1$

$\{\ 0\leqslant m\leqslant n<N \wedge a[0..m)+\!\!+a[m+1..n) \text{ sorted} \wedge x\preccurlyeq a[m+1..n)\ \}$

2.7. Restoring inner invariant

$\{\ 0<m\leqslant n<N \wedge a[0..m) +\!\!+ a[m+1..n) \text{ sorted} \wedge x\preccurlyeq a[m+1..n) \wedge x\leqslant a[m-1]\ \}$

$\{\ 0<m\leqslant n<N \wedge a[0..m-1) +\!\!+ a[m+1..n) \text{ sorted} \wedge x\preccurlyeq a[m+1..n)$
$\quad \wedge a[0..m-1)\preccurlyeq a[m-1]\preccurlyeq a[m+1..n) \wedge x\leqslant a[m-1]\ \}$

$a[m] := a[m-1]$;

$\{\ 0<m\leqslant n<N \wedge a[0..m-1) +\!\!+ a[m+1..n) \text{ sorted} \wedge x\preccurlyeq a[m+1..n)$
$\quad \wedge a[0..m-1)\preccurlyeq a[m]\preccurlyeq a[m+1..n) \wedge x\leqslant a[m]\ \}$

$\{\ 0\leqslant m-1\leqslant n<N \wedge a[0..m-1) +\!\!+ a[m..n) \text{ sorted} \wedge x\preccurlyeq a[m..n)\ \}$

$m := m-1$

$\{\ 0\leqslant m\leqslant n<N \wedge a[0..m) +\!\!+ a[m+1..n) \text{ sorted} \wedge x\preccurlyeq a[m+1..n)\ \}$

Phew!

3. Insertion sort, declaratively

Insert elements one by one into empty list:

$$
\begin{aligned}
&\mathit{isort}\ [\,] &&= [\,] \\
&\mathit{isort}\ (x : xs) &&= \mathit{insert}\ x\ (\mathit{isort}\ xs)
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathit{insert}\ x\ [\,] = [x] \\
&\mathit{insert}\ x\ (y : ys) \\
&\quad \mid x \leqslant y \quad = x : y : ys \\
&\quad \mid x > y \quad = y : \mathit{insert}\ x\ ys
\end{aligned}
$$

3.1. Checking for sortedness

Define

$$\begin{aligned} \mathit{sorted}\ [\,] &= \mathit{True} \\ \mathit{sorted}\ (x : xs) &= (x \preccurlyeq xs) \wedge \mathit{sorted}\ xs \end{aligned}$$

where '$x \preccurlyeq xs$' compares x to every element of xs:

$$\begin{aligned} x \preccurlyeq [\,] &= \mathit{True} \\ x \preccurlyeq (y : ys) &= (x \leqslant y) \wedge (x \preccurlyeq ys) \end{aligned}$$

3.2. Lemma 1: comparison with insertion

We show that

$$(z \preccurlyeq \mathit{insert}\ x\ ys) = (z \leqslant x) \wedge (z \preccurlyeq ys)$$

by induction over ys.

The base case is $[\,]$:

$$\begin{array}{ll} & z \preccurlyeq \mathit{insert}\ x\ [\,] \\ = & \quad [\![\ \mathit{insert}\ ]\!] \\ & z \preccurlyeq [x] \\ = & \quad [\![\ \preccurlyeq\ ]\!] \\ & z \leqslant x \wedge z \preccurlyeq [\,] \end{array}$$

The inductive case…

The inductive case is $y : ys$, assuming true for ys. There are two subcases:

for $x \leqslant y$:

$$
\begin{array}{ll}
& z \preccurlyeq insert\ x\ (y : ys) \\
= & [\![\ insert\ ]\!] \\
& z \preccurlyeq x : y : ys \\
= & [\![\ \preccurlyeq\ ]\!] \\
& (z \leqslant x) \wedge (z \preccurlyeq y : ys)
\end{array}
$$

and for $x > y$:

$$
\begin{array}{ll}
& z \preccurlyeq insert\ x\ (y : ys) \\
= & [\![\ insert\ ]\!] \\
& z \preccurlyeq y : insert\ x\ ys \\
= & [\![\ \preccurlyeq\ ]\!] \\
& (z \leqslant y) \wedge (z \preccurlyeq insert\ x\ ys) \\
= & [\![\ \text{inductive hypothesis}\ ]\!] \\
& (z \leqslant y) \wedge (z \leqslant x) \wedge (z \preccurlyeq ys) \\
= & [\![\ \wedge \text{ associative, commutative}\ ]\!] \\
& (z \leqslant x) \wedge (z \leqslant y) \wedge (z \preccurlyeq ys) \\
= & [\![\ \preccurlyeq\ ]\!] \\
& (z \leqslant x) \wedge (z \preccurlyeq y : ys)
\end{array}
$$

3.3. Lemma 2: insertion preserves sortedness

We show that

$$sorted\ (insert\ x\ ys) = sorted\ ys$$

by induction over ys.

The base case is $[\,]$:

$$\begin{array}{ll} & sorted\ (insert\ x\ [\,]) \\ = & \quad [\![\ insert\ ]\!] \\ & sorted\ [x] \\ = & \quad [\![\ sorted\ ]\!] \\ & (x \preccurlyeq [\,]) \wedge sorted\ [\,] \\ = & \quad [\![\ \preccurlyeq\ ]\!] \\ & sorted\ [\,] \end{array}$$

The inductive case. . .

The inductive case is $y : ys$, assuming true for ys. There are two subcases:

for $x \leqslant y$:

$$
\begin{aligned}
& \mathit{sorted}\ (\mathit{insert}\ x\ (y : ys)) \\
= \quad & [\![\ \mathit{insert}\ ]\!] \\
& \mathit{sorted}\ (x : y : ys) \\
= \quad & [\![\ \mathit{sorted}\ ]\!] \\
& (x \preccurlyeq y : ys) \wedge \mathit{sorted}\ (y : ys) \\
= \quad & [\![\ \preccurlyeq; \mathit{sorted}\ ]\!] \\
& (x \leqslant y) \wedge (x \preccurlyeq ys) \wedge (y \preccurlyeq ys) \\
& \qquad \wedge \mathit{sorted}\ ys \\
= \quad & [\![\ \text{transitivity}\ ]\!] \\
& (x \leqslant y) \wedge (y \preccurlyeq ys) \wedge \mathit{sorted}\ ys \\
= \quad & [\![\ \text{case assumption}\ ]\!] \\
& (y \preccurlyeq ys) \wedge \mathit{sorted}\ ys \\
= \quad & [\![\ \mathit{sorted}\ ]\!] \\
& \mathit{sorted}\ (y : ys)
\end{aligned}
$$

and for $x > y$:

$$
\begin{aligned}
& \mathit{sorted}\ (\mathit{insert}\ x\ (y : ys)) \\
= \quad & [\![\ \mathit{insert}\ ]\!] \\
& \mathit{sorted}\ (y : \mathit{insert}\ x\ ys) \\
= \quad & [\![\ \mathit{sorted}\ ]\!] \\
& (y \preccurlyeq \mathit{insert}\ x\ ys) \wedge \mathit{sorted}\ (\mathit{insert}\ x\ ys) \\
= \quad & [\![\ \text{inductive hypothesis}\ ]\!] \\
& (y \preccurlyeq \mathit{insert}\ x\ ys) \wedge \mathit{sorted}\ ys \\
= \quad & [\![\ \text{Lemma 1}\ ]\!] \\
& (y \leqslant x) \wedge (y \preccurlyeq ys) \wedge \mathit{sorted}\ ys \\
= \quad & [\![\ \text{case assumption}\ ]\!] \\
& (y \preccurlyeq ys) \wedge \mathit{sorted}\ ys \\
= \quad & [\![\ \mathit{sorted}\ ]\!] \\
& \mathit{sorted}\ (y : ys)
\end{aligned}
$$

3.4. Theorem: *isort* yields a sorted list

We show that

$$sorted\ (isort\ xs) = True$$

by induction over *xs*.

The base case is [ ]:

$$\begin{array}{ll} & sorted\ (isort\ [\,]) \\ = & [\![\ isort\ ]\!] \\ & sorted\ [\,] \\ = & [\![\ sorted\ ]\!] \\ & True \end{array}$$

The inductive step is $x : xs$, assuming true for *xs*:

$$\begin{array}{ll} & sorted\ (isort\ (x : xs)) \\ = & [\![\ isort\ ]\!] \\ & sorted\ (insert\ x\ (isort\ xs)) \\ = & [\![\ \text{Lemma 2}\ ]\!] \\ & sorted\ (isort\ xs) \\ = & [\![\ \text{inductive hypothesis}\ ]\!] \\ & True \end{array}$$

4. Reflection

Why was that so much easier?

- functional programs are also *equations*
- evaluation by *substitution of equals for equals*
- reasoning in terms of program text
- no need for a separate 'logical' domain

Plain ordinary *equational reasoning* suffices.

4.1. The possibility of failure

In order really to treat programs as equations, we have to adopt call-by-name (or lazy) semantics.

Consider this program:

$$\mathit{constant}\ x = 3$$

Whatever the argument is, the result is 3. Even when the argument is undefined!

$$\mathit{constant}\ (1\ /\ 0) = 3$$

We cannot afford call-by-value semantics.

5. Program calculation

Here's a naive definition of list reversal:

$$
\begin{aligned}
&\mathit{reverse}\ [\,] &&= [\,] \\
&\mathit{reverse}\ (x : xs) &&= \mathit{reverse}\ xs +\!\!+ [x]
\end{aligned}
$$

where list concatenation is as follows:

$$
\begin{aligned}
&[\,] &&+\!\!+\ ys = ys \\
&(x : xs) &&+\!\!+\ ys = x : (xs +\!\!+ ys)
\end{aligned}
$$

Time for $+\!\!+$ is proportional to length of left-hand argument.
Therefore *reverse* takes quadratic time.

Can we do better?

5.1. Accumulating parameter

Introduce an additional argument:

$$\mathit{revCat}\ \mathit{xs}\ \mathit{ys} = \mathit{reverse}\ \mathit{xs} +\!\!+\ \mathit{ys}$$

Of course, this is a generalization:

$$\mathit{reverse}\ \mathit{xs} = \mathit{reverse}\ \mathit{xs} +\!\!+ [\,] = \mathit{revCat}\ \mathit{xs}\ [\,]$$

This specification of *revCat* is no faster…

...but we can improve it, by calculation.

$$
\begin{array}{ll}
& \mathit{revCat}\ [\,]\ \mathit{ys} \\
= & \{\ \text{specification}\ \} \\
& \mathit{reverse}\ [\,] +\!\!+ \mathit{ys} \\
= & \{\ \mathit{reverse}\ \} \\
& [\,] +\!\!+ \mathit{ys} \\
= & \{\ +\!\!+\ \} \\
& \mathit{ys}
\end{array}
$$

$$
\begin{array}{ll}
& \mathit{revCat}\ (x : \mathit{xs})\ \mathit{ys} \\
= & \{\ \text{specification}\ \} \\
& \mathit{reverse}\ (x : \mathit{xs}) +\!\!+ \mathit{ys} \\
= & \{\ \mathit{reverse}\ \} \\
& (\mathit{reverse}\ \mathit{xs} +\!\!+ [\,x\,]) +\!\!+ \mathit{ys} \\
= & \{\ +\!\!+ \text{ is associative}\ \} \\
& \mathit{reverse}\ \mathit{xs} +\!\!+ ([\,x\,] +\!\!+ \mathit{ys}) \\
= & \{\ +\!\!+\text{; specification}\ \} \\
& \mathit{revCat}\ \mathit{xs}\ (x : \mathit{ys})
\end{array}
$$

We have calculated a different program:

$$
\begin{array}{l}
\mathit{revCat}\ [\,]\ \quad\quad \mathit{ys} = \mathit{ys} \\
\mathit{revCat}\ (x : \mathit{xs})\ \mathit{ys} = \mathit{revCat}\ \mathit{xs}\ (x : \mathit{ys})
\end{array}
$$

This one takes just linear time, not quadratic.

5.2. Maximum segment sum: a parting shot

$$
\begin{array}{ll}
& \mathit{maximum} \circ \mathit{map}\ \mathit{sum} \circ \mathit{segs} \\
= & [\![\ \text{definition of } \mathit{segs}\ ]\!] \\
& \mathit{maximum} \circ \mathit{map}\ \mathit{sum} \circ \mathit{concat} \circ \mathit{map}\ \mathit{inits} \circ \mathit{tails} \\
= & [\![\ \text{polymorphism of } \mathit{concat}\ ]\!] \\
& \mathit{maximum} \circ \mathit{concat} \circ \mathit{map}\ (\mathit{map}\ \mathit{sum}) \circ \mathit{map}\ \mathit{inits} \circ \mathit{tails} \\
= & [\![\ \text{bookkeeping law}\ ]\!] \\
& \mathit{maximum} \circ \mathit{map}\ \mathit{maximum} \circ \mathit{map}\ (\mathit{map}\ \mathit{sum}) \circ \mathit{map}\ \mathit{inits} \circ \mathit{tails} \\
= & [\![\ \mathit{map} \text{ distributes over composition}\ ]\!] \\
& \mathit{maximum} \circ \mathit{map}\ (\mathit{maximum} \circ \mathit{map}\ \mathit{sum} \circ \mathit{inits}) \circ \mathit{tails} \\
= & [\![\ \text{definition of } \mathit{scanl}\ ]\!] \\
& \mathit{maximum} \circ \mathit{map}\ (\mathit{maximum} \circ \mathit{scanl}\ (+)\ 0) \circ \mathit{tails} \\
= & [\![\ \text{Horner's rule: let } h\ x\ y = 0\ \text{`}\mathit{max}\text{`}\ (x+y)\ ]\!] \\
& \mathit{maximum} \circ \mathit{map}\ (\mathit{foldr}\ h\ 0) \circ \mathit{tails} \\
= & [\![\ \text{definition of } \mathit{scanr}\ ]\!] \\
& \mathit{maximum} \circ \mathit{scanr}\ h\ 0
\end{array}
$$

6. Conclusion

- programs are *mathematical objects*
- so we should be able to *calculate* with them:

  prove properties, transform, derive
- this requires *clean semantics* and *concise notation*
- lazy functional programming provides both

The 'maximum segment sum' example is from Richard Bird's *Algebraic Identities for Program Calculation*; see also my blog post `https://patternsinfp.wordpress.com/2011/05/05/horners-rule/`.

7. Appendix: definitions for MSS

7.1. Folds on lists

Fold right on lists, eg for *concat* and *map*:

$$foldr\ f\ e\ [\,] \quad = e$$
$$foldr\ f\ e\ (x:xs) = f\ x\ (foldr\ f\ e\ xs)$$

$$concat = foldr\ (+\!\!+)\ [\,]$$
$$map\ f \ = foldr\ (\lambda x\ ys \rightarrow f\ x : ys)\ [\,]$$

Fold left (accumulating fold) on lists, eg for *sum*:

$$foldl\ f\ e\ [\,] \quad = e$$
$$foldl\ f\ e\ (x:xs) = foldl\ f\ (f\ e\ x)\ xs$$

$$sum = foldl\ (+)\ 0$$

Fold on non-empty lists, eg for *maximum*:

$$foldr_1\ f\ [x] \quad = x$$
$$foldr_1\ f\ (x:xs) = f\ x\ (foldr_1\ f\ xs)$$

$$maximum = foldr_1\ max$$

7.2. Partitioning lists

Tail segments:

$$\begin{aligned} \mathit{tails}\ [\,] &= [[\,]] \\ \mathit{tails}\ (x : xs) &= (x : xs) : \mathit{tails}\ xs \end{aligned}$$

Initial segments:

$$\begin{aligned} \mathit{inits}\ [\,] &= [[\,]] \\ \mathit{inits}\ (x : xs) &= [\,] : \mathit{map}\ (x{:})\ (\mathit{inits}\ xs) \end{aligned}$$

All segments:

$$\mathit{segs} = \mathit{concat} \circ \mathit{map}\ \mathit{inits} \circ \mathit{tails}$$

7.3. Scans on lists

Scan right:

$$scanr\ f\ e = map\ (foldr\ f\ e) \circ tails$$

from which we can calculate:

$$\begin{aligned} scanr\ f\ e\ [\,] &= [\,e\,] \\ scanr\ f\ e\ (x : xs) &= \mathbf{let}\ ys = scanr\ f\ e\ xs \\ &\ \mathbf{in}\ \ f\ x\ (head\ ys) : ys \end{aligned}$$

Scan left:

$$scanl\ f\ e = map\ (foldl\ f\ e) \circ inits$$

from which we can calculate:

$$\begin{aligned} scanl\ f\ e\ [\,] &= [\,e\,] \\ scanl\ f\ e\ (x : xs) &= e : scanl\ f\ (f\ e\ x)\ xs \end{aligned}$$

7.4. Lemmas

Polymorphism of *concat*:

$$map\ f \circ concat = concat \circ map\ (map\ f)$$

'Bookkeeping law': for associative *op*,

$$foldr_1\ op \circ concat = foldr_1\ op \circ map\ (foldr_1\ op)$$

Distributiong of *map* over composition:

$$map\ (f \circ g) = map\ f \circ map\ g$$

Horner's rule: if g is associative with unit e, and g distributes over f, then

$$foldr_1\ f \circ scanl\ g\ e = foldr\ h\ e$$

where $h\ x\ y = f\ e\ (g\ x\ y)$.