Functional Programming: Folds, and Fold-Fusion

Shin-Cheng Mu

FLOLAC 2020

1 Folds On Lists

A Common Pattern We've Seen Many Times...

$$\begin{array}{lcl} sum\ [\,] & = & 0 \\ sum\ (x : xs) & = & x + sum\ xs \end{array}$$

$$\begin{array}{lcl} length\ [\,] & = & 0 \\ length\ (x : xs) & = & 1 + length\ xs \end{array}$$

$$\begin{array}{lcl} map\ f\ [\,] & = & [\,] \\ map\ f\ (x : xs) & = & f\ x : map\ f\ xs \end{array}$$

This pattern is extracted and called *foldr*:

$$\begin{array}{lcl} foldr\ f\ e\ [\,] & = & e, \\ foldr\ f\ e\ (x : xs) & = & f\ x\ (foldr\ f\ e\ xs). \end{array}$$

1.1 The Ubiquitous *foldr*

Replacing Constructors

$$\begin{array}{lcl} foldr\ f\ e\ [\,] & = & e \\ foldr\ f\ e\ (x : xs) & = & f\ x\ (foldr\ f\ e\ xs) \end{array}$$

- One way to look at $foldr\ (\oplus)\ e$ is that it replaces $[\,]$ with e and $(:)$ with $(\oplus)$:

$$\begin{array}{ll} & foldr\ (\oplus)\ e\ [1,2,3,4] \\ = & foldr\ (\oplus)\ e\ (1 : (2 : (3 : (4 : [\,])))) \\ = & 1 \oplus (2 \oplus (3 \oplus (4 \oplus e))). \end{array}$$

- $sum = foldr\ (+)\ 0$.
- $length = foldr\ (\lambda x\ n.1 + n)\ 0$.
- $map\ f = foldr\ (\lambda x\ xs.f\ x : xs)\ [\,]$.
- One can see that $id = foldr\ (:)\ [\,]$.

Some Trivial Folds on Lists

- Function *max* returns the maximum element in a list:

$$\begin{array}{lcl} max\ [\,] & = & \text{-}\infty, \\ max\ (x : xs) & = & x \uparrow max\ xs. \end{array}$$

  $max = foldr\ (\uparrow)\ \text{-}\infty$.

- This function is actually called *maximum* in the standard Haskell Prelude, while *max* returns the maximum between its two arguments. For brevity, we denote the former by *max* and the latter by $(\uparrow)$.
- Function *prod* returns the product of a list:

$$\begin{array}{lcl} prod\ [\,] & = & 1, \\ prod\ (x : xs) & = & x \times prod\ xs. \end{array}$$

  $prod = foldr\ (\times)\ 1$.

- Function *and* returns the conjunction of a list:

$$\begin{array}{lcl} and\ [\,] & = & true, \\ and\ (x : xs) & = & x \wedge and\ xs. \end{array}$$

  $and = foldr\ (\wedge)\ true$.

- Lets emphasise again that *id* on lists is a fold:

$$\begin{array}{lcl} id\ [\,] & = & [\,], \\ id\ (x : xs) & = & x : id\ xs. \end{array}$$

  $id = foldr\ (:)\ [\,]$.

Some Functions We Have Seen...

- $(+\!\!+\ ys) = foldr\ (:)\ ys$.

$$\begin{array}{lcl} (+\!\!+) & :: & [a] \to [a] \to [a] \\ [\,] +\!\!+ ys & = & ys \\ (x : xs) +\!\!+ ys & = & x : (xs +\!\!+ ys)\ . \end{array}$$

- $concat = foldr\ (+\!\!+)\ [\,]$.

$$\begin{array}{lcl} concat & :: & [[a]] \to [a] \\ concat\ [\,] & = & [\,] \\ concat\ (xs : xss) & = & xs +\!\!+ concat\ xss\ . \end{array}$$

Replacing Constructors

- Understanding *foldr* from its type. Recall

$$\mathbf{data}\ [a] \ = \ [\,] \mid a : [a] \ .$$

- Types of the two constructors: $[\,] :: [a]$, and $(:) :: a \to [a] \to [a]$.
- *foldr* replaces the constructors:

$$\begin{array}{lcl} foldr & :: & (a \to b \to b) \to b \to [a] \to b \\ foldr\ f\ e\ [\,] & = & e \\ foldr\ f\ e\ (x : xs) & = & f\ x\ (foldr\ f\ e\ xs) \ . \end{array}$$

1.2 The Fold-Fusion Theorem

Why Folds?

- "What are the three most important factors in a programming language?" Abstraction, abstraction, and abstraction!
- Control abstraction, procedure abstraction, data abstraction,... can programming patterns be abstracted too?
- Program structure becomes an entity we can talk about, reason about, and reuse.
  - We can describe algorithms in terms of fold, unfold, and other recognised patterns.
  - We can prove properties about folds,
  - and apply the proved theorems to all programs that are folds, either for compiler optimisation, or for mathematical reasoning.
- Among the theorems about folds, the most important is probably the *fold-fusion* theorem.

The Fold-Fusion Theorem

The theorem is about when the composition of a function and a fold can be expressed as a fold.

Theorem 1 (*foldr*-Fusion)**.** Given $f :: a \to b \to b$, $e :: b$, $h :: b \to c$, and $g :: a \to c \to c$, we have:

$$h \cdot foldr\ f\ e \ = \ foldr\ g\ (h\ e) \ ,$$

if $h\ (f\ x\ y) = g\ x\ (h\ y)$ for all x and y.

For program derivation, we are usually given h, f, and e, from which we have to construct g.

Tracing an Example

Let us try to get an intuitive understand of the theorem:

$$\begin{array}{ll} & h\ (foldr\ f\ e\ [a, b, c]) \\ = & \{ \text{ definition of } foldr \ \} \\ & h\ (f\ a\ (f\ b\ (f\ c\ e))) \\ = & \{ \text{ since } h\ (f\ x\ y) = g\ x\ (h\ y) \ \} \\ & g\ a\ (h\ (f\ b\ (f\ c\ e))) \\ = & \{ \text{ since } h\ (f\ x\ y) = g\ x\ (h\ y) \ \} \\ & g\ a\ (g\ b\ (h\ (f\ c\ e))) \\ = & \{ \text{ since } h\ (f\ x\ y) = g\ x\ (h\ y) \ \} \\ & g\ a\ (g\ b\ (g\ c\ (h\ e))) \\ = & \{ \text{ definition of } foldr \ \} \\ & foldr\ g\ (h\ e)\ [a, b, c] \ . \end{array}$$

Sum of Squares, Again

- Consider $sum \cdot map\ square$ again. This time we use the fact that $map\ f = foldr\ (mf\ f)\ [\,]$, where $mf\ f\ x\ xs = f\ x : xs$.
- $sum \cdot map\ square$ is a fold, if we can find a ssq such that $sum\ (mf\ square\ x\ xs) = ssq\ x\ (sum\ xs)$. Let us try:

$$\begin{array}{ll} & sum\ (mf\ square\ x\ xs) \\ = & \{ \text{ definition of } mf \ \} \\ & sum\ (square\ x : xs) \\ = & \{ \text{ definition of } sum \ \} \\ & square\ x + sum\ xs \\ = & \{ \text{ let } ssq\ x\ y = square\ x + y \ \} \\ & ssq\ x\ (sum\ xs) \ . \end{array}$$

Therefore, $sum \cdot map\ square = foldr\ ssq\ 0$.

Sum of Squares, without Folds

Recall that this is how we derived the inductive case of *sumsq* yesterday:

$$\begin{aligned}
& \mathit{sumsq}\ (x : xs) \\
= & \quad \{\ \text{definition of } \mathit{sumsq}\ \} \\
& \mathit{sum}\ (\mathit{map}\ \mathit{square}\ (x : xs)) \\
= & \quad \{\ \text{definition of } \mathit{map}\ \} \\
& \mathit{sum}\ (\mathit{square}\ x : \mathit{map}\ \mathit{square}\ xs) \\
= & \quad \{\ \text{definition of } \mathit{sum}\ \} \\
& \mathit{square}\ x + \mathit{sum}\ (\mathit{map}\ \mathit{square}\ xs) \\
= & \quad \{\ \text{definition of } \mathit{sumsq}\ \} \\
& \mathit{square}\ x + \mathit{sumsq}\ xs\ .
\end{aligned}$$

Comparing the two derivations, by using fold-fusion we supply only the "important" part.

More on Folds and Fold-fusion

- Compare the proof with the one yesterday. They are essentially the same proof.
- Fold-fusion theorem abstracts away the common parts in this kind of inductive proofs, so that we need to supply only the "important" parts.
- Tupling can be seen as a kind of fold-fusion. The derivation of *steepsum*, for example, can be seen as fusing:

  $$\mathit{steepsum} \cdot \mathit{id} = \mathit{steepsum} \cdot \mathit{foldr}\ (:)\ [\,].$$

  - Recall that $\mathit{steepsum}\ xs = (\mathit{steep}\ xs, \mathit{sum}\ xs)$. Reformulating *steepsum* into a fold allows us to compute it in one traversal.
- Not every function can be expressed as a fold. For example, $\mathit{tail} :: [a] \to [a]$ is not a fold!

1.3 More Useful Functions Defined as Folds

Longest Prefix

- The function call *takeWhile p xs* returns the longest prefix of *xs* that satisfies *p*:

  $$\begin{aligned}
  & \mathit{takeWhile}\ p\ [\,] = [\,] \\
  & \mathit{takeWhile}\ p\ (x : xs) = \\
  & \quad \textbf{if}\ p\ x\ \textbf{then}\ x : \mathit{takeWhile}\ p\ xs \\
  & \quad \textbf{else}\ [\,]\ .
  \end{aligned}$$

- E.g. $\mathit{takeWhile}\ (\leq 3)\ [1,2,3,4,5] = [1,2,3]$.
- It can be defined by a fold:

  $$\begin{aligned}
  \mathit{takeWhile}\ p &= \mathit{foldr}\ (\mathit{tke}\ p)\ [\,], \\
  \mathit{tke}\ p\ x\ xs &= \textbf{if}\ p\ x\ \textbf{then}\ x : xs\ \textbf{else}\ [\,].
  \end{aligned}$$

- Its dual, $\mathit{dropWhile}\ (\leq 3)\ [1,2,3,4,5] = [4,5]$, is not a fold.

All Prefixes

- The function *inits* returns the list of all prefixes of the input list:

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

- E.g. $\mathit{inits}\ [1,2,3] = [[\,],[1],[1,2],[1,2,3]]$.
- It can be defined by a fold:

  $$\begin{aligned}
  \mathit{inits} &= \mathit{foldr}\ \mathit{ini}\ [[\,]], \\
  \mathit{ini}\ x\ xss &= [\,] : \mathit{map}\ (x\ :)\ xss.
  \end{aligned}$$

All Suffixes

- The function *tails* returns the list of all suffixes of the input list:

  $$\begin{aligned}
  \mathit{tails}\ [\,] &= [[\,]], \\
  \mathit{tails}\ (x : xs) &= \textbf{let}\ (ys : yss) = \mathit{tails}\ xs \\
  & \quad\ \textbf{in}\ (x : ys) : ys : yss.
  \end{aligned}$$

- E.g. $\mathit{tails}\ [1,2,3] = [[1,2,3],[2,3],[3],[\,]]$.
- It can be defined by a fold:

  $$\begin{aligned}
  \mathit{tails} &= \mathit{foldr}\ \mathit{til}\ [[\,]], \\
  \mathit{til}\ x\ (ys : yss) &= (x : ys) : ys : yss.
  \end{aligned}$$

Scan

- $\mathit{scanr}\ f\ e = \mathit{map}\ (\mathit{foldr}\ f\ e) \cdot \mathit{tails}$.
- E.g.

  $$\begin{aligned}
  & \mathit{scanr}\ (+)\ 0\ [1,2,3] \\
  = & \ \mathit{map}\ \mathit{sum}\ (\mathit{tails}\ [1,2,3]) \\
  = & \ \mathit{map}\ \mathit{sum}\ [[1,2,3],[2,3],[3],[\,]] \\
  = & \ [6,5,3,0].
  \end{aligned}$$

- Of course, it is slow to actually perform *map* (*foldr f e*) separately. By fold-fusion, we get a faster implementation:

  $$\begin{aligned}
  \mathit{scanr}\ f\ e &= \mathit{foldr}\ (\mathit{sc}\ f)\ [e], \\
  \mathit{sc}\ f\ x\ (y : ys) &= f\ x\ y : y : ys.
  \end{aligned}$$

2 Folds on Other Algebraic Datatypes

- Folds are a specialised form of induction.
- Inductive datatypes: types on which you can perform induction.
- Every inductive datatype give rise to its fold.
- In fact, an inductive type can be defined by its fold.

Fold on Natural Numbers

- Recall the definition:

$$\mathbf{data}\ Nat\ =\ 0\ \mid\ \mathbf{1}_+\ Nat\ .$$

- Constructors: $0 :: Nat$, $(\mathbf{1}_+) :: Nat \to Nat$.
- What is the fold on Nat?

$$\begin{aligned}
&foldN &&:: (a \to a) \to a \to Nat \to a\\
&foldN\ f\ e\ 0 &&= e\\
&foldN\ f\ e\ (\mathbf{1}_+\ n) &&= f\ (foldN\ f\ e\ n)\ .
\end{aligned}$$

Examples of *foldN*

- $(+n) = foldN\ (\mathbf{1}_+)\ n$.

$$\begin{aligned}
&0 + n &&= n\\
&(\mathbf{1}_+\ m) + n &&= \mathbf{1}_+\ (m + n)\ .
\end{aligned}$$

- $(\times n) = foldN\ (n+)\ 0$.

$$\begin{aligned}
&0 \times n &&= 0\\
&(\mathbf{1}_+\ m) \times n &&= n + (m \times n)\ .
\end{aligned}$$

- $even = foldN\ not\ True$.

$$\begin{aligned}
&even\ 0 &&= True\\
&even\ (\mathbf{1}_+\ n) &&= not\ (even\ n)\ .
\end{aligned}$$

Fold-Fusion for Natural Numbers

Theorem 2 (*foldN*-Fusion)**.** Given $f :: a \to a$, $e :: a$, $h :: a \to b$, and $g :: b \to b$, we have:

$$h \cdot foldN\ f\ e\ =\ foldN\ g\ (h\ e)\ ,$$

if $h\ (f\ x) = g\ (h\ x)$ for all x.

Exercise: fuse *even* into (+)?

Folds on Trees

- Recall some datatypes for trees:

$$\begin{aligned}
&\mathbf{data}\ ITree\ a &&= \mathrm{Null}\ \mid\ \mathrm{Node}\ \alpha\ (ITree\ a)\ (ITree\ a)\ ,\\
&\mathbf{data}\ ETree\ a &&= \mathrm{Tip}\ a\ \mid\ \mathrm{Bin}\ (ETree\ a)\ (ETree\ a)\ .
\end{aligned}$$

- The fold for *ITree*, for example, is defined by:

$$\begin{aligned}
&foldIT &&:: (a \to b \to b \to b) \to b \to ITree\ a \to b\\
&foldIT\ f\ e\ \mathrm{Null} &&= e\\
&foldIT\ f\ e\ (\mathrm{Node}\ a\ t\ u) &&= f\ a\ (foldIT\ f\ e\ t)\ (foldIT\ f\ e\ u)\ .
\end{aligned}$$

- The fold for *ETree*, is given by:

$$\begin{aligned}
&foldET &&:: (b \to b \to b) \to (a \to b) \to ETree\ a \to b\\
&foldET\ f\ g\ (\mathrm{Tip}\ x) &&= g\ x\\
&foldET\ f\ g\ (\mathrm{Bin}\ t\ u) &&= f\ (foldET\ f\ g\ t)\ (foldET\ f\ g\ u)\ .
\end{aligned}$$

Some Simple Functions on Trees

- To compute the size of an *ITree*:

$$sizeITree\ =\ foldIT\ (\lambda x\ m\ n \to \mathbf{1}_+\ (m + n))\ 0\ .$$

- To sum up labels in an *ETree*:

$$sumETree\ =\ foldET\ (+)\ id.$$

- To compute a list of all labels in an *ITree* and an *ETree*:

$$\begin{aligned}
&flattenIT &&= foldIT\ (\lambda x\ xs\ ys \to xs \mathbin{+\!\!+} [x] \mathbin{+\!\!+} ys)\ [\,],\\
&flattenET &&= foldET\ (\mathbin{+\!\!+})\ (\lambda x \to [x]).
\end{aligned}$$

- **Exercise**: what are the fusion theorems for *foldIT* and *foldET*?

3 Finally, Solving Maximum Segment Sum

Specifying Maximum Segment Sum

- Finally we have introduced enough concepts to tackle the maximum segment sum problem!
- A segment can be seen as a prefix of a suffix.
- The function *segs* computes the list of all the segments.

$$segs\ =\ concat \cdot map\ inits \cdot tails.$$

- Therefore, *mss* is specified by:

$$mss\ =\ max \cdot map\ sum \cdot segs.$$

The Derivation!

We reason:

$$
\begin{aligned}
& max \cdot map\ sum \cdot concat \cdot map\ inits \cdot tails \\
= \ & \{ \text{ since } map\ f \cdot concat = concat \cdot map\ (map\ f) \ \} \\
& max \cdot concat \cdot map\ (map\ sum) \cdot map\ inits \cdot tails \\
= \ & \{ \text{ since } max \cdot concat = max \cdot map\ max \ \} \\
& max \cdot map\ max \cdot map\ (map\ sum) \cdot map\ inits \cdot tails \\
= \ & \{ \text{ since } map\ f \cdot map\ g = map\ (f.g) \ \} \\
& max \cdot map\ (max \cdot map\ sum \cdot inits) \cdot tails \ .
\end{aligned}
$$

Recall the definition $scanr\ f\ e = map\ (foldr\ f\ e) \cdot tails$. If we can transform $max \cdot map\ sum \cdot inits$ into a fold, we can turn the algorithm into a *scanr*, which has a faster implementation.

Maximum Prefix Sum

Concentrate on $max \cdot map\ sum \cdot inits$:

$$
\begin{aligned}
& max \cdot map\ sum \cdot inits \\
= \ & \{ \text{ definition of } init,\ ini\ x\ xss = [\,] : map\ (x\ :)\ xss \ \} \\
& max \cdot map\ sum \cdot foldr\ ini\ [[\,]] \\
= \ & \{ \text{ fold fusion, see below } \} \\
& max \cdot foldr\ zplus\ [0] \ .
\end{aligned}
$$

The fold fusion works because:

$$
\begin{aligned}
& map\ sum\ (ini\ x\ xss) \\
= \ & map\ sum\ ([\,] : map\ (x\ :)\ xss) \\
= \ & 0 : map\ (sum \cdot (x\ :))\ xss \\
= \ & 0 : map\ (x+)\ (map\ sum\ xss) \ .
\end{aligned}
$$

Define $zplus\ x\ yss = 0 : map\ (x+)\ yss$.

Maximum Prefix Sum, 2nd Fold Fusion

Concentrate on $max \cdot map\ sum \cdot inits$:

$$
\begin{aligned}
& max \cdot map\ sum \cdot inits \\
= \ & \{ \text{ definition of } init,\ ini\ x\ xss = [\,] : map\ (x\ :)\ xss \ \} \\
& max \cdot map\ sum \cdot foldr\ ini\ [[\,]] \\
= \ & \{ \text{ fold fusion, } zplus\ x\ xss = 0 : map\ (x+)\ xss \ \} \\
& max \cdot foldr\ zplus\ [0] \\
= \ & \{ \text{ fold fusion, let } zmax\ x\ y = 0 \uparrow (x + y) \ \} \\
& foldr\ zmax\ 0 \ .
\end{aligned}
$$

The fold fusion works because $\uparrow$ distributes into $(+)$:

$$
\begin{aligned}
& max\ (0 : map\ (x+)\ xs) \\
= & 0 \uparrow max\ (map\ (x+)\ xs) \\
= & 0 \uparrow (x + max\ xs) \ .
\end{aligned}
$$

Back to Maximum Segment Sum

We reason:

$$
\begin{aligned}
& max \cdot map\ sum \cdot concat \cdot map\ inits \cdot tails \\
= \ & \{ \text{ since } map\ f \cdot concat = concat \cdot map\ (map\ f) \ \} \\
& max \cdot concat \cdot map\ (map\ sum) \cdot map\ inits \cdot tails \\
= \ & \{ \text{ since } max \cdot concat = max \cdot map\ max \ \} \\
& max \cdot map\ max \cdot map\ (map\ sum) \cdot map\ inits \cdot tails \\
= \ & \{ \text{ since } map\ f \cdot map\ g = map\ (f.g) \ \} \\
& max \cdot map\ (max \cdot map\ sum \cdot inits) \cdot tails \\
= \ & \{ \text{ reasoning in the previous slides } \} \\
& max \cdot map\ (foldr\ zmax\ 0) \cdot tails \\
= \ & \{ \text{ introducing } scanr \ \} \\
& max \cdot scanr\ zmax\ 0 \ .
\end{aligned}
$$

Maximum Segment Sum in Linear Time!

- We have derived $mss = max \cdot scanr\ zmax\ 0$, where $zmax\ x\ y = 0 \uparrow (x + y)$.
- The algorithm runs in linear time, but takes linear space.
- A tupling transformation eliminates the need for linear space.

  $$mss \ = \ fst \cdot maxhd \cdot scanr\ zmax\ 0$$

  where $maxhd\ xs = (max\ xs, head\ xs)$. We omit this last step in the lecture.
- The final program is $mss = fst \cdot foldr\ step\ (0, 0)$, where $step\ x\ (m, y) = ((0 \uparrow (x + y)) \uparrow m, 0 \uparrow (x + y))$.