Folds and unfolds all around us

Conal Elliott

Tabula

Spring, 2013
This talk is a literate Haskell program.

```haskell
module FoldsAndUnfolds where
```

I’ll use some non-standard (for Haskell) type notation:

- `type 1 = ()`
- `type (+) = Either`
- `type (∗) = (,)`

- `infixl 7 ×`
- `infixl 6 +`
Recursive functional programming

On numbers:

\[
\text{fact}_0 0 = 1 \\
\text{fact}_0 n = n \times \text{fact}_0 (n - 1)
\]

On lists:

\[
\text{data } [a] = [] | a : [a]
\]

\[
\text{product}_L :: [\text{Integer}] \rightarrow \text{Integer} \\
\text{product}_L [] = 1 \\
\text{product}_L (a : as) = a \times \text{product}_L as
\]

\[
\text{range}_L :: \text{Integer} \rightarrow \text{Integer} \rightarrow [\text{Integer}] \\
\text{range}_L l h \mid l > h = [] \\
| \text{otherwise} = l : \text{range}_L (\text{succ } l) h
\]
Recursive functional programming

On (binary leaf) trees:

\[
\text{data } T \ a = L \ a \mid B \ (T \ a) \ (T \ a) \ \text{deriving Show}
\]

\[
\text{product}_T :: T \ \text{Integer} \to \text{Integer}
\]
\[
\text{product}_T \ (L \ a) = a
\]
\[
\text{product}_T \ (B \ s \ t) = \text{product}_T \ s \times \text{product}_T \ t
\]

\[
\text{range}_T :: \text{Integer} \to \text{Integer} \to T \ \text{Integer}
\]
\[
\text{range}_T \ l \ h \mid l \equiv h = L \ l
\]
\[
\mid \text{otherwise} = B \ (\text{range}_T \ l \ m) \ (\text{range}_T \ (m + 1) \ h)
\]
\[
\text{where } m = (l + h) \text{‘}\div\text{‘} 2
\]
Recursive functional programming?
Structured functional programming

... recursive equations are the “assembly language” of functional programming, and direct recursion the goto.

Jeremy Gibbons, Origami programming

A structured alternative:

- identify commonly useful patterns,
- determine their properties, and
- apply the patterns and properties.
Folds ("catamorphisms")

Contract a structure \textit{down to} a single value.

For lists:

\[
\begin{align*}
\text{fold}_L &:: (a \to b \to b) \to b \to ([a] \to b) \\
\text{fold}_L &- b \; [] = b \\
\text{fold}_L & f \; b \; (a : as) = f \; a \; (\text{fold}_L \; f \; b \; as)
\end{align*}
\]

\[
\begin{align*}
\text{sum}_L & = \text{fold}_L \; (+) \; 0 \\
\text{product}_L & = \text{fold}_L \; (\times) \; 1 \\
\text{reverse}_L & = \text{fold}_L \; (\lambda a \; r \to r + [a]) \; []
\end{align*}
\]

For trees:

\[
\begin{align*}
\text{fold}_T &:: (b \to b \to b) \to (a \to b) \to (T \; a \to b) \\
\text{fold}_T &- l \; (L \; a) = l \; a \\
\text{fold}_T & b \; l \; (B \; s \; t) = b \; (\text{fold}_T \; b \; l \; s) \; (\text{fold}_T \; b \; l \; t)
\end{align*}
\]

\[
\begin{align*}
\text{product}_T & = \text{fold}_T \; (\times) \; \text{id}
\end{align*}
\]
Unfolds ("anamorphisms")

Expand a structure \textit{up from} a single value.

Lists:

\[
\begin{align*}
\text{unfold}_L :: (b \to \text{Maybe} (a \times b)) & \to (b \to [a]) \\
\text{unfold}_L f \ b &= \text{case } f \ b \text{ of} \\
& \quad \text{Just} \ (a, b') \to a : \text{unfold}_L f \ b' \\
& \quad \text{Nothing} \quad \to []
\end{align*}
\]

\[
\begin{align*}
\text{range}_L' :: \text{Integer} \times \text{Integer} & \to [\text{Integer}] \\
\text{range}_L' &= \text{unfold}_L g \\
\text{where} \\
& \quad g \ (l, h) \mid l > h \quad = \text{Nothing} \\
& \quad \mid \text{otherwise} \quad = \text{Just} \ (l, (\text{succ} \ l, h))
\end{align*}
\]
Unfolds ("anamorphisms")

Trees:

\[
\begin{align*}
\text{unfold}_T &:: (b \to a + b \times b) \to (b \to T\ a) \\
\text{unfold}_T\ g\ x & = \text{case } g\ x\ \text{of} \\
& \quad \text{Left } a \quad \to L\ a \\
& \quad \text{Right } (c, d) \to B\ (\text{unfold}_T\ g\ c)\ (\text{unfold}_T\ g\ d)
\end{align*}
\]

\[
\begin{align*}
\text{range}_{TP} &:: \text{Integer} \times \text{Integer} \to T\ \text{Integer} \\
\text{range}_{TP} & = \text{unfold}_T\ g \\
\text{where} & \\
& g\ (l, h) \mid l \equiv h = \text{Left } l \\
& \quad \mid \text{otherwise} = \text{Right } ((l, m), (m + 1, h)) \\
& \text{where } m = (l + h) \text{‘}\text{div}‘\ 2
\end{align*}
\]
Factorial again

Assembly language:

\[
\text{fact}_0 \ 0 = 1 \\
\text{fact}_0 \ n = n \times \text{fact}_0 \ (n - 1)
\]

You may have seen this Haskelly definition:

\[
\text{fact}_1 \ n = \text{product} \ [1..n]
\]

*Theme*: replace control structures by data structures and standard combining forms.

Carry this theme further.
Combining *unfold* and *fold*

Equivalently,

\[ \text{fact}_1 = \text{product}_L \circ \text{range}_L 1 \]

*Note*: composition of unfold \((\text{range}_L)\) and fold \(\text{product}_L\).

More explicit:

\[ \text{fact}_2 = \text{fold}_L (\times) 1 \circ \text{unfold}_L g \]

*where*

\[ g \ 0 = \text{Nothing} \]
\[ g \ n = \text{Just} (n, n - 1) \]

This combination of *unfold* and *fold* is called a “hylomorphism”. 
Fibonacci

Assembly language:

\[
\begin{align*}
\text{fib}_0 \ 0 &= 0 \\
\text{fib}_0 \ 1 &= 1 \\
\text{fib}_0 \ n &= \text{fib}_0 \ (n - 1) + \text{fib}_0 \ (n - 2)
\end{align*}
\]

Via trees:

\[
\begin{align*}
\text{fib}_T &= \text{Integer} \to T \text{ Integer} \\
\text{fib}_T \ 0 &= \text{L} \ 0 \\
\text{fib}_T \ 1 &= \text{L} \ 1 \\
\text{fib}_T \ n &= B \ (\text{fib}_T \ (n - 1)) \ (\text{fib}_T \ (n - 2))
\end{align*}
\]

\[
\begin{align*}
\text{sum}_T &= T \text{ Integer} \to \text{ Integer} \\
\text{sum}_T &= \text{fold}_T \ (+) \ \text{id}
\end{align*}
\]

\[
\begin{align*}
\text{fib}_1 &= \text{Integer} \to \text{ Integer} \\
\text{fib}_1 &= \text{sum}_T \circ \text{fib}_T
\end{align*}
\]
Fibonacci

More explicitly hylomorphic:

\[
\text{unfold}_T :: (b \to a + b \times b) \to (b \to T a)
\]

\[
\text{fib}_2 :: \text{Integer} \to \text{Integer} \\
\text{fib}_2 = \text{fold}_T (+) \text{id} \circ \text{unfold}_T g
\]

where

\[
g \ 0 = \text{Left} \ 0 \\
g \ 1 = \text{Left} \ 1 \\
g \ n = \text{Right} \ (n - 1, n - 2)
\]
Summary of *fold* and *unfold*:

\[
fold_L :: (a \to b \to b) \to b \to ([a] \to b)
\]

\[
unfold_L :: (b \to Maybe (a \times b)) \to (b \to [a])
\]

\[
fold_T :: (b \to b \to b) \to (a \to b) \to (T a \to b)
\]

\[
unfold_T :: (b \to a + b \times b) \to (b \to T a)
\]

Why the asymmetry?
Playing with type isomorphisms

\[
fold_L :: (a \to b \to b) \to b \to ([a] \to b)
\]

\[
\simeq (a \times b \to b) \to b \to ([a] \to b)
\]

\[
\simeq (a \times b \to b) \to (\mathbf{1} \to b) \to ([a] \to b)
\]

\[
\simeq (a \times b \to b) \times (\mathbf{1} \to b) \to ([a] \to b)
\]

\[
\simeq ((a \times b + \mathbf{1}) \to b) \to ([a] \to b)
\]

\[
\simeq (\text{Maybe} (a \times b) \to b) \to ([a] \to b)
\]

Why \text{Maybe} (a \times b)?

Because

\[
[a] \simeq \text{Maybe} (a \times (\text{Maybe} (a \times (\text{Maybe} (a \times (\ldots))))))
\]

\[
\simeq \text{Fix} (\Lambda b \to \text{Maybe} (a \times b))
\]
Regularizing

Recall:

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

A more standard interface:

\[ \text{fold}_{LF} :: (\text{Maybe} (a \times b) \to b) \to ([a] \to b) \]
\[ \text{fold}_{LF} \ h = \text{fold}_L \ (\text{curry} \ (h \circ \text{Just})) \ (h \ \text{Nothing}) \]

Now the duality emerges:

\[ \text{unfold}_L :: (b \to \text{Maybe} (a \times b)) \to (b \to [a]) \]
\[ \text{fold}_{LF} \ :: (\text{Maybe} (a \times b) \to b) \to ([a] \to b) \]

Similarly for tree fold and unfold.
List and tree *unfold* and *fold* – pictures

\[
\begin{align*}
\text{Maybe } (a \times b) & & \text{Maybe } (a \times b) \\
g & \uparrow & h \\
b & \xrightarrow{\text{unfold}_L g} [a] & [a] & \xrightarrow{\text{fold}_L h} b \\
a + b \times b & & a + b \times b \\
g & \uparrow & h \\
b & \xrightarrow{\text{unfold}_T g} Ta & Ta & \xrightarrow{\text{fold}_T h} b
\end{align*}
\]
Build up from “base functor” $F$ to fixpoint $\mu F$:

$F b \xrightarrow{g} b \xrightarrow{\text{unfold } g} \mu F$

$\mu F \xrightarrow{\text{fold } h} b$
Build up from “base functor” $f$:

$$\textbf{newtype} \; \text{Fix} \; f = \text{Roll} \{ \text{unRoll} :: f \; (\text{Fix} \; f) \}$$

$$\text{fold} :: \text{Functor} \; f \Rightarrow (f \; b \rightarrow b) \rightarrow (\text{Fix} \; f \rightarrow b)$$

$$\text{fold} \; h = h \circ \text{fmap} \; (\text{fold} \; h) \circ \text{unRoll}$$

$$\text{unfold} :: \text{Functor} \; f \Rightarrow (a \rightarrow f \; a) \rightarrow (a \rightarrow \text{Fix} \; f)$$

$$\text{unfold} \; g = \text{Roll} \circ \text{fmap} \; (\text{unfold} \; g) \circ g$$

$$\text{hylo} :: \text{Functor} \; f \Rightarrow (f \; b \rightarrow b) \rightarrow (a \rightarrow f \; a) \rightarrow (a \rightarrow b)$$

$$\text{hylo} \; h \; g = \text{fold} \; h \circ \text{unfold} \; g$$

Let’s revisit our examples.
data LF a t = NilF | ConsF a t deriving Functor

type L' a = Fix (LF a)

fact₃ :: Integer → Integer
fact₃ = hylo h g

where
  g :: Integer → LF Integer Integer
  g 0 = NilF
  g n = ConsF n (n - 1)

  h :: LF Integer Integer → Integer
  h NilF = 1
  h (ConsF n u) = n × u
data TF a t = LF a | BF t t deriving Functor

type T' a = Fix (TF a)

fib₃ :: Integer → Integer
fib₃ = hylo h g

where
  g :: Integer → TF Integer Integer
  g 0 = LF 0
  g 1 = LF 1
  g n = BF (n - 1) (n - 2)
  h :: TF Integer Integer → Integer
  h (LF n) = n
  h (BF u v) = u + v
type Range = Integer × Integer

fact₄ :: Integer → Integer

fact₄ n = hylo h g (1, n)

where

  g :: Range → TF Integer Range
  g (lo, hi) = case lo `compare` hi of
                 GT → LF 1
                 EQ → LF lo
                 LT → let mid = (lo + hi) `div` 2 in
                 BF (lo, mid) (mid + 1, hi)

  h :: TF Integer Integer → Integer
  h (LF i)    = i
  h (BF u v)  = u × v

Parallel-friendly!
Another look and *unfold* and *fold*

\[
F a \xrightarrow{F (\text{unfold } g)} F (\mu F) \\
\uparrow g \downarrow \text{Roll} \\
a \quad \text{unfold } g \quad \mu F
\]

\[
F (\mu F) \xrightarrow{F (\text{fold } h)} F b \\
\uparrow \text{unRoll} \downarrow h \\
\mu F \quad \text{fold } h \quad b
\]

**newtype** \(\text{Fix } f = \text{Roll } \{ \text{unRoll} :: f (\text{Fix } f) \} \)

\(\text{unfold} :: \text{Functor } f \Rightarrow (a \to f a) \to (a \to \text{Fix } f)\)

\(\text{unfold } g = \text{Roll} \circ \text{fmap } (\text{unfold } g) \circ g\)

\(\text{fold} :: \text{Functor } f \Rightarrow (f b \to b) \to (\text{Fix } f \to b)\)

\(\text{fold } h = h \circ \text{fmap } (\text{fold } h) \circ \text{unRoll}\)
Another look and \textit{hylo}

\[
a \overset{\text{\textit{hylo} h g}}{\longrightarrow} b
\]
Another look and *hylo*

\[ a \xrightarrow{\text{unfold } g} \mu F \xrightarrow{\text{fold } h} b \]

Definition of *hylo*.
Another look and *hylo*

By definitions of *fold* and *unfold*.
Another look and hylo

\[
F \ a \xrightarrow{F(\text{unfold } g)} F (\mu F) \xrightarrow{F(\text{fold } h)} F \ b
\]

\[
\begin{array}{c}
g \downarrow \\
a \xrightarrow{\text{unfold } g} \mu F \xrightarrow{\text{fold } h} b
\end{array}
\]

Since \textit{unRoll} and \textit{Roll} are inverses.
Another look and *hylo*

By the *Functor* law: $fmap v \circ fmap u \equiv fmap (v \circ u)$. 
Another look and *hylo*

\[
F a \quad \overset{F \, (hylo \, h \, g)}{\longrightarrow} \quad F b
\]

\[
\begin{array}{c}
g \\
\downarrow \\
a
\end{array} \quad \overset{hylo \, h \, g}{\longrightarrow} \quad \begin{array}{c}
h \\
\downarrow \\
b
\end{array}
\]

Definition of *hylo*. Directly recursive!
All together

Conal Elliott (Tabula)  Folds and unfolds all around us  Spring, 2013  30 / 1
Reversed

Conal Elliott (Tabula)  Folds and unfolds all around us  Spring, 2013  31 / 1
fold and unfold via hylo

hylo subsumes both fold and unfold:

\[ \text{unfold } g = \text{hylo } \text{Roll } g \]

\[ \text{fold } h = \text{hylo } h \text{ unRoll} \]

since

\[ \text{hylo } h \ g \equiv \text{fold } h \circ \text{unfold } g \]

and

\[ \text{fold } \text{Roll} \equiv \text{id} \equiv \text{unfold } \text{unRoll} \]
• Fold and unfold are structured replacements for the “assembly language” of recursive definitions.

• Unifying view of fold & unfold across data types via functor fixpoints.

• Recursive programs have a systematic translation to unfold and fold.

• The translation reveals parallelism clearly and simply.
A cautionary tale

I could restructure the program’s flow or use one little ‘goto’ instead.

Eh, screw good practice. How bad can it be?

goto main_sub3;

*Compile*

Conal Elliott

(Spring, 2013)
Robert Lang’s Origami BiCurve Pot 13

Maine Organic Farmers

unknown

Randall Munroe (xkcd)