Folds and unfolds all around us

Conal Elliott

Tabula

Spring, 2013
This talk is a literate Haskell program.

```
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:

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

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

\[
\text{range}_L :: \text{Integer} \to \text{Integer} \to [\text{Integer}] \\
\text{range}_L \ l \ h \ | \ l > h = [] \\
\text{otherwise} = l : \text{range}_L \ (\text{succ} \ l) \ h
\]

Conal Elliott (Tabula)  
Folds and unfolds all around us  
Spring, 2013 3 / 35
Recursive functional programming

On (binary leaf) trees:

```haskell
data T a = L a | B (T a) (T a) deriving Show

productT :: T Integer → Integer
productT (L a) = a
productT (B s t) = productT s × productT t

rangeT :: Integer → Integer → T Integer
rangeT l h | l ≡ h = L l
            | otherwise = B (rangeT l m) (rangeT (m + 1) h)
where m = (l + h) ‘div‘ 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.
Contract a structure *down to* a single value.

For lists:

\[
\begin{align*}
fold_L &: (a \to b \to b) \to b \to ([a] \to b) \\
fold_L &\ b \ [ ] \ = \ b \\
fold_L &\ f \ b \ (a : as) = f \ a \ (fold_L f \ b \ as)
\end{align*}
\]

\[
\begin{align*}
sum_L &= fold_L (+) 0 \\
product_L &= fold_L (\times) 1 \\
reverse_L &= fold_L (\lambda a \ r \to r + + [a]) [ ]
\end{align*}
\]

For trees:

\[
\begin{align*}
fold_T &: (b \to b \to b) \to (a \to b) \to (T a \to b) \\
fold_T &\ l \ (L a) \ = \ l \ a \\
fold_T &\ b \ l \ (B s t) = b \ (fold_T b \ l \ s) \ (fold_T b \ l \ t)
\end{align*}
\]

\[
\begin{align*}
product_T &= fold_T (\times) id
\end{align*}
\]
Unfolds ("anamorphisms")

Expand a structure *up from* a single value.

Lists:

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

\[
\text{unfold}_L f \ b = \text{case } f \ b \text{ of}
\]

\[
\text{Just } (a, b') \to a : \text{unfold}_L f b'
\]

\[
\text{Nothing} \quad \to []
\]

\[
\text{range}_L' :: \text{Integer} \times \text{Integer} \to [\text{Integer}]
\]

\[
\text{range}_L' = \text{unfold}_L g
\]

where

\[
g \ (l, h) \mid l > h \quad = \text{Nothing}
\]

\[
\mid \text{otherwise} = \text{Just} \ (l, (\text{succ} \ l, h))
\]
Unfolds ("anamorphisms")

Trees:

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

\[
\text{unfold}_T \ g \ x = \text{case} \ g \ x \ \text{of}
\]

\[
\text{Left } a \quad \to \quad L \ a
\]

\[
\text{Right } (c, d) \to B (\text{unfold}_T \ g \ c) (\text{unfold}_T \ g \ d)
\]

\[
\text{range}_{TP} :: \text{Integer} \times \text{Integer} \to T \text{ Integer}
\]

\[
\text{range}_{TP} = \text{unfold}_T \ g
\]

where

\[
g \ (l, h) \mid \ l \equiv h \quad = \text{Left } l
\]

\[
\mid \ \text{otherwise} = \text{Right } ((l, m), (m + 1, h))
\]

where \( m = (l + h) \, \text{`div`} \, 2 \)
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.
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 \textit{unfold} and \textit{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:

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

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

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

More explicitly hylomorphic:

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

\[ fib_2 :: Integer \to Integer \]
\[ fib_2 = fold_T (+) \circ id \circ unfold_T g \]

where

\[ g \; 0 = Left \; 0 \]
\[ g \; 1 = Left \; 1 \]
\[ g \; n = Right \; (n - 1, \; n - 2) \]
Generalizing folds and unfolds

Summary of \textit{fold} and \textit{unfold}:

\begin{align*}
\text{fold}_L & :: (a \to b \to b) \to b \to ([a] \to b) \\
\text{unfold}_L & :: (b \to \text{Maybe} (a \times b)) \to (b \to [a])
\end{align*}

\begin{align*}
\text{fold}_T & :: (b \to b \to b) \to (a \to b) \to (T a \to b) \\
\text{unfold}_T & :: (b \to a + b \times b) \to (b \to T a)
\end{align*}

Why the asymmetry?
Playing with type isomorphisms

\[ \text{fold}_L :: (a \to b \to b) \to b \to ([a] \to b) \]
\[ \cong (a \times b \to b) \to b \to ([a] \to b) \]
\[ \cong (a \times b \to b) \to (1 \to b) \to ([a] \to b) \]
\[ \cong (a \times b \to b) \times (1 \to b) \to ([a] \to b) \]
\[ \cong ((a \times b + 1) \to b) \to ([a] \to b) \]
\[ \cong (\text{Maybe} (a \times b) \to b) \to ([a] \to b) \]

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

Because

\[
\begin{align*}
[a] & \cong \text{Maybe} (a \times (\text{Maybe} (a \times (\text{Maybe} (a \times (\ldots)))))]) \\
& \cong \text{Fix} (\Lambda b \to \text{Maybe} (a \times b))
\end{align*}
\]
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

\[ \text{Maybe} \ (a \times b) \]
\[ g \]
\[ b \xrightarrow{\text{unfold}_L \ g} [a] \]
\[ a + b \times b \]
\[ g \]
\[ b \xrightarrow{\text{unfold}_T \ g} T\ a \]

\[ \text{Maybe} \ (a \times b) \]
\[ h \]
\[ [a] \xrightarrow{\text{fold}_L \ h} b \]
\[ a + b \times b \]
\[ h \]
\[ T\ a \xrightarrow{\text{fold}_T \ h} b \]
Build up from “base functor” $F$ to fixpoint $\mu F$:

$$
\begin{align*}
F(b) & \underset{g}{\xrightarrow{\text{unfold } g}} \mu F \\
\mu F & \underset{\text{fold } h}{\xrightarrow{\text{fold } h}} b
\end{align*}
$$
Build up from “base functor” $f$:

```haskell
newtype Fix f = Roll { unRoll :: f (Fix f) }

fold :: Functor f ⇒ (f b → b) → (Fix f → b)
fold h = h ◦ fmap (fold h) ◦ unRoll

unfold :: Functor f ⇒ (a → f a) → (a → Fix f)
unfold g = Roll ◦ fmap (unfold g) ◦ g

hylo :: Functor f ⇒ (f b → b) → (a → f a) → (a → b)
hylo h g = fold h ◦ unfold g
```

Let’s revisit our examples.
**Factorial via list hylo**

```haskell
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
\textbf{Factorial via tree hylo}

type Range \equiv Integer \times Integer

\texttt{fact}_4 :: Integer \to Integer
\texttt{fact}_4 \ n = \texttt{hylo} \ h \ g \ (1, n)

\textbf{where}

\begin{align*}
g &:: Range \to TF \ \text{Integer} \ \text{Range} \\
g \ (lo, hi) &\equiv \text{case} \ lo \ \text{\ 'compare' hi of} \\
& \quad GT \to LF \ 1 \\
& \quad EQ \to LF \ lo \\
& \quad LT \to \text{let} \ mid = (lo + hi) \ 'div' \ 2 \ \text{in} \\
& \quad BF \ (lo, mid) \ (mid + 1, hi) \\

h &:: TF \ \text{Integer} \ \text{Integer} \to \text{Integer} \\
h \ (LF \ i) &\equiv i \\
h \ (BF \ u \ v) &\equiv u \times v
\end{align*}

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

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

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

**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 *hylo*

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

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

Definition of *hylo*. 

Conal Elliott (Tabula) Folds and unfolds all around us Spring, 2013
By definitions of *fold* and *unfold*. 

\[
\begin{array}{c}
F a \xrightarrow{F (\text{unfold } g)} F (\mu F) \xrightarrow{F (\text{fold } h)} F b \\
g \uparrow \quad \text{unRoll} \quad \text{Roll} \quad \downarrow h \\
a \xrightarrow{\text{unfold } g} \mu F \xrightarrow{\text{fold } h} b
\end{array}
\]
Another look and *hylo*

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

Since *unRoll* and *Roll* are inverses.
Another look and *hylo*

\[
\begin{array}{ccc}
F \ a & \xrightarrow{F\ (\text{fold } h \circ \text{ unfold } g)} & F \ b \\
\uparrow g & & \downarrow h \\
\text{fold } h \circ \text{ unfold } g & \xleftarrow{\text{fold } h \circ \text{ unfold } g} & b
\end{array}
\]

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

Definition of *hylo*. Directly recursive!
All together
Conal Elliott (Tabula)  
Folds and unfolds all around us  
Spring, 2013 31 / 35
**fold and unfold via hylo**

*hylo* subsumes both *fold* and *unfold*:

\[
\begin{align*}
\text{unfold } g &= \text{hylo } \text{Roll } g \\
\text{fold } h &= \text{hylo } h \text{ unRoll}
\end{align*}
\]

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

Conal Elliott (Tabula)

Folds and unfolds all around us

Spring, 2013
Picture credits

- Robert Lang’s Origami BiCurve Pot 13
- Maine Organic Farmers
- unknown
- Randall Munroe (xkcd)