16. Fold/unfold transformation

**Summary**: This lecture discusses the fold/unfold technique for program transformation.
Transforming programs

• Producing programs which are both correct and efficient is very difficult

• One good approach is to concentrate initially on constructing a correct program $P$, then to transform $P$ into a program $P'$ that is equivalent to $P$ but more efficient
  – correctness *then* efficiency

• But how do we know whether $P$ and $P'$ are equivalent?

• Given a set of provably correct (or obviously correct) transformations, a composition (or sequence) of those transformations is also correct
  – equivalence is a *transitive* relation

• Such transformations (and sequences of transformations) are known as *correctness-preserving* transformations

• Given a function definition $F$, any definition we can derive from $F$ by correctness-preserving transformations is guaranteed to be equivalent to $F$
Transformation steps

- Given a program $P$, there are six types of correctness-preserving transformations

**unfold**

Given equations $A = A'$ and $B = B'$, such that $A'$ contains an instance of $B$:

replace $A = A'$ with $A = A'[B'/B]$

**fold**

Given equations $A = A'$ and $B = B'$, such that $A'$ contains an instance of $B'$:

replace $A = A'$ with $A = A'[B/B']$

**definition**

Introduce a new definition into $P$ *avoiding name-clashes*
Transformation steps

**instantiation**

Given an equation \( A = A' \) in \( P \) containing a bound variable \( x \):

replace \( A = A' \) with \( A[E/x] = A'[E/x] \)

for each pattern \( E \) from the type of \( x \)

The set of patterns should be *disjoint* and *complete*

**abstraction**

Given an equation \( A = A' \) in \( P \) containing exp's \( E_1 \ldots E_n \):

replace \( A = A' \) with

\[
A = A'[u_1/E_1] \ldots [u_n/E_n]
\]

where \((u_1, \ldots, u_n) = (E_1, \ldots, E_n)\)

**law application**

Given an equation \( A = A' \) in \( P \) containing an expression \( E_1 \):

replace \( A = A' \) with \( A = A'[E_2/E_1] \)

where \( E_1 = E_2 \) is a provably correct law in \( P \)

(E.g. the law can be proved by induction.)
Example 1

sum, len, average :: [Int] -> Int

sum [] = 0
sum (n : ns) = n + sum ns

len [] = 0
len (n : ns) = 1 + len ns

average l = sum l \`div` len l

av :: [Int] -> (Int, Int)

av l = (sum l, len l) definition

av [] = (sum [], len [])

av (n : ns) = (sum (n : ns), len (n : ns)) instantiation

av [] = (0, 0) 2 unfolds

av (n : ns) = (n + sum ns, 1 + len ns) 2 unfolds

av (n : ns) = (n + u, 1 + v)
where (u, v) = (sum ns, len ns) abstraction

av (n : ns) = (n + u, 1 + v)
where (u, v) = av ns fold
Example 1

average l = sum l `div` len l

average l = u `div` v
    where (u, v) = (sum l, len l)  abstraction

average l = u `div` v
    where (u, v) = av l  fold

---------------------------------------------------------------

av :: [Int] -> (Int, Int)

av [] = (0, 0)
av (n : ns) = (n + u, 1 + v)
    where (u, v) = av ns

average :: [Int] -> Int

average l = u `div` v
    where (u, v) = av l
Example 2

fib :: Int -> Int

fib 0 = 1
fib 1 = 1
fib n | n >= 2 = fib (n – 1) + fib (n – 2)

fib’ :: Int -> (Int, Int)

fib’ n = (fib n, fib (n + 1))  definition

fib’ 0 = (fib 0, fib (0 + 1))

fib’ n | n > 0 = (fib n, fib (n + 1))  instantiation

fib’ 0 = (1, 1)  2 unfolds

fib’ n | n > 0 = (fib n, fib n + fib (n – 1))  unfold

fib’ n | n > 0 = (u, u + v)
    where (v, u) = (fib (n – 1), fib n)  abstraction

fib’ n | n > 0 = (u, u + v)
    where (v, u) = fib’ (n – 1)  fold
Example 2

\[ \text{fib } n \mid n \geq 2 = \text{fib} (n - 1) + \text{fib} (n - 2) \]

\[ \text{fib } n \mid n \geq 2 = u + v \]

where \((v, u) = (\text{fib} (n - 2), \text{fib} (n - 1))\)

\[ \text{fib } n \mid n \geq 2 = u + v \]

where \((v, u) = \text{fib'} (n - 2)\)

\[
\begin{align*}
\text{fib'} \colon & \text{Int} \to (\text{Int}, \text{Int}) \\
\text{fib'} 0 &= (1, 1) \\
\text{fib'} n \mid n > 0 &= (u, u + v) \\
& \text{where } (v, u) = \text{fib'} (n - 1)
\end{align*}
\]

\[
\begin{align*}
\text{fib} \colon & \text{Int} \to \text{Int} \\
\text{fib } 0 &= 1 \\
\text{fib } 1 &= 1 \\
\text{fib } n \mid n \geq 2 &= u + v \\
& \text{where } (v, u) = \text{fib'} (n - 2)
\end{align*}
\]
Example 3

fact :: Int -> Int

fact 0 = 1
fact n | n >= 1 = n * fact (n - 1)

head :: [a] -> a
head (x : xs) = x

map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x : xs) = f x : map f xs

factlist :: Int -> [Int]
factlist n = map fact [n .. 0]

factlist 0 = map fact [0 .. 0]
factlist n | n > 0 = map fact [n .. 0]  
               instantiation

factlist 0 = [1]  
           unfold

factlist n | n > 0
            = map fact (n : (n - 1) : [n - 2 .. 0])  
              unfold

factlist n | n > 0
            = fact n : fact (n - 1) : map fact [n - 2 .. 0]  
              2 unfolds
Example 3

factlist n | n > 0
            = (n * fact (n – 1)) :
                fact (n – 1) : map fact [n – 2 .. 0]  unfold

factlist n | n > 0
            = (n * head (fact (n – 1) : map fact [n – 2 .. 0])) :
                fact (n – 1) : map fact [n – 2 .. 0]  unfold

factlist n | n > 0  = (n * head us) : us
            where us = fact (n – 1) : map fact [n – 2 .. 0]  abstraction

factlist n | n > 0  = (n * head us) : us
            where us = map fact ((n – 1) : [n – 2 .. 0])  fold

factlist n | n > 0  = (n * head us) : us
            where us = map fact [n – 1 .. 0]  fold

factlist n | n > 0  = (n * head us) : us
            where us = factlist (n – 1)  fold

---------------------------------------------------------------

factlist :: Int -> [Int]

factlist 0 = [1]

factlist n | n > 0  = (n * head us) : us
            where us = factlist (n – 1)
Example 4

data Tree a = Leaf a | Branch (Tree a) a (Tree a)

frontier :: Tree a -> [a]
frontier (Leaf x) = [x]
frontier (Branch l x r) = frontier l ++ (x : frontier r)

(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)

frontier' :: Tree a -> [a] -> [a]
frontier' t xs = frontier t ++ xs  
frontier' (Leaf x) xs = frontier (Leaf x) ++ xs
frontier' (Branch l x r) xs
  = frontier (Branch l x r) ++ xs
  = (frontier l ++ (x : frontier r)) ++ xs
  = frontier l ++ ((x : frontier r) ++ xs)

unfold

assoc. of ++
Example 4

\[
\text{frontier'} (\text{Branch } l \times r) \text{ xs} \\
= \text{frontier } l ++ (x : (\text{frontier } r ++ \text{ xs})) \quad \text{unfold}
\]

\[
\text{frontier'} (\text{Branch } l \times r) \text{ xs} \\
= \text{frontier } l ++ (x : \text{frontier'} r \text{ xs}) \quad \text{fold}
\]

\[
\text{frontier'} (\text{Branch } l \times r) \text{ xs} \\
= \text{frontier'} l (x : \text{frontier'} r \text{ xs}) \quad \text{fold}
\]

\[
\text{frontier } t = \text{frontier } t ++ [] \quad \text{identity of ++}
\]

\[
\text{frontier } t = \text{frontier'} t [] \quad \text{fold}
\]

\[
\text{frontier } :: \text{ Tree a } \rightarrow [a]
\]

\[
\text{frontier } t = \text{frontier'} t []
\]

\[
\text{frontier'} :: \text{ Tree a } \rightarrow [a] \rightarrow [a]
\]

\[
\text{frontier'} (\text{Leaf } x) \text{ xs } = x : \text{ xs}
\]

\[
\text{frontier'} (\text{Branch } l \times r) \text{ xs } = \text{frontier'} l (x : \text{frontier'} r \text{ xs})
\]