foldListProduct

Peter Marks, London HUG, April 2014
hyloListProduct

Peter Marks, London HUG, April 2014
hyloStructureProduct

Peter Marks, London HUG, April 2014
hyloCombineTree

Peter Marks, London HUG, April 2014
hyloList of OneOrBoth

Peter Marks, London HUG, April 2014
Recursion considered harmful

Peter Marks, London HUG, April 2014
Recursion considered harmful for the same reasons as goto

Peter Marks, London HUG, April 2014
Credits

Ben Moseley
Roland Zumkeller
Tim Williams
goto considered harmful

Go To Statement Considered Harmful

Edsger W. Dijkstra

Communications of the ACM, Vol. 11, No. 3, March 1968
got to considered harmful

although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity

our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed

we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process
goto considered harmful

- We need to be able to *reason* about our code
- Named control structures allow us to easily recognise abstractions
- goto is just too primitive
- At any label we do not immediately know where we came from and under what circumstances
- Easy to make jump errors
- Easy to loop without making progress
goto in action

```c
if (x < 0) goto neg;
doSomethingPositive();
goto cont;

neg:
doSomethingNegative();
cont:
carryOn();
```
goto in action

```plaintext
if (x < 0) goto neg;
doSomethingPositive();
goto cont;

neg:
doSomethingNegative();
cont:
carryOn();
```

```plaintext
if (x >= 0) {
   doSomethingPositive();
}
else {
   doSomethingNegative();
}
carryOn();
```
goto in action

sum = 0;
i = 0;
loop:
    if (i >= size) goto done;
    sum += items[i];
i++;
goto loop;
done:
return sum;
goto in action

```c
sum = 0;
for (i = 0; i < s; i++) {
    sum += items[i];
}
return sum;
```

```c
sum = 0;
if (i >= s) goto done;
sum += items[i];
i++;
goto loop;
```

```c
done:
return sum;
```
...and now to Haskell
...and now to Haskell

factorial :: Int -> Int
factorial 1 = 1
factorial n = n * factorial (n - 1)
...and now to Haskell

factorial :: Int -> Int
factorial 1 = 1
factorial n = n * factorial (n - 1)

\[ n! = \prod_{k=1}^{n} k \]
...and now to Haskell

factorial :: Int -> Int
factorial 1 = 1
factorial n = n * factorial (n - 1)

\[ n! = \prod_{k=1}^{n} k \]

factorial :: Int -> Int
factorial n = product [1..n]
Hylomorphism

hylomorphism = catamorphism \circ anamorphism
Hylomorphism

hylomorphism = catamorphism \circ \ anamorphism
Hylomorphism

hylomorphism = catamorphism \circ anamorphism
Hylomorphism

hylomorphism = catamorphism \circ anamorphism
Hylomorphism

hylomorphism = catamorphism $\circ$ anamorphism
Factorial as a hylomorphism

factorial :: Int -> Int
factorial n = product [1..n]
Factorial as a hylomorphism

factorial :: Int -> Int
factorial n = product [1..n]

factorial :: Int -> Int
factorial = foldr (*) 1 . unfoldr dec
  where
dec 0 = Nothing
dec n = Just (n, n - 1)
The problem
The problem

original :: [(Int, Bool)]
original =
  [(1000, False),
   (1001, True),
   (1004, False),
   (1006, False)]

updates :: [(Int, Operation)]
updates =
  [(1003, Delete),
   (1004, Delete),
   (1006, SetTrue)]]
Naïve solution

processNaive :: [(Int, Operation)] -> [(Int, Bool)] -> [(Int, Bool)]
processNaive us xs = foldr processOne xs us

processOne :: (Int, Operation) -> [(Int, Bool)] -> [(Int, Bool)]
processOne (i, Delete) = filter ((/= i) . fst)
processOne (i, SetTrue) = map (\(j, b) ->
  if i == j then (j, True) else (j, b))
Recursive solution

process :: [(Int, Operation)] -> [(Int, Bool)] -> [(Int, Bool)]
process _ [] [] = []
process [] xs         = xs
process us@((i, op):us') xs@((j, b):xs') = case i `compare` j of
  LT -> process us' xs
  GT -> (j, b) : process us xs'
  EQ -> case op of
        Delete   -> process us' xs'
        SetTrue  -> (j, True) : process us' xs'
A utility type and function

```
data OneOrBoth a b = A a | B b | AB a b
```
**A utility type and function**

```haskell
-- data
data OneOrBoth a b = A a | B b | AB a b

mergeOrdLists :: (Ord k) =>
 (a -> k) -> (b -> k) -> [a] -> [b] -> [OneOrBoth a b]
mergeOrdLists akey bkey as bs = unfoldr ca (as, bs)

where
    ca (as@(a:as'), bs@(b:bs')) =
        case akey a `compare` bkey b of
        LT -> Just (A a, (as', bs ))
        GT -> Just (B b, (as, bs'))
        EQ -> Just (AB a b, (as', bs'))
    ca (a:as', [])     = Just (A a, (as', []))
    ca ([],    b:bs')  = Just (B b, ([],    bs'))
    ca ([],      []    )  = Nothing
```
Hylomorphism solution

process :: [(Int, Operation)] -> [(Int, Bool)] -> [(Int, Bool)]
process ops src = mapMaybe f $ mergeOrdLists fst fst ops src

where
    f (A _) = Nothing
    f (B x) = Just x
    f (AB (_, Delete) _) = Nothing
    f (AB (_, SetTrue) (d, _)) = Just (d, True)
Advantages

- Recursion is separated from processing
- Known, named recursion pattern
- (given utility function) solution code is simpler
- Date matching is separated from operation processing
- mergeOrdLists is generic and useful elsewhere
- Still lazy
- Intermediate structure *could* be optimized away
Recursion considered harmful for the same reasons as goto

- We need to be able to *reason* about our code.
- Named control structures allow us to easily recognise abstractions.
- `goto` *Recursion* is just too primitive.
- At any label *recursion entry* we do not immediately know where we came from and under what circumstances.
- Easy to make jump errors.
- Easy to loop without making progress.
An alternative approach
An alternative approach

factorial :: Int -> Int
factorial = foldr (*) 1 . unfoldr dec

where
  dec 0 = Nothing
  dec n = Just (n, n - 1)
An alternative approach

data ListAbs e v a = ListAbs (e -> v -> a) a
An alternative approach

**data** ListAbs e v a = ListAbs (e -> v -> a) a

nil :: ListAbs e v a -> a
nil (ListAbs _ n) = n

cons :: e -> v -> ListAbs e v a -> a
cons e v (ListAbs c _) = c e v
An alternative approach

```haskell
type ListAlg e a = ListAbs e a a

cataList :: ListAlg e a -> [e] -> a

cataList alg = go

where
  go (x:xs) = cons x (go xs) alg
  go [] = nil a alg
```
An alternative approach

type ListCoAlg v e = forall a. v -> ListAbs e v a -> a

anaList :: ListCoAlg v e -> v -> [e]
anaList f v = f v x

where
  x = ListAbs (\e v -> e : f v x) []
An alternative approach

hyloList :: ListAlg e a -> ListCoAlg v e -> v -> a
hyloList a ca = cataList a . anaList ca
An alternative approach

factorial :: Int -> Int
factorial = foldr (*) 1 . unfoldr dec

where
  dec 0 = Nothing
  dec n = Just (n, n - 1)

factorial :: Int -> Int
factorial = hyloList a ca

where
  a = ListAbs (*) 1
  ca 0 = nil
  ca n = cons n (n - 1)
An alternative approach

hyloList :: ListAlg e a -> ListCoAlg v e -> v -> a
hyloList (ListAbs c n) f v = f v x
    where
        x = ListAbs (\e v -> c e (f v x)) n

factorial :: Int -> Int
factorial = hyloList a ca
    where
        a = ListAbs (*) 1
        ca 0 = nil
        ca n = cons n (n - 1)
An alternative approach

\[\text{hyloList} :: \text{ListAlg} \ e \ a \rightarrow \text{ListCoAlg} \ v \ e \rightarrow \ v \rightarrow \ a\]

\[
\text{hyloList} \ (\text{ListAbs} \ c \ n) \ f \ v \ = \ f \ v \ x
\]

\[\text{where} \]
\[x \ = \ \text{ListAbs} \ (\lambda e \ v \rightarrow c \ e \ (f \ v \ x)) \ n\]

\[\text{factorial} :: \text{Int} \rightarrow \text{Int}\]

\[\text{factorial} \ = \ \text{hyloList} \ a \ ca \]

\[\text{where} \]
\[a \ = \ \text{ListAbs} \ (*) \ 1\]
\[ca \ 0 \ = \ \text{nil}\]
\[ca \ n \ = \ \text{cons} \ n \ (n - 1)\]

Look, no lists!
...back to the problem
...back to the problem

data OneOrBothListAbs a b x r = OneOrBothListAbs
  { consA :: a -> r -> x
    , consB :: b -> r -> x
    , consAB :: a -> b -> r -> x
    , nil :: x
  }
...back to the problem

data OneOrBothListAbs a b x r = OneOrBothListAbs
  { consA :: a -> r -> x
    , consB :: b -> r -> x
    , consAB :: a -> b -> r -> x
    , nil :: x
  }

class Cofunctor f where
  cofmap :: (b -> a) -> f a -> f b

instance Cofunctor (OneOrBothListAbs a b x) where
  cofmap f OneOrBothListAbs{..} = OneOrBothListAbs
  { consA = \a r -> consA a (f r)
    , consB = \ b r -> consB b (f r)
    , consAB = \a b r -> consAB a b (f r)
    , nil = nil
  }
hyloOneOrBothList

\textbf{type} \ OneOrBothListAlg \ a \ b \ x = \ OneOrBothListAbs \ a \ b \ x \ x

\textbf{type} \ OneOrBothListCoalg \ a \ b \ r =
  \forall x. \ OneOrBothListAbs \ a \ b \ x \ r \rightarrow r \rightarrow x

hyloOneOrBothList :: OneOrBothListAlg \ a \ b \ x \rightarrow OneOrBothListCoalg \ a \ b \ r \rightarrow r \rightarrow x

\textbf{where} \ hyloOneOrBothList \ alg \ f = f'
  f' = f (cofmap f' alg)
Generalised hylomorphism

\textbf{type} Alg \ f x = f x x

\textbf{type} Coalg \ f r = \forall x . f x r \rightarrow r \rightarrow x

hylo :: (Cofunctor (f x)) \Rightarrow Alg f x \rightarrow Coalg f r \rightarrow r \rightarrow x

hylo alg f = f'

\textbf{where}

f' = f (cofmap f' alg)
Merge coalgebra

mergeOrdListsCoalg :: (Ord k) =>
    (a -> k) -> (b -> k) -> Coalg (OneOrBothListAbs a b) ([a], [b])
mergeOrdListsCoalg akey bkey OneOrBothListAbs{..} s =
    case s of
      (as@(a:as'), bs@(b:bs')) ->
        case akey a `compare` bkey b of
          LT -> consA a (as', bs)
          GT -> consB b (as, bs')
          EQ -> consAB a b (as', bs')
      (a:as', [])  -> consA a (as', [])
      ([], b:bs')  -> consB b ([], bs')
      ([], [])  -> nil
Process algebra

processAlg :: Alg (OneOrBothListAbs (Int, Operation) (Int, Bool))
                  [(Int, Bool)]
processAlg = OneOrBothListAbs{..}

where
    consA _                   r = r
    consB               v      r = v : r
    consAB (_ , Delete) (d, _) r = r
    consAB (_ , SetTrue) (d, _) r = (d, True) : r
    nil                   = []

process :: [(Int, Operation)] -> [(Int, Bool)] -> [(Int, Bool)]
process ops src =
    hylo processAlg (mergeOrdListsCoalg fst fst) (ops, src)