List Comprehensions

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

October 3, 2006

Higher-order List abstractions

```
map f []          = []
map f (x:xs)      = (f x):(map f xs)        ?
map :: (tx -> ty) -> (List tx) -> (List ty)

foldl f z []      = z
foldl f z (x:xs)  = foldl f (f z x) xs     ?
foldl :: (tz -> tx -> tz) -> tz -> (List tx) -> tz

foldr f z []      = z
foldr f z (x:xs)  = f x (foldr f z xs)     ?
foldr :: (tx -> tz -> tz) -> tz -> (List tx) -> tz

filter p []       = []
filter p (x:xs)   = if p x                      ?
                       then x:(filter p xs)
                       else filter p xs
```

Using maps and folds

1. Write **sum** in terms of **fold**

```
sum = foldr plus 0
```

2. Write **split** using **foldr**

```
split :: (List Token) -> ((List String),(List Int))
```

```
split                  = foldr f ([],[])

f (Word w)   (ws,ns) = ((w:ws),ns)
f (Number n) (ws,ns) = (ws,(n:ns))
```

3. What does function **fy** do?

```
fy xys = map second xys
second (x,y) = y
fy :: (List (t1, t2)) -> (List t2)
```


Flattening a List of Lists

```
append :: (List t) -> (List t) -> (List t)
append [] ys      = ys
append (x:xs) ys  = (x:(append xs ys))
```

```
flatten :: (List (List t)) -> (List t)
flatten []        = []
flatten (xs:xss) = append xs (flatten xss)
```

Zipping two lists

```
zipWith :: (tx -> ty -> tz) ->
                  (List tx) ->
                  (List ty) -> (List tz)

zipWith f [] []         = []
zipWith f (x:xs) (y:ys) =                    ?
                  ((f x y):(zipWith f xs ys))
```

What does `f` do?

```
f xs = zipWith append xs (init ([]:xs))
```

Suppose `xs` is:

x_0 , x_1 , $\boxed{x_2}$, ... , x_n

[] , x_0 , $\boxed{x_1}$, ... , x_{n-1}

Arithmetic Sequences: Special Lists

[1 .. 4]	≡	[1,2,3,4]	
[1,3 .. 10]	≡	[1,3,5,7,9]	
[5,4 .. 1]	≡	[5,4,3,2,1]	
[5,5 .. 10]	≡	[5,5,5,...]	?
[5 ..]	≡	[5,6,7,...]	?

Infinite Data Structures

1. `ints_from i  = i:(ints_from (i+1))`

```
nth n (x:xs) = if n == 1 then x
               else nth (n - 1) xs
   nth 50 (ints_from 1) -->  50        ?
```

2. `ones = 1:ones`

```
   nth 50 ones  -->  1                 ?
```

3. `xs = map f (a:xs)`

```
   nth 10 xs    -->  f(f...(f a)...))  ?
```

These are well defined programs in Haskell. In pH you will get an answer but the program may *not* terminate.

Primes: *The Sieve of Eratosthenes*

```
primes = sieve [2..]

sieve (x:xs) = x:(sieve (filter (p x) xs))

p x y = (y mod x) ≠ 0

      nth 100 primes
```

List Comprehensions

List Comprehensions
a convenient syntax

```
[ e |  gen, gen, ...]
```

Examples

```
[ f x | x <- xs ]
```

means `map f xs`

```
[ x | x <- xs, (p x)]
```

means `filter p xs`

```
[ f x y | x <- xs, y <- ys ]
```

means the list

```
[(f x1 y1),...(f x1 yn),
 (f x2 y1),......(f xm yn)]
```

which is defined by

```
flatten (map (\ x -> (map (\ y -> e) ys) xs))
```

Three-Partitions

Generate a list containing all three-partitions (nc1, nc2, nc3) of a number m, such that

- nc1 ≤ nc2 ≤ nc3
- nc1 + nc2 + nc3 = m

```
three_partitions m =

  [ (nc1,nc2,nc3) | nc1 <- [0..m],
                    nc2 <- [0..m],
                    nc3 <- [0..m],
                    nc1+nc2+nc3 == m,
                    nc1 <= nc2,
                    nc2 <= nc3          ]
```

?

Efficient Three-Partitions

```
three_partitions m =

  [ (nc1,nc2,nc3) | nc1 <- [0..floor(m/3)],
                    nc2 <- [nc1..floor ((m-nc1)/2)],
                    nc3 = m-nc1-nc2          ]
```

?

The Power of List Comprehensions

```
[ (i,j) | i <- [1..m], j <- [1..n] ]
```

using map

```
point i j      = (i,j)
points i       = map (point i) [1..n]
all_points     = map points [1..m]          ?
```

Is this correct?

No, we still need to flatten the list of lists.

Desugaring!

- Most high-level languages have constructs whose meaning is difficult to express precisely in a direct way
- Compilers often translate ("desugar") high-level constructs into a simpler language
- *Two examples:*
  - *List comprehensions:* eliminate List compressions usings maps etc.
  - *Pattern Matching:* eliminate complex pattern matching using simple case-expressions

List Comprehensions: Abstract Syntax

`[ e | Q ]` where **e** is an expression and **Q** is a list of generators and predicates

There are three cases on **Q**

1. First element of **Q** is a generator
   `[ e | x <- L, Q’ ]`
2. First element of **Q** is a predicate
   `[ e | B, Q’ ]`
3. **Q** is empty
   `[ e | ]`

List Comprehensions Semantics

Rule 1.1	`[e	x <- [], Q]`	⇒	`[]`
Rule 1.2	`[e	x <- (`$e_x$` : `$e_{xs}$`), Q]`	⇒	`(`*let* `x = `e_x` `*in*` [e | Q]) ++ [e	x <- `$e_{xs}$`, Q]`
Rule 2.1	`[e	False, Q]`	⇒	`[]`
Rule 2.2	`[e	True , Q]`	⇒	`[e	Q]`
Rule 3	`[e	]`	⇒	`e : []`

Desugering: *First Attempt*

TE[[[e |]]] = e :[]

TE[[[e | B, Q]]] =
    if B *then* TE[[[e | Q]]] *else* []

TE[[[e | x <- L, Q]]] =

case L *of*
    [] -> []
    t:ts -> (*let* x = t *in* TE[[[e | Q]]])
        ++ TE[[[e | x <- ~~ts,~~ Q]]]

Will unfold infinitely!
Need to be more systematic.

Not structural induction

Eliminating Generators

```
[ e | x <- xs] ⇒ map (\x-> e) xs

[ e | x <- xs, y <- ys] ⇒
   concat (map (\x-> map (\y-> e) ys) xs)
```

where **concat** flattens a list:

```
   concat []       = []
   concat (xs:xss) = xs ++ (concat xss)


[ e | x <- xs, y <- ys, z <- zs] ⇒
   concat  (map (\x->
            map (\y->
            map (\z-> e) zs) ys) xs)
```

A More General Solution

- Flatten the list after each map.
- Start the process by turning the expression into a one element list

```
[ e | x <- xs] ⇒
   concat (map (\x-> [e]) xs)

[ e | x <- xs, y <- ys] ⇒
   concat (map (\x->
   concat (map (\y-> [e]) ys)) xs )

[ e | x <- xs, y <- ys, z <- zs] ⇒
   concat (map (\x->
   concat (map (\y->
   concat (map (\z-> [e]) zs) ys) xs)
```


Eliminate the intermediate list

```
[ e | x <- xs] ⇒ concat (map (\x-> [e]) xs)
```

Notice **map** creates a list which is immediately consumed by **concat**. This intermediate list is avoided by **concatMap**

```
concatMap f []     = []
concatMap f (x:xs) = (f x) ++ (concatMap f xs)

[ e | x <- xs] ⇒ concatMap (\x-> [e]) xs

[ e | x <- xs, y <- ys] ⇒
      concatMap (\x->
      concatMap (\y-> [e]) ys) xs

[ e | x <- xs, y <- ys, z <- zs] ⇒
      concatMap (\x->
      concatMap (\y->
      concatMap (\z-> [e]) zs) ys) xs
```

List Comprehensions with Predicates

```
[ e | x <- xs, p ]
⇒ (map (\x-> e) (filter (\x-> p) xs)
⇒ concatMap (\x-> if p then [e] else []) xs

[ e | x <- xs, p, y <- ys]
⇒ concatMap (\x-> if p then
  concatMap (\y-> [e]) ys) else []) xs
```


List Comprehensions:
First Functional Implementation- Wadler

```
TE[[[ e | x <- L, Q]]] =
          concatMap (\x-> TE[[[e | Q]]]) L

TE[[[ e | B, Q]]]      =
          if B then TE[[[e | Q]]] else []

TE[[[ e | ]]]          = e :[]
```

Can we avoid concatenation altogether?

Idea: *Build the list from right-to-left*

TQ[[`[e | Q] ++ L`]]

where **L** has already been translated.

Building the output from right-to-left

```
[ e | x <- xs] ⇒
    concat (map (\x-> e) xs)
```

versus

```
[ e | x <- xs] ⇒
     let f []        = []
         f (x:xs')   = e: (f xs')
     in
         (f xs)
```


Building the output from right-to-left

```
[ e | x <- xs, y <- ys] ⇒
    concat (map (\x-> map (\y-> e) ys) xs)
```

versus

```
[ e | x <- xs, y <- ys] ⇒
     let f []        = []
         f (x:xs')   =
              let  g []      = f xs'
                   g (y:ys') = e:(g ys')
              in
                   (g ys)
     in
         (f xs)
```

List Comprehensions:
Second Functional Implementation-Wadler

```
TE[[[e | Q]]]          = TQ[[[e | Q]]] [[[]]]

TQ[[[e | x <- L1, Q]]] [[ L]] =
      let  f []       = L
           f (x:xs)   = TQ[[[e | Q]]] [[(f xs)]]
      in
         (f L1)

TQ[[[e | B, Q]]] [[L]] =
      if B then TQ[[[e | Q]]] [[L]] else L

TQ[[ [e | ] ]] [[L]]    = e : L
```

This translation is efficient because it never flattens.
The list is built right-to-left, consumed left-to-right.

The Correctness Issue

How do we decide if a translation is *correct?*

- if it produces the same answer as some reference translation, or
- if it obeys some other high-level laws

In the case of comprehensions one may want to prove that a translation satisfies the comprehension rewrite rules.