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
filter p [] = []
filter p (x:xs) = if p x
                then x:(filter p xs)
                else filter p xs
Using maps and folds

1. Write \texttt{sum} in terms of \texttt{fold} \\
\texttt{sum} = \texttt{foldr plus 0}

2. Write \texttt{split} using \texttt{foldr} \\
\texttt{split} :: \texttt{(List Token)} \rightarrow ((\texttt{List String}), (\texttt{List Int})) \\
\texttt{split} = \texttt{foldr} \texttt{f} ([],[])

\texttt{f} \texttt{(Word w)} \texttt{(ws,ns)} = ((\texttt{w:ws}), \texttt{ns}) \\
\texttt{f} \texttt{(Number n)} \texttt{(ws,ns)} = (\texttt{ws}, (\texttt{n:ns}))

3. What does function \texttt{fy} do? \\
\texttt{fy} \texttt{xys} = \texttt{map} \texttt{second} \texttt{xys} \\
\texttt{second} \texttt{(x,y)} = \texttt{y} \\
\texttt{fy} :: \texttt{(List (t1, t2))} \rightarrow \texttt{(List t2)}

Flattening a List of Lists

\texttt{append} :: \texttt{(List t)} \rightarrow \texttt{(List t)} \rightarrow \texttt{(List t)} \\
\texttt{append} \texttt{[]} \texttt{ys} = \texttt{ys} \\
\texttt{append} \texttt{(x:xs)} \texttt{ys} = (\texttt{x:(append xs ys)})

\texttt{flatten} :: \texttt{(List (List t))} \rightarrow \texttt{(List t)} \\
\texttt{flatten} \texttt{[]} = \texttt{[]} \\
\texttt{flatten} \texttt{(xs:xss)} = \texttt{append xs (flatten xss)}
Zipping two lists

\[
\text{zipWith} :: (\text{tx} \rightarrow \text{ty} \rightarrow \text{tz}) \rightarrow \\
(\text{List tx}) \rightarrow \\
(\text{List ty}) \rightarrow (\text{List tz})
\]

\[
\text{zipWith } f \; [] \; [] = []
\]
\[
\text{zipWith } f \; (x:xs) \; (y:ys) = \\
(f \; x \; y) \; : \; (\text{zipWith } f \; xs \; ys)
\]

What does \( f \) do?
\[
f \; xs = \text{zipWith append} \; xs \; (\text{init} \; (\; [] : xs))
\]

Suppose \( xs \) is:
\[
\begin{align*}
x_0, x_1, & x_2, \ldots, x_n \\
[], x_0, & x_1, \ldots, x_{n-1}
\end{align*}
\]

Arithmetic Sequences: Special Lists

\[
[1 \ldots 4] \equiv [1,2,3,4]
\]
\[
[1,3 \ldots 10] \equiv [1,3,5,7,9]
\]
\[
[5,4 \ldots 1] \equiv [5,4,3,2,1]
\]
\[
[5,5 \ldots 10] \equiv [5,5,\ldots]
\]
\[
[5 \ldots ] \equiv [5,6,7,\ldots]
\]
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) --> ?`

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

a convenient syntax

Examples

\[ [ e \mid \text{gen, gen, ...}] \]

\[ [ f x \mid x \leftarrow xs ] \]
means \( \text{map f xs} \)

\[ [ x \mid x \leftarrow xs, (p x)] \]
means \( \text{filter p xs} \)

\[ [ f x y \mid x \leftarrow xs, y \leftarrow ys ] \]
means the list
\[
[(f x_1 y_1),\ldots,(f x_1 y_n),
(f x_2 y_1),\ldots,(f x_m y_n)]
\]

which is defined by
\( \text{flatten (map (\ x \rightarrow \text{map (\ y \rightarrow e) ys}) xs)} \)
Three-Partitions

Generate a list containing all three-partitions \((nc_1, nc_2, nc_3)\) of a number \(m\), such that

- \(nc_1 \leq nc_2 \leq nc_3\)
- \(nc_1 + nc_2 + nc_3 = m\)

\[
\text{three_partitions } m = \\
[ (nc_1,nc_2,nc_3) | \begin{array}{l}
nc_1 \leftarrow [0..m], \\
nc_2 \leftarrow [0..m], \\
nc_3 \leftarrow [0..m], \\
n_1 + nc_2 + nc_3 = m, \\
n_1 \leq nc_2, \\
n_2 \leq nc_3 \\
\end{array} ]
\]

Efficient Three-Partitions

\[
\text{three_partitions } m = \\
[ (nc_1,nc_2,nc_3) | \begin{array}{l}
n_1 \leftarrow [0..\lfloor m/3 \rfloor], \\
n_2 \leftarrow [\lfloor n_1 \rfloor \ldots \lfloor (m-n_1)/2 \rfloor], \\
n_3 = m - n_1 - n_2 \\
\end{array} ]
\]
The Power of List Comprehensions

\[(i,j) \mid i \leftarrow [1..m], \ j \leftarrow [1..n]\]

using map

\[
\begin{align*}
point\ i\ j & = (i,j) \\
points\ i & = map\ (point\ i)\ [1..n] \\
all\_points & = \text{map}\ points\ [1..m]
\end{align*}
\]

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 using maps etc.
  - Pattern Matching: eliminate complex pattern matching using simple case-expressions
List Comprehensions: Abstract Syntax

\[[ e \mid Q \]\] 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
   \[
   [ e \mid x <- L, Q' \]
   \]

2. First element of \(Q\) is a predicate
   \[
   [ e \mid B, Q' \]
   \]

3. \(Q\) is empty
   \[
   [ e \mid ]
   \]

List Comprehensions Semantics

Rule 1.1
\[
[ e \mid x <- [], Q ] \Rightarrow [ ]
\]

Rule 1.2
\[
[ e \mid x <- (e_x : e_xs), Q ] \Rightarrow \\
(let x = e_x in [ e \mid Q ]) ++ \\
[ e \mid x <- e_xs, Q ]
\]

Rule 2.1
\[
[ e \mid False, Q ] \Rightarrow [ ]
\]

Rule 2.2
\[
[ e \mid True , Q ] \Rightarrow [ e \mid Q ]
\]

Rule 3
\[
[ e \mid ] \Rightarrow e : [ ]
\]
Desugering: *First Attempt*

\[
\begin{align*}
\text{TE}[[\text{e} \mid \text{L}]] & = \text{e} : [] \\
\text{TE}[[\text{e} \mid \text{B}, \text{Q}]] & = \text{if } \text{B} \text{ then } \text{TE}[[\text{e} \mid \text{Q}]] \text{ else } [] \\
\text{TE}[[\text{e} \mid \text{x} <- \text{L}, \text{Q}]] & = \\
\text{case } \text{L of} & \\
[] & \rightarrow [] \\
t:ts & \rightarrow (\text{let } x = t \text{ in } \text{TE}[[\text{e} \mid \text{Q}]])) \\
& \quad++ \text{TE}[[\text{e} \mid \text{x} <- \text{ts}, \text{Q}]]
\end{align*}
\]

Will unfold infinitely!  
Need to be more systematic.

Not structural induction

---

Eliminating Generators

\[
[\text{e} \mid \text{x} <- \text{xs}] \Rightarrow \text{map } (\text{x} \rightarrow \text{e}) \text{ xs}
\]

\[
[\text{e} \mid \text{x} <- \text{xs}, \text{y} <- \text{ys}] \Rightarrow \\
\quad \text{concat } (\text{map } (\text{x} \rightarrow \text{map } (\text{y} \rightarrow \text{e}) \text{ ys}) \text{ xs})
\]

where \text{concat} flattens a list:
\[
\begin{align*}
\text{concat } [] & = [] \\
\text{concat } (\text{xs}:\text{xss}) & = \text{xs} ++ (\text{concat } \text{xss})
\end{align*}
\]

\[
[\text{e} \mid \text{x} <- \text{xs}, \text{y} <- \text{ys}, \text{z} <- \text{zs}] \Rightarrow \\
\quad \text{concat } (\text{map } (\text{x} \rightarrow \\
\quad \text{map } (\text{y} \rightarrow \\
\quad \text{map } (\text{z} \rightarrow \text{e}) \text{ zs}) \text{ ys}) \text{ xs})
\]
A More General Solution

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

\[
[e \mid x \leftarrow xs] \Rightarrow \\
\text{concat} (\text{map} (\lambda x \rightarrow [e]) \; xs)
\]

\[
[e \mid x \leftarrow xs, y \leftarrow ys] \Rightarrow \\
\text{concat} (\text{map} (\lambda x \rightarrow \\
\text{concat} (\text{map} (\lambda y \rightarrow [e]) \; ys)) \; xs)
\]

\[
[e \mid x \leftarrow xs, y \leftarrow ys, z \leftarrow zs] \Rightarrow \\
\text{concat} (\text{map} (\lambda x \rightarrow \\
\text{concat} (\text{map} (\lambda y \rightarrow \\
\text{concat} (\text{map} (\lambda z \rightarrow [e]) \; zs) \; ys)) \; xs)
\]

Eliminate the intermediate list

\[
[e \mid x \leftarrow xs] \Rightarrow \text{concat} (\text{map} (\lambda x \rightarrow [e]) \; xs)
\]

Notice \text{map} creates a list which is immediately consumed by \text{concat}. This intermediate list is avoided by \text{concatMap}

\[
\text{concatMap} \; f \; \text{[]} = [] \\
\text{concatMap} \; f \; (x:xss) = (f \; x) \; ++ \; (\text{concatMap} \; f \; xss)
\]

\[
[e \mid x \leftarrow xs] \Rightarrow \text{concatMap} (\lambda x \rightarrow [e]) \; xs
\]

\[
[e \mid x \leftarrow xs, y \leftarrow ys] \Rightarrow \\
\text{concatMap} (\lambda x \rightarrow \\
\text{concatMap} (\lambda y \rightarrow [e]) \; ys) \; xs
\]

\[
[e \mid x \leftarrow xs, y \leftarrow ys, z \leftarrow zs] \Rightarrow \\
\text{concatMap} (\lambda x \rightarrow \\
\text{concatMap} (\lambda y \rightarrow \\
\text{concatMap} (\lambda z \rightarrow [e]) \; zs) \; ys) \; xs
\]
List Comprehensions with Predicates

\[
\begin{align*}
[ e | x <- xs, p ] & \\
& \Rightarrow (\text{map } (\lambda x \rightarrow e) \ (\text{filter } (\lambda x \rightarrow p) \ xs) \\
& \Rightarrow \text{concatMap } (\lambda x \rightarrow \text{if } p \text{ then } [e] \text{ else } []) \ xs \\
[ e | x <- xs, p, y <- ys ] & \\
& \Rightarrow \text{concatMap } (\lambda x \rightarrow \text{if } p \text{ then } \\
& \text{concatMap } (\lambda y \rightarrow [e]) \ ys \text{ else } []) \ xs
\end{align*}
\]

List Comprehensions:
First Functional Implementation- Wadler

\[
\begin{align*}
\text{TE}[\{ e | x <- L, Q \}] &= \text{concatMap } (\lambda x \rightarrow \text{TE}[\{ e \ | Q \}]) \ L \\
\text{TE}[\{ e | B, Q \}] &= \text{if } B \text{ then } \text{TE}[\{ e \ | Q \} \text{ else } []} \\
\text{TE}[\{ e \ | \}] &= e : []
\end{align*}
\]

Can we avoid concatenation altogether?

Idea: Build the list from right-to-left

\[
\text{TEQ}[\{ e \ | Q \} \ + \ L]
\]

where \( L \) has already been translated.
Building the output from right-to-left

\[ [ e \mid x \leftarrow xs] \Rightarrow \]
\[ concat (map (\times \rightarrow e) xs) \]

versus

\[ [ e \mid x \leftarrow xs] \Rightarrow \]
\[ let f [] = [] \]
\[ f (x:xs') = e: (f xs') \]
\[ in \]
\[ (f xs) \]

Building the output from right-to-left

\[ [ e \mid x \leftarrow xs, y \leftarrow ys] \Rightarrow \]
\[ concat (map (\times \rightarrow map (\times \rightarrow e) ys) xs) \]

versus

\[ [ e \mid x \leftarrow xs, y \leftarrow ys] \Rightarrow \]
\[ let f [] = [] \]
\[ f (x:xs') = \]
\[ let g [] = f xs' \]
\[ g (y:ys') = e: (g ys') \]
\[ in \]
\[ (g ys) \]
\[ in \]
\[ (f xs) \]
List Comprehensions:
Second Functional Implementation-Wadler

\[ T_E[[e | Q]] = T_Q[[e | Q]] [[[]]] \]

\[ T_Q[[e | x <- L_1, Q]] [[L]] = \]

\[ \begin{align*}
& \text{let } f [] = L \\
& f (x:xs) = T_Q[[e | Q]] [[(f xs)]] \\
& \text{in } \\
& (f L_1)
\end{align*} \]

\[ T_Q[[e | B, Q]] [[L]] = \begin{cases} 
T_Q[[e | Q]] [[L]] & \text{if } B \\
\text{else } L
\end{cases} \]

\[ T_Q[[e | ]] [[L]] = e : L \]

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.