PROGRAMMING IN HASKELL

Chapter 5 - List Comprehensions

Set Comprehensions

In mathematics, the <u>comprehension</u> notation can be used to construct new sets from old sets.

$\{x^2 \mid x \in \{1...5\}\}$

The set $\{1,4,9,16,25\}$ of all numbers x^2 such that x is an element of the set $\{1...5\}$.

Lists Comprehensions

In Haskell, a similar comprehension notation can be used to construct new <u>lists</u> from old lists.

[x^2 | x ← [1..5]]

The list [1,4,9,16,25] of all numbers x^2 such that x is an element of the list [1..5].

Note:

- The expression x ← [1..5] is called a <u>generator</u>, as it states how to generate values for x.

- Comprehensions can have <u>multiple</u> generators, separated by commas. For example:

```
> [(x,y) | x ← [1,2,3], y ← [4,5]]

[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

- Changing the order of the generators changes the order of the elements in the final list:

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

- Multiple generators are like nested loops, with later generators as more deeply nested loops whose variables change value more frequently.

- For example:

```
> [(x,y) | y ← [4,5], x ← [1,2,3]]

[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

x ← [1,2,3] is the last generator, so the value of the x component of each pair changes most frequently.

Dependant Generators

Later generators can <u>depend</u> on the variables that are introduced by earlier generators.

$[(x,y) \mid x \leftarrow [1..3],\ y \leftarrow [x..3]]$

The list [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)] of all pairs of numbers (x,y) such that x,y are elements of the list [1..3] and $y \geq x$.

Using a dependant generator we can define the library function that concatenates a list of lists:

```
concat     :: [[a]] → [a]
concat xss = ?
```

For example:

```
> concat [[1,2,3],[4,5],[6]]

[1,2,3,4,5,6]
```

Guards

List comprehensions can use guards to restrict the values produced by earlier generators.

$[x \mid x \leftarrow [1..10], \text{even } x]$

The list [2,4,6,8,10] of all numbers x such that x is an element of the list [1..10] and x is even.

Using a guard we can define a function that maps a positive integer to its list of factors:

```
factors   :: Int → [Int]
factors n = ?
```

For example:

```
> factors 15

[1,3,5,15]
```

Hint: Using n `mod` x == 0 checks whether the remainder of integer division is 0.

A positive integer is <u>prime</u> if its only factors are 1 and itself. Hence, using factors we can define a function that decides if a number is prime:

```
prime  :: Int → Bool
prime n = ?
```

For example:

```
> prime 15
False

> prime 7
True
```

Using a guard we can now define a function that returns the list of all primes up to a given limit:

```
primes   :: Int → [Int]
primes n = [x | x ← [2..n], prime x]
```

For example:

```
> primes 40

[2,3,5,7,11,13,17,19,23,29,31,37]
```

The Zip Function

A useful library function is zip, which maps two lists to a list of pairs of their corresponding elements.

```
zip :: [a] → [b] → [(a,b)]
```

For example:

```
> zip ['a','b','c'] [1,2,3,4]

[('a',1),('b',2),('c',3)]
```

Using zip we can define a function returns the list of all <u>pairs</u> of adjacent elements from a list:

```
pairs    :: [a] → [(a,a)]
pairs xs = zip xs (tail xs)
```

For example:

```
> pairs [1,2,3,4]

[(1,2),(2,3),(3,4)]
```

Using pairs we can define a function that decides if the elements in a list are <u>sorted</u>:

```
sorted    :: Ord a ⇒ [a] → Bool
sorted xs =
   and [x ≤ y | (x,y) ← pairs xs]
```

For example:

```
> sorted [1,2,3,4]
True

> sorted [1,3,2,4]
False
```

Using zip we can define a function that returns the list of all <u>positions</u> of a value in a list:

```
positions :: Eq a ⇒ a → [a] → [Int]
positions x xs = ?
```

For example:

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```

String Comprehensions

A string is a sequence of characters enclosed in double quotes. Internally, however, strings are represented as lists of characters.

"abc" :: String

Means ['a', 'b', 'c'] :: [Char].

Because strings are just special kinds of lists, any polymorphic function that operates on lists can also be applied to strings. For example:

```
> length "abcde"
5

> take 3 "abcde"
"abc"

> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

Similarly, list comprehensions can also be used to define functions on strings, such counting how many times a character occurs in a string:

```
count      :: Char → String → Int
count x xs =
   length [x' | x' ← xs, x == x']
```

For example:

```
> count 's' "Mississippi"
4
```

Exercises

(1)

A triple (x,y,z) of positive integers is called pythagorean if $x^2 + y^2 = z^2$. Using a list comprehension, define a function

```
pyths :: Int → [(Int,Int,Int)]
```

that maps an integer n to all such triples with components in [1..n]. For example:

```
> pyths 5
[(3,4,5),(4,3,5)]
```

(2) A positive integer is perfect if it equals the sum of all of its factors, excluding the number itself. Using a list comprehension, define a function

```
perfects :: Int → [Int]
```

that returns the list of all perfect numbers up to a given limit. For example:

```
> perfects 500

[6,28,496]
```

(3) The <u>scalar product</u> of two lists of integers xs and ys of length n is give by the sum of the products of the corresponding integers:

$$\sum_{i=0}^{n-1} (xs_i * ys_i)$$

Using a list comprehension, define a function that returns the scalar product of two lists.

PROGRAMMING IN HASKELL

Chapter 7 - Higher-Order Functions

Introduction

A function is called higher-order if it takes a function as an argument or returns a function as a result.

```
twice     :: (a → a) → a → a
twice f x = f (f x)
```

twice is higher-order because it
takes a function as its first argument.

Why Are They Useful?

- Common programming idioms can be encoded as functions within the language itself.

- Domain specific languages can be defined as collections of higher-order functions.

- Algebraic properties of higher-order functions can be used to reason about programs.

The Map Function

The higher-order library function called map applies a function to every element of a list.

```
map :: (a → b) → [a] → [b]
```

For example:

```
> map (+1) [1,3,5,7]

[2,4,6,8]
```

The map function can be defined in a particularly simple manner using a list comprehension:

```
map f xs = ?
```

Alternatively, for the purposes of proofs, the map function can also be defined using recursion:

```
map f []     = ?
map f (x:xs) = ?
```

The Filter Function

The higher-order library function <u>filter</u> selects every element from a list that satisfies a predicate.

```
filter :: (a → Bool) → [a] → [a]
```

For example:

```
> filter even [1..10]

[2,4,6,8,10]
```

Filter can be defined using a list comprehension:

```
filter p xs = ?
```

Alternatively, it can be defined using recursion:

```
filter p []      = []
filter p (x:xs)
   | p x         = x : filter p xs
   | otherwise   = filter p xs
```

The Foldr Function

A number of functions on lists can be defined using the following simple pattern of recursion:

```
f []     = v
f (x:xs) = x ⊕ f xs
```

f maps the empty list to some value v, and any non-empty list to some function $\oplus$ applied to its head and f of its tail.

For example:

```
sum []     = 0
sum (x:xs) = x + sum xs
```

V = 0
$\oplus$ = +

```
product []     = 1
product (x:xs) = x * product xs
```

V = 1
$\oplus$ = *

```
and []     = True
and (x:xs) = x && and xs
```

V = True
$\oplus$ = &&

The higher-order library function foldr (fold right) encapsulates this simple pattern of recursion, with the function $\oplus$ and the value v as arguments.

For example:

```
sum     = foldr (+) 0

product = foldr (*) 1

or      = foldr (||) False

and     = foldr (&&) True
```

Foldr itself can be defined using recursion:

```
foldr :: (a → b → b) → b → [a] → b

foldr f v []     = ?

foldr f v (x:xs) = ?
```

However, it is best to think of foldr <u>non-recursively</u>, as simultaneously replacing each (:) in a list by a given function, and [] by a given value.

For example:

```
    sum [1,2,3]
=
    foldr (+) 0 [1,2,3]
=
    foldr (+) 0 (1:(2:(3:[])))
=
    1+(2+(3+0))
=
    6
```

Replace each (:)
by (+) and [] by 0.

For example:

```
  product [1,2,3]
=
  foldr (*) 1 [1,2,3]
=
  foldr (*) 1 (1:(2:(3:[])))
=
  1*(2*(3*1))
=
  6
```

Replace each (:) by (*) and [] by 1.

Other Foldr Examples

Even though foldr encapsulates a simple pattern of recursion, it can be used to define many more functions than might first be expected.

Recall the length function:

```
length        :: [a] → Int
length []      = 0
length (_:xs) = 1 + length xs
```

For example:

```
    length [1,2,3]
=
    length (1:(2:(3:[])))
=
    1+(1+(1+0))
=
    3
```

Hence, we have:

```
length = foldr (λ_ n → ?) ?
```

Now recall the reverse function:

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

For example:

```
  reverse [1,2,3]
=
  reverse (1:(2:(3:[])))
=
  (([] ++ [3]) ++ [2]) ++ [1]
=
  [3,2,1]
```

Hence, we have:

```
reverse =
  foldr (λx xs → ?) ?
```

Finally, we note that the append function (++) has a particularly compact definition using foldr:

```
(++ ys) = foldr (:) ys
```

Replace each (:) by (:) and [] by ys.

Why Is Foldr Useful?

- Some recursive functions on lists, such as sum, are <u>simpler</u> to define using foldr.
- Properties of functions defined using foldr can be proved using algebraic properties of foldr, such as <u>fusion</u> and the <u>banana split</u> rule.
- Advanced program <u>optimisations</u> can be simpler if foldr is used in place of explicit recursion.

Other Library Functions

The library function (.) returns the composition of two functions as a single function.

```
(.)   :: (b → c) → (a → b) → (a → c)
f . g  = λx → f (g x)
```

For example:

```
odd :: Int → Bool
odd  = not . even
```

The library function <u>all</u> decides if every element of a list satisfies a given predicate.

```
all      :: (a → Bool) → [a] → Bool
all p xs = and [p x | x ← xs]
```

For example:

```
> all even [2,4,6,8,10]

True
```

Dually, the library function any decides if at least one element of a list satisfies a predicate.

```
any      :: (a → Bool) → [a] → Bool
any p xs = or [p x | x ← xs]
```

For example:

```
> any (== ' ') "abc def"

True
```

The library function takeWhile selects elements from a list while a predicate holds of all the elements.

```
takeWhile :: (a → Bool) → [a] → [a]
takeWhile p xs = ?
```

For example:

```
> takeWhile (/= ' ') "abc def"

"abc"
```

Dually, the function <u>dropWhile</u> removes elements while a predicate holds of all the elements.

```
dropWhile :: (a → Bool) → [a] → [a]
dropWhile p []     = []
dropWhile p (x:xs)
   | p x           = dropWhile p xs
   | otherwise     = x:xs
```

For example:

```
> dropWhile (== ' ') "   abc"

"abc"
```

Exercises

(1) What are higher-order functions that return functions as results better known as?

(2) Express the comprehension [f x | x ← xs, p x] using the functions map and filter.

(3) Redefine map f and filter p using foldr.