CSE 3302 Programming Languages

Functional Programming Language: Haskell (cont'd)

Chengkai Li
Spring 2008

Defining Functions

Conditional Expressions

As in most programming languages, functions can be defined using conditional expressions.

```
abs   :: Int → Int
abs n = if n ≥ 0 then n else -n
```

abs takes an integer n and returns n if it is non-negative and -n otherwise.

Conditional expressions can be nested:

```
signum   :: Int → Int
signum n = if n < 0 then -1 else
              if n == 0 then 0 else 1
```

Note:

- In Haskell, conditional expressions must always have an else branch, which avoids any possible ambiguity problems with nested conditionals.

Guarded Equations

As an alternative to conditionals, functions can also be defined using guarded equations.

```
abs n | n >= 0     = n
      | otherwise  = -n
```

As previously, but using guarded equations.

Guarded equations can be used to make definitions involving multiple conditions easier to read:

```
signum n | n < 0     = -1
         | n == 0    = 0
         | otherwise = 1
```

Note:

- The catch all condition otherwise is defined in the prelude by otherwise = True.

Pattern Matching

Many functions have a particularly clear definition using pattern matching on their arguments.

```
not       :: Bool → Bool
not False = True
not True  = False
```

not maps False to True, and True to False.

Functions can often be defined in many different ways using pattern matching. For example

```
(&&)           :: Bool → Bool → Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

can be defined more compactly by

```
True  && True = True
_     && _    = False
```

```
False && _ = False
True  && b = b
```

⌘ The underscore symbol _ is the wildcard pattern that matches any argument value.

List Patterns

In Haskell, every non-empty list is constructed by repeated use of an operator : called "cons" that adds a new element to the start of a list.

```
[1,2,3]
```

Means 1:(2:(3:[])).

Note: ++ is another list concatenation operator that concatenates two lists

```
[1,4] ++ [5,3]
```

Result is [1,4,5,3].

The cons operator can also be used in patterns, in which case it destructs a non-empty list.

```
head        :: [a] -> a
head (x:_)  = x

tail        :: [a] -> [a]
tail (_:xs) = xs
```

head and tail map any non-empty list to its first and remaining elements.

List Comprehensions

List Comprehension

```
> [1..10]

[1,2,3,4,5,6,7,8,9,10]
```

Lists Comprehensions

List comprehension can be used to construct new lists from old lists.

In mathematical form {f(x) | x∊s ∧ p(x)}

```
[x^2 | x <- [1..5]]
```

```
i.e.,{ 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].

Generators

- The expression x ← [1..5] is called a generator, as it states how to generate values for x.
- Comprehensions can have multiple generators, separated by commas. For example:

```
> [(x,y) | x <- [1..3], y <- [1..2]]
[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]
```


Order Matters

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

```
> [(x,y) | y <- [1..2], x <- [1..3]]
[(1,1),(2,1),(3,1),(1,2),(2,2),(3,2)]
```

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

Dependant Generators

Later generators can depend on the variables that are introduced by earlier generators.

```
[(x,y) | x <- [1..3], y <- [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 x ≤ y.

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

```
concat     :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]
```

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 | x <- [1..10], 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 = [x | x <- [1..n]
               , n `mod` x == 0]
```

For example:

```
> factors 15

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



```
prime   :: Int -> Bool
prime n = factors n == [1,n]
```

For example:

```
> prime 15
False

> prime 7
True
```



```
primes   :: Int -> [Int]
primes n = [x | x <- [1..n], prime x]
```

For example:

```
> primes 40

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


Recursive Functions


```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

factorial maps 0 to 1, and any other integer to the product of itself with the factorial of its predecessor.

For example:

```
  factorial 3
=
  3 * factorial 2
=
  3 * (2 * factorial 1)
=
  3 * (2 * (1 * factorial 0))
=
  3 * (2 * (1 * 1))
=
  3 * (2 * 1)
=
  3 * 2
=
  6
```

Recursion on Lists

Recursion is not restricted to numbers, but can also be used to define functions on lists.

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

product maps the empty list to 1, and any non-empty list to its head multiplied by the product of its tail.

For example:

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


Quicksort

The quicksort algorithm for sorting a list of integers can be specified by the following two rules:

- The empty list is already sorted;
- Non-empty lists can be sorted by sorting the tail values ≤ the head, sorting the tail values > the head, and then appending the resulting lists on either side of the head value.

++ is list concatenation

```
qsort        :: [Int] -> [Int]
qsort []      = []
qsort (x:xs) = qsort [a | a <- xs, a <= x]
               ++
               [x]
               ++
               qsort [b | b <- xs, b > x]
```

- This is probably the simplest implementation of quicksort in any programming language!

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.

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 factorial [1,3,5]

[1,6,120]
```


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

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

Alternatively, the map function can also be defined using recursion:

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


The Filter Function

The higher-order library function filter 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 = [x | x <- xs, p x]
```

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 a value v, and any non-empty list to a function ⊕ applied to its head and f of its tail.

For example:

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

v = 0
⊕ = +

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

v = 1
⊕ = *

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

v = True
⊕ = &&

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

For example:

```
sum     = foldr (+) 0

product = foldr (*) 1

and     = foldr (&&) True
```



```
Foldr: ⊕ is right-associative

Foldl: ⊕ is left-associative
```

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

foldl (-) 1 [2,3,4]

(section 3.3.2 in the tutorial)
```