Functional Programming Language: Haskell (cont’d)

Chengkai Li
Spring 2008

Conditional Expressions

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

\[
\text{abs} :: \text{Int} \rightarrow \text{Int} \\
\text{abs} \ n = \text{if } n \geq 0 \text{ then } n \text{ else } -n
\]

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

Conditional expressions can be nested:

\[
\text{signum} :: \text{Int} \rightarrow \text{Int} \\
\text{signum} \ n = \text{if } n < 0 \text{ then } -1 \text{ else if } n == 0 \text{ then } 0 \text{ else } 1
\]

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.

\[
\text{abs} \ n | n \geq 0 = n \\
| \text{otherwise} = -n
\]

As previously, but using guarded equations.

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

\[
\text{signum} \ n | n < 0 = -1 \\
| n == 0 = 0 \\
| \text{otherwise} = 1
\]

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.

\[
\text{not} :: \text{Bool} \rightarrow \text{Bool} \\
\text{not False} = \text{True} \\
\text{not True} = \text{False}
\]

not maps False to True, and True to False.

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

\[[1,4] \text{ ++ } [5,3]\]

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

List Comprehensions

List Comprehensions

> [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: \( \{ f(x) \mid x \in S \land p(x) \} \)

\[ x^2 \mid x \leftarrow [1..5] \]

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

\[ \{(x, y) \mid x \leftarrow [1..3], y \leftarrow [1..2]\} \]

\[ \{(1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2)\} \]

Order Matters

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

\[ (x, y) \mid y \leftarrow [1..2], x \leftarrow [1..3] \]

\[ [(1,1), (2,1), (3,1), (1,2), (2,2), (3,2)] \]

- 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.

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

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

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:

```haskell
factors :: Int -> [Int]
factors n = [x | x <- [1..n], n `mod` x == 0]
```

For example:

```haskell
> factors 15
[1,3,5,15]
```

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

For example:

```haskell
> prime 15
False
> prime 7
True
```

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

For example:

```haskell
> primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
```

**Recursive Functions**

```haskell
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:

```haskell
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.

\[
\text{product} :: \text{[Int]} \to \text{Int}
\]

\[
\text{product} [] = 1
\]

\[
\text{product} (x:xs) = x \times \text{product} \; xs
\]

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

For example:

\[
\text{product} [1, 2, 3] =
\]

\[
\text{product} (1:(2:(3:[]))) =
\]

\[
1 \times \text{product} (2:(3:[])) =
\]

\[
1 \times (2 \times \text{product} (3:[])) =
\]

\[
1 \times (2 \times (3 \times \text{product} [])) =
\]

\[
1 \times (2 \times (3 \times 1)) =
\]

\[
1 \times (2 \times 3) = 6
\]

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.

\[
\text{qsort} :: \text{[Int]} \to \text{[Int]}
\]

\[
\text{qsort} [] = []
\]

\[
\text{qsort} (x:xs) = \text{qsort} [a \mid a \leftarrow xs, a < x] \mathbin{\text{++}} [x] \mathbin{\text{++}} \text{qsort} [b \mid b \leftarrow xs, b > x]
\]

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

Higher-Order Functions

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

\[
\text{twice} :: (a \to a) \to a \to a
\]

\[
\text{twice} \; f \; x = f \; (f \; x)
\]

\text{twice} is higher-order because it takes a function as its first argument.
The Map Function

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

$$\texttt{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$

For example:

- \texttt{map factorial [1,3,5] = [1,6,120]}

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

$$\texttt{map f xs = \{ f x | x \leftarrow xs \}}$$

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

- \texttt{map f [] = []}
- \texttt{map f (x:xs) = f x : map f xs}

The Filter Function

The higher-order library function \texttt{filter} selects every element from a list that satisfies a predicate.

$$\texttt{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$$

For example:

- \texttt{filter even [1..10] = [2,4,6,8,10]}

Filter can be defined using a list comprehension:

$$\texttt{filter p xs = \{ x | x \leftarrow xs, p x \}}$$

Alternatively, it can be defined using recursion:

- \texttt{filter p [] = []}
- \texttt{filter p (x:xs) = if p x then x : filter p xs else filter p xs}

The Foldr Function

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

- \texttt{f [] = v}
- \texttt{f (x:xs) = x \oplus f xs}

For example:

- \texttt{sum [] = 0}
- \texttt{sum (x:xs) = x + sum xs}
- \texttt{product [] = 1}
- \texttt{product (x:xs) = x * product xs}
- \texttt{and [] = True}
- \texttt{and (x:xs) = x && and xs}
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)