Introduction to Haskell II

Rolf Fagerberg

Spring 2005

Operators

Operators = built-in set of functions with short non-letter names.

Examples: `+` (addition), `-` (subtraction), `++` (list concatenation).

Most have two parameters and are written using *infix* notation:

```
2 + 3        ← infix
add 2 3      ← usual prefix notation for functions
```

We can convert between “operator” and “standard” version of two parameter functions

Def:

```
add x y = x + y
```

```
add 2 3      ⇝  5
(+) 2 3      ⇝  5
2 ‘add‘ 3    ⇝  5
```

Associativity and Binding Power

To save on parentheses, operators (along with function application) are given diffent *binding powers*:

```
2 * 3 + f 4 ^ 2  =  ((2 * 3) + ((f 4) ^ 2))
```

To resolve evaluation order of sequences of operators of equal binding power, they have an associativity assigned:

```
4 + 3 + 2 + 1  =  (((4 + 3) + 2) + 1)
4 - 3 - 2 - 1  =  (((4 - 3) - 2) - 1)
4 ^ 3 ^ 2 ^ 1  =  (4 ^ (3 ^ (2 ^ 1)))
```

So `+` and `-` are *left associative*, whereas `^` is *right associative*.

Do-it-yourself operators

You can define new operators (see Appendix C for rules).

Example: Minimum operator:

```
(??) :: Int -> Int -> Int
x ?? y
  | x > y       = y
  | otherwise = x
```

Now:

```
3 ?? 4  ⇝  3
```

Define associativiy and binding power:

```
infixl 7 ??
```

Pattern Matching

Definitions may use *pattern matching* on the parameters:

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

fliptuple (x,y) = (y,x)

onAxe (0,y) = True
onAxe (x,0) = True
onAxe (x,y) = False

onAxe (0,_) = True
onAxe (_,0) = True
onAxe (_,_) = False
```

```
or True _ = True
or _ True = True
or _ _    = False

sum ::  [Int] -> Int
sum []       = 0
sum (x:xs) = x + sum xs

sum [1,2,3]  ⇝  6
sum []  ⇝  0
```

Pattern Matching

A pattern is made of:

- Literals `24`, `True`, `'s'`, `[]`
- Identifiers `x`, `y` (wild card `_` is a nameless variable)
- Tuple constructor `(x,y,z)`
- List constructor `(x:xs)`
- More constructors later. . .

A pattern can be hierarchical: `("hi",(x:(x':xs),(2,0)))`

A pattern can match or fail. To match, all sub-patterns must recursively match. When a match occurs, any matched identifiers are bound to the value matched.

Polymorphism

Types can be *parametric*

```
concat ::  [[Int]] -> [Int]
concat []     = []
concat (x:xs) = x ++ concat xs

concat [[1,2],[4,5,6]]  ⇝  [1,2,4,5,6]

concat ::  [[a]] -> [a]
concat []     = []
concat (x:xs) = x ++ concat xs

zip ::  [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) :  zip xs ys
zip (x:xs) []     = []
zip []     zs     = []

zip [1,2,3] [’a’,’b’]  ⇝  [(1,’a’),(2,’b’)]
```

Functions as parameters and results

In Haskell, functions are values (value $\sim$ expression trees with empty leaves).

Can be passed to and from functions (then called high-order functions).

Very useful high-order functions:

`map`, `filter`, `zipWith`, `foldl`, `foldr`, `foldl1`, `foldr1`

```
map ::  (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x :  map f xs
```

Functions as parameters and results

Generating functions as results:

- Composition:

```
f = g .  h
twice f = f .  f
```

- Partial application (currying):

```
add ::  Int -> Int -> Int
add x y = x + y

addOne ::  Int -> Int
addOne = add 1   or
addOne = (1+)

addOneAll ::  [Int] -> [Int]
addOneAll = map (add 1)
```

Some Library Functions in Prelude

Check *A Tour of the Haskell Prelude*
See
`http://www.cs.uu.nl/~afie/haskell/tourofprelude.html`