Introduction to Haskell

Rolf Fagerberg

Spring 2006

Haskell

Functional language (no assignments)

- Purely functional
- Statically typed
- Rich typesystem
- Lazy (infinite data structures OK)

Named after Haskell Brooks Curry (1900–1982, USA, mathematical logic).

Language in development. Haskell-1998: frozen version (used here). Concrete implementation: Hugs interpreter + libraries.

Functions

Math:

$$f(x) = 2x + 5$$ ← definitions

$$g(y, z) = yz^2 + z + 2$$ ⋮

$$\text{abs}(x) = \begin{cases} x & \text{, if } x \geq 0 \\ -x & \text{, otherwise} \end{cases}$$

$$\text{abs}(f(g(3, 2)))$$ ←evaluation

Haskell:

```
f x = 2*x + 5
g y z = y*z^2 + z + 2
abs x
  | x >= 0     =  x
  | otherwise  = -x
```

← definitions ⋮

```
abs(f(g 3 2))
```

←evaluation

Hugs

Interpreter (+ libraries) for Haskell-1998.

Reads **definitions** in script file(s).

Evaluates expressions written in its shell using definitions in script and in built-in definitions in standard library `Prelude.hs`

Note: definitions cannot be given at command line, only in scripts.

Types

Math:

$$3.0 \in R$$

$$g: R \times R \rightarrow R$$
$$g(y, z) = yz^2 + z + 2$$

Haskell:

`3.0` is of type `Float`

```
g ::  Float -> Float -> Float
g y z = y*z^2 + z + 2
```

Haskell Basic Elements

Names (identifiers, “variables”) associated with **Values** (integers, booleans, strings, and also functions)

Each value belongs to a **Type** (a domain/set of values)

Definitions associate names with values.

Literals and other **Constructors** creates basic values.

Functions (including **operators**: $+$, $*$, $\ldots$) take values to new values

Evaluation of **Expressions** build using basic values and functions.

Haskell

Literals:

`277, -3.141527, 7.89e-6, 'A',"Hello World"`

Built-In Types

`Int, Bool, Float, Double, Integer, Rational, Complex,...`

Type Constructors (even more to come)

Lists (∼ arrays): `[]`

```
a :: [Int]
a = [1,2,3]
```

Tuples (∼ records): `()`

```
b :: (Char,Bool,Int)
b = ('A', True,1)
```

Some Haskell Syntax

- Off-side rule (indentation gives block structure)
- Comments:

  Single line: `-- ...comment...`
  Block Comment: `{- ...comment... -}`

- Identifiers: Letter [Letter, Digit, `_` , `'`]*

  Value names, parameters, (type parameters):

  Small initial letter

  Type names, (constructors, modules, type classes):

  Capital initial letter

- Some words reserved (`case, class, data, default, deriving, do, else, if, import, in, infix, infixl, infixr, instance, let, module, newtype, of, then, type, where`)

Recursion

No assignments $\Rightarrow$ no loops

(Loops over lists exist - see *list comprehensions* below)

Hence, in functional programming, *recursion* is used a lot.

```
power2 :: Int -> Int
power2 n
  | n==0        = 1
  | n>0         = 2 * power2 (n-1)
```

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`	$\longleftarrow$ infix
`add 2 3`	$\longleftarrow$ 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`	$\leadsto$	`5`
`(+) 2 3`	$\leadsto$	`5`
``2 `add` 3``	$\leadsto$	`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))
```

Haskell has nine levels of binding powers (9 is strongest).
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. Example: Minimum operator:

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

Now:

$$3 \;??\; 4 \;\rightsquigarrow\; 3$$

Define associativiy and binding power: `infixl 7 ??`

The names of operators must be created using the following characters:

```
!#$%&*+./<=>?@\^|-~
```

Pattern Matching

Definitions may use *pattern matching* on the parameters (often more elegant than guards):

```
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] $\leadsto$ 6
sum [] $\leadsto$ 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]] $\leadsto$ [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'] $\leadsto$ [(1,'a'),(2,'b')]

Functions as parameters and results

In Haskell, functions are values.

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

Very useful high-order functions (most discussed later):

`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)
```