Lecture 8: Introduction to Haskell

Viliam Lisý

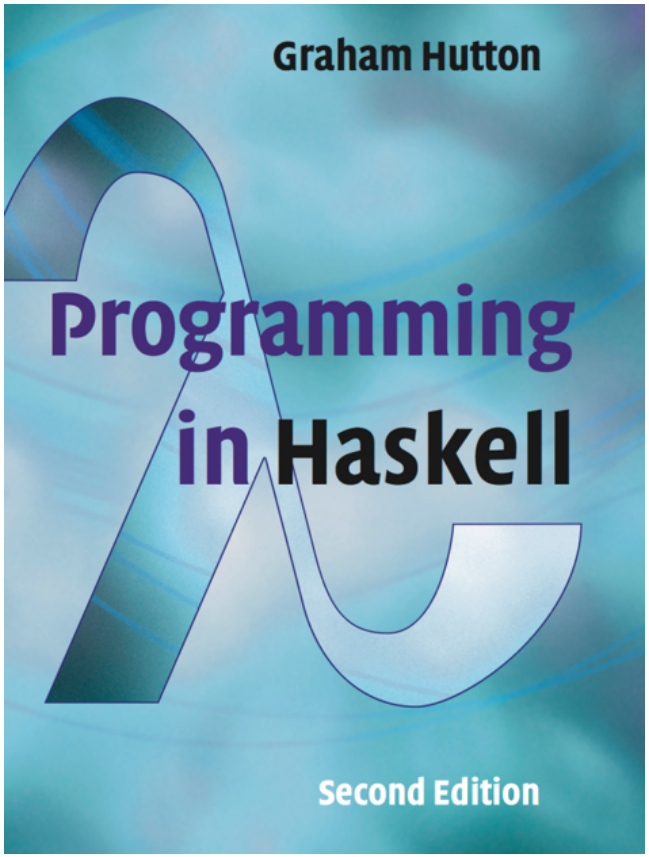
Artificial Intelligence Center
Department of Computer Science, Faculty of Electrical Eng.
Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

April, 2020

Acknowledgement

Slides for next few lectures based on slides for the book:



Why Haskell?

- Purely functional language
  - promotes understanding the paradigm
- Rich syntactic sugar (contrast to Lisp)
- Rich system of types (contrast to Lisp)
- Most popular purely functional language
- Fast prototyping of complex systems
- Active user community
  - Haskell platform, packages, search
  - `https://haskell.org/`

Main properties

- Purely functional language
  - necessary exceptions (IO) wrapped as monads
- Statically typed
  - types are derived and checked at compile time
  - types can be automatically inferred
  - have a crucial role in controlling flow of the program
- Lazy
  - function argument evaluated only when needed
  - almost everything is initially a thunk

Haskell standard

Haskell is a standardization of ideas in over a dozen of pre-existing (often proprietary) lazy functional languages

- Haskell 1.0 to 1.4
  - standardization efforts since 1990
- Haskell 98
  - first stable standard
- Haskell 2010
  - minor changes based on best practices in existing implementations
    - integration with other programming languages
    - hierarchical module names
    - pattern guards

Haskell implementations

Glasgow Haskell Compiler (GHC)

- the leading implementation of Haskell
- comprises a compiler and interpreter
- written in Haskell
- is freely available from: `www.haskell.org/platform`

Haskell Users Gofer System (Hugs)

- small and portable interpreter
- Windows version with simple GUI called WinHugs
- no longer in development

Starting GHCi

The interpreter can be started from the terminal command prompt $ by simply typing ghci:

```
$ ghci

GHCi, version X: http://www.haskell.org/ghc/  :? for help

Prelude>
```

The GHCi prompt ">" means that the interpreter is now ready to evaluate an expression.

Basic interaction

- REPL interaction as in scheme
- Common infix syntax
- Space denotes function application
- Infix operators have priorities
  - function application is first
  - otherwise use brackets
- Left associativity for functions (as in lambda calculus)
- Up arrow recalls the last entered expression

Lists

The basic data structure

```
[1,2,3,4,5]
[1..10]
[1,3..]
```

- Build by "cons" operator : , ended by the empty list []
- All elements must be of the **same type**
- Includes all basic functions
  - take, length, reverse, ++, head, tail
- In addition, you can index by !!

Special commands

Commands to the interpreter start with ":"

- `:?` for help
- `:load <module>`
- `:reload`
- `:quit`

Can be abbreviated to the first letter

Haskell scripts

New functions are defined within a script

- Text file comprising a sequence of definitions
- Usually have a `.hs` suffix
- Can by loaded by

```
$ ghci <filename>
> :load <filename>
```

Defining functions

```
fact1 1 = 1
fact1 n = n * fact1 (n-1)
```

```
fact2 n = product [1..n]
```

```
power n 0 = 1
power n k = n * power n (k-1)
```

Comments

```
-- Comment until the end of the line
```

```
{-
    A long comment
    over multiple
    lines.
-}
```

Naming requirements and conventions

Function and **argument** names must begin with a lower-case letter. For example:

```
myFun    fun1    arg_2    x'
```

By convention, **list arguments** usually have an `s` suffix on their name. For example:

```
xs    ns    nss
```

Names with only special symbols are **infix operators**:

```
++++    +/+    %-
```

Infix operators

Can be defined in prefix notations:

```
x +/+ y = 2*x + y
```

A prefix function turns infix by ` ` and infix turns prefix by ( )

`mod`, `elem`, (+), (+/+)

Precedence/asociativity of infix operators set by

```
infixr <0-9> <name>
infixl <0-9> <name>
infix <0-9> <name>
```

Information about associativity, precedence, and much else

```
> :info
```

Interesting infix operators

```
.   $   unary -
```

Pattern matching

The first LHS that matches the function call is executed

```
not False = True
not True  = False
```

not maps False to True, and True to False

Pattern matching

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

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

The underscore symbol _ is a **wildcard pattern** that matches any argument value.

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

A more efficient definition does not evaluate the second argument:

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

Pattern matching

The order of the definitions matters

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

Patterns may **not repeat variables**, due to efficiency. The following gives an error:

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

List patterns

Functions on lists can be defined using `x:xs` patterns

```
head (x:_) = x
tail (_:xs) = xs
```

We will see later it works similarly for other composite data types. `x:xs` patterns only match **non-empty** lists:

```
> head []
*** Exception: empty list
```

`x:xs` patterns must be **parenthesised**, because application has priority over `(:)`. The following definition gives an error:

```
head x:_ = x
```

Tuples

Tuples are **fixed length** sequences of elements of **arbitrary types**

```
(1,2)
('a','b')
(1,2,'c',False)
```

Their element can be accessed by pattern matching

```
first  (x,_,_) = x
second (_,x,_) = y
third  (_,_,x) = x
```

Pattern matching can be nested

```
f (1, (x:xs), 'a', (2,y)) = x:y:xs
```

As pattern

Sometimes it is useful to repeat larger parts of patterns form RHS on the LHS

```
copyfirst x:xs = x:x:xs
```

A part of the pattern can be assigned a name

```
copyfirst s@(x:xs) = x:s
```

Let / where

```
dist1 (x1,y1) (x2,y2) =
        let d1 = x1-x2
            d2 = y1-y2
        in sqrt(d1^2+d2^2)
```

```
dist2 (x1,y1) (x2,y2) = sqrt(d1^2+d2^2)
  where d1 = x1-x2
        d2 = y1-y2
```

The layout rule

The layout rule avoids the need for explicit syntax to indicate the grouping of definitions.

```
a = b + c where
              b = 1
              c = 2
```

means

```
a = b + c where {b=1; c=2}
```

The layout rule

Keywords (such as `where`, `let`, etc.) start a block:

- The first word after the keyword defines the **pivot column**.
- Lines exactly on the pivot define a new entry in the block.
- Start a line after the pivot to continue the previous lines.
- Start a line before the pivot to end the block.

Conditional expressions

```
abs n = if n >= 0 then n else -n
```

Conditional expressions can be nested:

```
signum n = if n < 0 then -1 else
               if n == 0 then 0 else 1
```

The must **always** have an else branch.

Guarded equations

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

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

Definitions with multiple conditions are then easier to read:

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

`otherwise` is defined in the prelude by `otherwise = True`

Set comprehensions

In mathematics, the **comprehension** notation can be used to construct new sets from old sets.

$$\boxed{\{x^2 \mid x \in \{1 \ldots 5\}\}}$$

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

List comprehensions

In Haskell, a similar comprehension notation can be used to construct new **lists** from old lists.

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

`x <- [1..5]` is called a **generator**.

Comprehensions can have **multiple** generators

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

Generator order

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.

Dependent generator

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

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

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

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

Infinite generator

Generators can be infinite (almost everything is lazy)

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

The order then matters even more

```
[x^y | x <- [1..], y <- [1,2]]
```

vs.

```
[x^y | y <- [1,2], x <- [1..]]
```

Guards

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

```
[x | x <- [1..10], even x]
```

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

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

Example: primes

A prime's only factors are 1 and itself

```
prime n = factors n == [1,n]
```

List of all primes

```
[x | x <- [2..], prime x]
```

Example: quicksort

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

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

Summary

Haskell is the unified standard for FP

- purely functional, lazy, statically typed

It has rich 2D syntax to write compactly

Functions are defined by pattern matching