Functional Programming
Lecture 7

Rostislav Horčík
Niklas Heim

Czech Technical University in Prague
Faculty of Electrical Engineering
xhorcik@fel.cvut.cz
heimnikl@fel.cvut.cz

Haskell

Main properties

- Purely functional programming language
    - necessary exceptions (IO) wrapped as monads
- Statically typed
    - types are derived and checked at compile time
    - types are automatically inferred
    - have a crucial role in controlling flow of the program
- Lazy
    - function arguments evaluated only when needed
    - strict evaluation has to be forced syntactically

Haskell implementation

Glasgow Haskell Compiler (GHC)

- the leading implementation of Haskell
- comprises a compiler and interpreter
- written in Haskell, runtime system in C and C--
- compatible with the latest standard Haskell 2010
- further provides a lot of extensions
- is freely available from:
  `https://www.haskell.org/ghcup/`

Starting GHCi

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

```
$ ghci

GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help

Prelude>
```

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

`Prelude` is a standard module imported by default.

Special commands

Commands to the interpreter start with `:`

- `:?` for help
- `:load <filename>`
- `:reload`
- `:type <expr>` displays the type of `expr`
- `:info <name>` displays info on a function or type
- `:quit`

Can be abbreviated to the first letter, e.g. `:r`

Haskell scripts

At the top level a Haskell program is a set of modules.

Each module consists of type and function declarations.

A module is defined within a script

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

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

Expressions

Every well-formed expression `e` has a well-formed type `t`, written `e :: t`.

Given `e` for evaluation, GHCi follows the following steps:

1. checks that `e` is syntactically correct.
2. infers a type for `e`, or checks that the type supplied by the programmer is correct.
3. evaluates `e` by reducing it to its simplest possible form to produce a value.
4. Provided the value is printable, GHCi then prints it at the terminal.

Basic syntax

Comments

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

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

Expressions

Expressions are built from

- literals representing constants of basic data types, e.g. `3.14`
- variables
- functions (function calls use prefix notation), e.g. `cos 3.14`
- operators (binary functions using infix notation), e.g. `3+5*8`

Infix notation brings precedence and left/right associativity stuff.

Basic types

Haskell has a number of basic types, including:

Bool	logical values `True, False`
Char	single characters `'a'`
String	strings of characters `"abc"`
Int	fixed-precision integers
Integer	arbitrary-precision integers
Float	single-precision floating-point numbers
Double	double-precision floating-point numbers

Function declarations

Function names must start with lower-case letter, e.g. `myFun`, `fun1`, `g_2`, `h'`

We may declare a function type, e.g.,

```
factorial :: Integer -> Integer
```

A function is defined by means of equations, e.g.,

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

```
power :: Integer -> (Integer -> Integer)
power _ 0 = 1
power n k = n * power n (k-1)
```

Operators

Names of operators consist only of special symbols, e.g. `+/+`

Can be defined in infix notation:

```
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
```

Pattern matching

The first LHS that matches the function call is evaluated

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

More efficient definition:

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

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

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

Let/where

```haskell
discr :: Float -> Float -> Float -> Float
discr a b c =
    let x = b*b
        y = 4*a*c
    in x - y
```

Alternatively

```haskell
discr a b c = x - y
    where x = b*b
          y = 4*a*c
```

`where` cannot be used inside guarded equations unlike `let`

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}
```

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.

Conditionals

```
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
```

There must always be an else branch.

Type of then-clause and else-clause must be the same.

```
(if True then 1 else "0")
```

throws a type error.

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`

Lists

Lists are sequences of elements of the **same type**, e.g. `[Int]`

```
[1,2,3,4,5]
[1..10]
['a'..'z']
[1,3..]
[10,9..1]
```

- Built by "cons" operator `:`, ended by the empty list `[]`
- Includes all basic functions
  `take, length, reverse, ++, head, tail`
- In addition, you can index by `!!`
- Data type `String` is just `[Char]`

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` pattern mathes only 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
```

A part of the pattern can be assigned a name

```
copyfirst s@(x:xs) = x:s -- same as x:x:xs
```

Tuples

Tuples are fixed-size sequences of elements of arbitrary types, e.g. `(Int, Char)`

```
(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 :: (Int, [Char], (Int, Char)) -> [Char]
f (1, (x:xs), (2,y)) = x:y:xs
```

List comprehensions

In Haskell, there is a list comprehension notation to construct new lists from existing lists.

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

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

Comprehensions can have multiple generators behaving like nested loops

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

Generators can be infinite (almost everything is lazy)

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

Dependent generator

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

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

Using a dependent generator, we can define a function that concatenates a list of lists:

```
flatten :: [[Int]] -> [Int]
flatten xss = [x | xs <- xss, x <- xs]
```

```
> flatten [[1,2],[3,4],[5]]
[1,2,3,4,5]
```

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

A prime's only factors are 1 and itself

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

List of all primes

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

What have we learned?

- Haskell is a statically typed pure functional programming language.
- It has a rich 2D syntax (layout rule).
- It has an automatic type inference mechanism.
- Every expression has a type.
- Lists store elements of the same type.
- Tuples have a fixed length but elements could be of different types.
- List comprehension allows to define new list from another lists.