AC21007: Haskell Lecture 3
Non-strict semantics, tuples, higher order functions

František Farka

Recapitulation

- Data type List ([], (:))

Recapitulation

- Data type List (`[]`, `(:)`)
- Function definition:

Recapitulation

- Data type List (`[]`, `(:)`)
- Function definition:
  - a set of equations:
    `<identifier> <pat`$_1$`> ... <pat`$_n$`> = <expr>`

Recapitulation

- Data type List ([], (:))
- Function definition:
  - a set of equations:
    <identifier> <pat$_1$> ... <pat$_n$> = <expr>
  - patterns:
    - a value (True, False, 0, ...)
    - a variable (x, xs, myVariable, ...)
    - _ – wildcard, "don't care" pattern
    - list constructors, i.e.: [], (<pat$_{head}$> : <pat$_{tail}$>)

Recapitulation

- Data type List (`[]`, `(:)`)
- Function definition:
  - a set of equations:
    <identifier> <pat$_1$> ... <pat$_n$> = <expr>
  - patterns:
    - a value (`True`, `False`, `0`, `...`)
    - a variable (`x`, `xs`, `myVariable`, `...`)
    - _ – wildcard, "don't care" pattern
    - list constructors, i.e.: `[]`, (<pat$_{head}$> : <pat$_{tail}$>)

Demo ...

Non-strict (lazy) semantics

- In Haskell, expressions are evaluated lazily – not evaluated until needed

Non-strict (lazy) semantics

- In Haskell, expressions are evaluated lazily – not evaluated until needed
- Consider a variant of our `power` function:

```
power' :: Int -> Int -> Float -> Int
power' b 0 _ = 1
power' b n x = b * (power b (n - 1) x)
```

Non-strict (lazy) semantics

- In Haskell, expressions are evaluated lazily – not evaluated until needed
- Consider a variant of our `power` function:

```
power’ :: Int -> Int -> Float -> Int
power’ b 0 _ = 1
power’ b n x = b * (power b (n - 1) x)
```

- Consider the following function call:

```
power’ 7 2 (1.0 / 0)
```

Non-strict (lazy) semantics

- In Haskell, expressions are evaluated lazily – not evaluated until needed
- Consider a variant of our power function:

```
power' :: Int -> Int -> Float -> Int
power' b 0 _ = 1
power' b n x = b * (power b (n - 1) x)
```

- Consider the following function call:

```
power' 7 2 (1.0 / 0)
==> 7 * (power' 7 (2 - 1) (1.0 / 0))
```

Non-strict (lazy) semantics

- In Haskell, expressions are evaluated lazily – not evaluated until needed
- Consider a variant of our power function:

```
power' :: Int -> Int -> Float -> Int
power' b 0 _ = 1
power' b n x = b * (power b (n - 1) x)
```

- Consider the following function call:

```
power' 7 2 (1.0 / 0)
==> 7 * (power' 7 (2 - 1) (1.0 / 0))
==> 7 * (power' 7 1) (1.0 / 0)
```

Non-strict (lazy) semantics

- In Haskell, expressions are evaluated lazily – not evaluated until needed
- Consider a variant of our power function:

```
power' :: Int -> Int -> Float -> Int
power' b 0 _ = 1
power' b n x = b * (power b (n - 1) x)
```

- Consider the following function call:

```
power' 7 2 (1.0 / 0)
==> 7 * (power' 7 (2 - 1) (1.0 / 0))
==> 7 * (power' 7 1) (1.0 / 0)
==> 7 * (7 * (power' 7 (1 - 1) (1.0 / 0)))
```

Non-strict (lazy) semantics

- In Haskell, expressions are evaluated lazily – not evaluated until needed
- Consider a variant of our power function:

```
power' :: Int -> Int -> Float -> Int
power' b 0 _ = 1
power' b n x = b * (power b (n - 1) x)
```

- Consider the following function call:

```
power' 7 2 (1.0 / 0)
==> 7 * (power' 7 (2 - 1) (1.0 / 0))
==> 7 * (power' 7 1) (1.0 / 0)
==> 7 * (7 * (power' 7 (1 - 1) (1.0 / 0)))
==> 7 * (7 * (power' 7 0 (1.0 / 0)))
```

Non-strict (lazy) semantics

- In Haskell, expressions are evaluated lazily – not evaluated until needed
- Consider a variant of our power function:

```
power' :: Int -> Int -> Float -> Int
power' b 0 _ = 1
power' b n x = b * (power b (n - 1) x)
```

- Consider the following function call:

```
power' 7 2 (1.0 / 0)
==> 7 * (power' 7 (2 - 1) (1.0 / 0))
==> 7 * (power' 7 1) (1.0 / 0)
==> 7 * (7 * (power' 7 (1 - 1) (1.0 / 0)))
==> 7 * (7 * (power' 7 0 (1.0 / 0)))
==> 7 * (7 * (1))
...
==> 49
```

Non-strict (lazy) semantics - infinite lists

- Consider the following function:

```
repeat :: a -> [a]
repeat x    =   x : (repeat x)
```

Non-strict (lazy) semantics - infinite lists

- Consider the following function:

```
repeat :: a -> [a]
repeat x    =   x : (repeat x)
```

  this function defines an infinite list of elements, e.g:

```
repeat 1    ==> [1, 1, 1, 1, 1, 1, ... ]
```

Non-strict (lazy) semantics - infinite lists (cont.)

- A more useful example – powers of an integer:

```
powersof :: Integer -> [Integer]
powersof b  = pow b 1
    where
        pow b p = b : pow b (b * p)
```

Non-strict (lazy) semantics - infinite lists (cont.)

- A more useful example – powers of an integer:

```
powersof :: Integer -> [Integer]
powersof b  = pow b 1
    where
         pow b p = b : pow b (b * p)
```

- Note:
  - `Int` is machine integer (32/64 bits), `Integer` is arbitrary precision integer
  - `where` block allows for local-scope definitions

Non-strict (lazy) semantics - infinite lists (cont.)

- A more useful example – powers of an integer:

```
powersof :: Integer -> [Integer]
powersof b  = pow b 1
    where
        pow b p = b : pow b (b * p)
```

  this function defines an infinite list, e.g.:

  `powersof 2 ==> [1, 2, 4, 8, 16, 32, ... ]`

- Note:
  - `Int` is machine integer (32/64 bits), `Integer` is arbitrary precision integer
  - `where` block allows for local-scope definitions

Non-strict (lazy) semantics - infinite lists (cont.)

- A more useful example – powers of an integer:

```
powersof :: Integer -> [Integer]
powersof b  = pow b 1
     where
          pow b p = b : pow b (b * p)
```

  this function defines an infinite list, e.g.:

  `powersof 2 ==> [1, 2, 4, 8, 16, 32, ... ]`

- Our power function:

```
power :: Integer -> Integer -> Integer
power b n = (powersof b) !! n
```

- Note:
  - `Int` is machine integer (32/64 bits), `Integer` is arbitrary precision integer
  - `where` block allows for local-scope definitions

Tuple Datatype – `(a, b)`

- Data type `(a, b)` – type of pairs of values, polymorphic in both of its components `a` and `b`

Tuple Datatype – (a, b)

- Data type (a, b) – type of pairs of values, polymorphic in both of its components a and b
- One constructor (a, b) :: a -> b -> (a, b)

Tuple Datatype – (a, b)

- Data type (a, b) – type of pairs of values, polymorphic in both of its components a and b
- One constructor (a, b) :: a -> b -> (a, b)
- E.g. (True, "hello") :: (Bool, String)

Tuple Datatype – `(a, b)`

- Data type `(a, b)` – type of pairs of values, polymorphic in both of its components `a` and `b`
- One constructor `(a, b) :: a -> b -> (a, b)`
- E.g. `(True, "hello") :: (Bool, String)`
- Functions (projections) `fst` and `snd`:

```
fst :: (a, b) -> a
fst (x, _) = x

snd :: (a, b) -> b
snd (_, y) = y
```

Tuple Datatype – `(a, b)`

- Data type `(a, b)` – type of pairs of values, polymorphic in both of its components `a` and `b`
- One constructor `(a, b) :: a -> b -> (a, b)`
- E.g. `(True, "hello") :: (Bool, String)`
- Functions (projections) `fst` and `snd`:

```
fst :: (a, b) -> a
fst (x, _) = x

snd :: (a, b) -> b
snd (_, y) = y
```

- Note: tuple constructor may be used as a pattern

Tuple Datatype – (a, b)

- Data type (a, b) – type of pairs of values, polymorphic in both of its components a and b
- One constructor (a, b) :: a -> b -> (a, b)
- E.g. (True, "hello") :: (Bool, String)
- Functions (projections) fst and snd:

```
fst :: (a, b) -> a
fst (x, _) = x

snd :: (a, b) -> b
snd (_, y) = y
```

- Note: tuple constructor may be used as a pattern
- There are also triples (a, b, c), quadruples (a, b, c, d), etc. (no genertic fst and snd though)

Combining lists and tuples – zip

- `zip` takes two lists and returns a list of corresponding pairs
- If one input list is short, excess elements of the longer list are discarded

Combining lists and tuples – zip

- zip takes two lists and returns a list of corresponding pairs
- If one input list is short, excess elements of the longer list are discarded

```
zip :: [a] -> [b] -> [(a,b)]
zip []     _      = []
zip _      []     = []
zip (a:as) (b:bs) = (a,b) : zip as bs
```

Syntactic intermezzo: `if then else`

- Haskell has a conditional **expression**:

  if <cnd> **then** <x> **else** <y>

Syntactic intermezzo: if then else

- Haskell has a conditional **expression**:

  if <cnd :: Bool> **then** <x > **else** <y >

- <cnd> is an expresion that evaluates to Bool

Syntactic intermezzo: `if then else`

- Haskell has a conditional **expression**:

  if `<cnd :: Bool>` **then** `<x :: a>` **else** `<y :: a>`

- `<cnd>` is an expresion that evaluates to `Bool`
- Both branches are expressions that evaluates to a value of a type `a`

Syntactic intermezzo: `if then else`

- Haskell has a conditional **expression**:

  if `<cnd :: Bool>` **then** `<x :: a>` **else** `<y :: a>` `:: a`

- `<cnd>` is an expresion that evaluates to `Bool`
- Both branches are expressions that evaluates to a value of a type `a`
- The whole expression evaluates to the appropriate value of a type `a`

Syntactic intermezzo: `if then else`

- Haskell has a conditional **expression**:

  if `<cnd :: Bool>` **then** `<x :: a>` **else** `<y :: a>` `:: a`

- `<cnd>` is an expresion that evaluates to `Bool`
- Both branches are expressions that evaluates to a value of a type `a`
- The whole expression evaluates to the appropriate value of a type `a`
- `then` and `else` branches may be indented by whitespace

Syntactic intermezzo: `if then else`

- Haskell has a conditional **expression**:

  if `<cnd :: Bool>` **then** `<x :: a>` **else** `<y :: a>` `:: a`

- `<cnd>` is an expresion that evaluates to `Bool`
- Both branches are expressions that evaluates to a value of a type `a`
- The whole expression evaluates to the appropriate value of a type `a`
- `then` and `else` branches may be indented by whitespace
- E.g.:

```
max :: Int -> Int -> Int
max x y = if x > y  then x
                    else y
```

Anonymous (lambda) functions

```
2 + 3 ::  Int
2 + x ::           Int
```

Anonymous (lambda) functions

```
   2 + 3 ::  Int
   2 + x ::            Int
Not in scope:  ‘x‘
```

Anonymous (lambda) functions

- Functions without a name
- Syntax:

\<var$_1$> ... <var$_n$> -> <expr>

```
      2 + 3 ::  Int
\x -> 2 + x ::  Int -> Int
```

Anonymous (lambda) functions

- Functions without a name
- Syntax:

```
\<var_1> ... <var_n> -> <expr>
```

- Variables var_1 to var_n in scope in the expression *expr*

```
      2 + 3 ::  Int
\x -> 2 + x ::  Int -> Int
```

Anonymous (lambda) functions

- Functions without a name
- Syntax:

```
\<var_1> ... <var_n> -> <expr>
```

- Variables var_1 to var_n in scope in the expression *expr*
- Anonymous functions:
  - can be applied to an argument:
    `(\x -> 2 + x) 3 ==> 5`

```
      2 + 3 ::  Int
\x -> 2 + x ::  Int -> Int
```

Anonymous (lambda) functions

- Functions without a name
- Syntax:

  \<var$_1$> ... <var$_n$> -> <expr>

- Variables var_1 to var_n in scope in the expression *expr*
- Anonymous functions:
  - can be applied to an argument:
    `(\x -> 2 + x) 3 ==> 5`
  - can be passed as an argument ... functions **are** values

```
      2 + 3 ::  Int
\x -> 2 + x ::  Int -> Int
```

Anonymous (lambda) functions

- Functions without a name
- Syntax:

```
\<var1> ... <varn> -> <expr>
```

- Variables *var*$_1$ to *var*$_n$ in scope in the expression *expr*
- Anonymous functions:
  - can be applied to an argument:
    `(\x -> 2 + x) 3 ==> 5`
  - can be passed as an argument . . . functions **are** values
- E.g.:

```
     2 + 3 ::  Int
\x -> 2 + x ::  Int -> Int
```

Anonymous (lambda) functions (cont.)

- `filter`, applied to a predicate and a list, returns the list of those elements that satisfy the predicate

Anonymous (lambda) functions (cont.)

- `filter`, applied to a predicate and a list, returns the list of those elements that satisfy the predicate

```
filter :: (a -> Bool) -> [a] -> [a]
filter _    []     = []
filter pred (x:xs) = if (pred x)
    then x : filter pred xs
    else filter pred xs
```

Anonymous (lambda) functions (cont.)

- `filter`, applied to a predicate and a list, returns the list of those elements that satisfy the predicate

```
filter :: (a -> Bool) -> [a] -> [a]
filter _     []     = []
filter pred (x:xs) = if (pred x)
    then x : filter pred xs
    else filter pred xs
```

- E.g:

```
filter (\x -> x `mod` 2 == 1) [1, 2, 3, 4, 5, 6]
    ==> [1, 3, 5, 7]

filter (\x -> x `mod` 2 == 0) [1, 2, 3, 4, 5, 6]
    ==> [2, 4, 6]
```

First-class functions

- All functions can be passed as arguments, e.g standard functions `even` and `odd`:

```
filter even [1, 2, 3, 4, 5, 6]
     ==> [1, 3, 5, 7]

filter even [1, 2, 3, 4, 5, 6]
     ==> [2, 4, 6]
```

First-class functions (cont)

- Function type `a -> b` (right-associative)

First-class functions (cont)

- Function type `a -> b` (right-associative)
- Values of this type are constructed by

First-class functions (cont)

- Function type `a -> b` (right-associative)
- Values of this type are constructed by
  - usual function definitions
  - lambda constructions

First-class functions (cont)

- Function type `a -> b` (right-associative)
- Values of this type are constructed by
  - usual function definitions
  - lambda constructions

```
max ::   Int ->  Int -> Int
max x y = if x > y then x else y
```

First-class functions (cont)

- Function type `a -> b` (right-associative)
- Values of this type are constructed by
  - usual function definitions
  - lambda constructions

```
   max ::   Int -> (Int -> Int)
-- max x y = if x > y then x else y
   max x =
```

First-class functions (cont)

- Function type `a -> b` (right-associative)
- Values of this type are constructed by
  - usual function definitions
  - lambda constructions

```
   max ::   Int -> (Int -> Int)
-- max x y = if x > y then x else y
   max x = \y -> if x > y then x else y
```

First-class functions (cont)

- Function type `a -> b` (right-associative)
- Values of this type are constructed by
  - usual function definitions
  - lambda constructions

```
   max ::  (Int -> (Int -> Int))
-- max x y = if x > y then x else y
-- max x = \y -> if x > y then x else y
   max =
```

First-class functions (cont)

- Function type `a -> b` (right-associative)
- Values of this type are constructed by
  - usual function definitions
  - lambda constructions

```
   max ::  (Int -> (Int -> Int))
-- max x y = if x > y then x else y
-- max x = \y -> if x > y then x else y
   max = \x y -> if x > y then x else y
```

First-class functions (cont)

- Function type `a -> b` (right-associative)
- Values of this type are constructed by
  - usual function definitions
  - lambda constructions
- The following definitions of `max` are equivalent:

```
   max ::  (Int -> (Int -> Int))
-- max x y = if x > y then x else y
-- max x = \y -> if x > y then x else y
   max = \x y -> if x > y then x else y
```

First-class functions (cont)

- Function type `a -> b` (right-associative)
- Values of this type are constructed by
  - usual function definitions
  - lambda constructions
- The following definitions of `max` are equivalent:

```
   max ::  (Int -> (Int -> Int))
-- max x y = if x > y then x else y
-- max x = \y -> if x > y then x else y
   max = \x y -> if x > y then x else y
```

- Haskell compiler will figure out types from LHS patterns and type of RHS expression

First-class functions (cont)

- Function type `a -> b` (right-associative)
- Values of this type are constructed by
  - usual function definitions
  - lambda constructions
- The following definitions of `max` are equivalent:

```
   max ::  (Int -> (Int -> Int))
-- max x y = if x > y then x else y
-- max x = \y -> if x > y then x else y
   max = \x y -> if x > y then x else y
```

- Haskell compiler will figure out types from LHS patterns and type of RHS expression
- Note: In a function definition all equations must have the same number of LHS patterns

Next time

- Monday the the 1st of February, 2-3PM, Dalhousie 3G05 LT2
- More (higher-order) list functions (`map`, ...)
- Recursion, folds over lists