AC21007: Haskell Lecture 3
Non-strict semantics, tuples

František Farka

Recapitulation

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

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 = p : 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`
- 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 genetic `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

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

- `<cnd>` is an expression 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 white-space

Syntactic intermezzo: `if then else` (cont.)

- `if` is an expression – it can be used as such, e. g.:

```
ghci> (if ("a" == "b") then 3 else 5) + 2
    ==> 7
```

- In function definition (note the indentation):

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

Next time

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