Programming in Haskell
Aug-Nov 2015

LECTURE 5

AUGUST 18, 2015

S P Suresh
Chennai Mathematical Institute

The datatype Char

- Values are written with single quotes
  - `’a’, ’3’, ’%’, ’#’, …`
- Character symbols stored in a table (e.g. ASCII)
  - Functions `ord` and `chr` connect characters and table
  - Inverses: `c == chr (ord c)`, `j == ord (chr j)`
  - Note: import `Data.Char` to use `ord` and `char`

Example: capitalize

- Function to convert lower case to upper case
- Brute force, enumerate all cases

```
capitalize :: Char -> Char
capitalize ’a’ = ’A’
capitalize ’b’ = ’B’
…
capitalize ’z’ = ’Z’
capitalize ch  = ch
```

Example: capitalize ...

- Can assume that `'a'`,...,`'z'` and `'A'`,...,`'Z'` and `'0'`,...,`'9'` each have consecutive `ord` values
- A more intelligent solution

```
capitalize :: Char -> Char
capitalize ch
  | ('a' <= ch && ch <= 'z') =
         chr (ord ch + (ord 'A' - ord 'a'))
  | otherwise = ch
```

Strings

- A string is a sequence of characters
- In Haskell, `String` is a synonym for `[Char]`
  - `['h','e','l','l','o'] == "hello"`
  - `"" == []`
- Usual list functions can be used on `String`
  - `length`, `reverse`, …

Example: occurs

- Search for a character in a string
- `occurs c s` returns `True` if `c` is found in `s`

```
occurs :: Char -> String -> Bool
occurs c ”” = False
occurs c (x:xs)
  | c == x    = True
  | otherwise = occurs c xs
```

Example: touppercase

- Convert an entire string to uppercase
- Apply `capitalize` to each character

```
touppercase :: String -> String
touppercase ""     = ""
touppercase (c:cs) = (capitalize c):
                         (touppercase cs)
```

- Apply `f` to each element in a list—will come back to this later

Example: position

- `position c s` : first position in `s` where `c` occurs
- Return `length s` if no occurrence of `c` in `s`
  - `position 'a' "battle axe"` ⇛ `1`
  - `position 'd' "battle axe"` ⇛ `10`

```
position :: Char -> String -> Int
position c "" = 0
position c (d:ds)
   | c == d    = 0
   | otherwise = 1 + (position c ds)
```

Example: Counting words

- `wordc` : count the number of words in a string
- Words separated by white space: `' '`, `'\t'`, `'\n'`

```
whitespace :: Char -> Bool
whitespace ' '  = True
whitespace '\t' = True
whitespace '\n' = True
whitespace _    = False
```

Example: Counting words

- Count white space in a string?

```
wscount :: String -> Int
wscount ”” = 0
wscount (c:cs)
  | whitespace c = 1 + wscount cs
  | otherwise    = wscount cs
```

- Not enough!
  - Consider `”abc        d”`

Example: Counting words

- Keep track of whether we are inside a word or outside a word
  - Outside a word: ignore whitespace, but non-whitespace starts a new word
  - Inside a word: ignore non-whitespace, but whitespace ends current word
  - Count the number of times a new word starts

Example: Counting words

- New word starts whenever a non-whitespace follows a whitespace—insert initial space to catch first character

```
wordcaux :: String -> Int
wordcaux [c] = 0
wordcaux (c:d:ds)
  | (whitespace c) && not (whitespace d) =
          1 + wordcaux (d:ds)
  | otherwise = wordcaux (d:ds)

wordc :: String -> Int
wordc s = wordcaux (’ ’:s)
```

Tuples

- Keep multiple types of data together
  - Student info: Name, ID, Date of birth

    ("Abhirup", 2106, "Jan 1, 2000")

  - List of marks in a course

    [("Sasha",95),
    ("Becky",95),
    ("Charlotte",98)]

Tuples ..

- Tuple type `(T1,T2,…,Tn)` groups together multiple types
  - `(3,-21) :: (Int,Int)`
  - `(13, True, 97) :: (Int, Bool, Int)`
  - `([1,2],73) :: ([Int],Int)`

Pattern matching

* Use tuple structure for pattern matching

* Sum pairs of integers

```
sumpairs :: (Int,Int) -> Int
sumpairs (x,y) = x+y
```

* Sum pairs of integers in a list of pairs

```
sumpairlist :: [(Int,Int)] -> Int
sumpairlist [] = 0
sumpairlist (x,y):zs = x + y + sumpairlist zs
```

Example: Marks list

- List of pairs (Name,Marks) — `[(String,Int)]`
- Given a name, find the marks

```
lookup :: String -> [(String,Int)] -> Int
lookup p [] = -1
lookup p ((name,marks):ms)
  | (p == name) = marks
  | otherwise   = lookup p ms
```

Type aliases

- Tedious to keep writing `[(String,Int)]`
- Introduce a new name for this type

```
type Marklist = [(String,Int)]
```

- Then

```
lookup :: String -> Marklist -> Int
```

Type aliases …

- A type definition only creates an alias for a type
  - Both `Marklist` and `[(String,Int)]` are the same type
  - `String` is a type alias for `[Char]`

Example: Point

- ```
  type Point2D = (Float,Float)

  distance :: Point2D -> Point2D -> Float
  distance (x1,y1) (x2,y2) =
       sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1))
  ```
- ```
  type Point3D = (Float,Float,Float)

  distance :: Point3D -> Point3D -> Float
  distance (x1,y1,z1) (x2,y2,z2) =
       sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) +
            (z2-z1)*(z2-z1))
  ```

Type aliases are same type

- Suppose
  - `f :: Float -> Float -> Point2D`
  - `g :: (Float,Float) -> (Float,Float) ->Float`
- Then
  - `g (f x1 x2) (f y1 y2)` is well typed
  - `f` produces `Point2D` that is same as `(Float,Float)`

Local definitions

- Let us return to distance

- ```
  type Point2D = (Float,Float)

  distance :: Point2D -> Point2D -> Float
  distance (x1,y1) (x2,y2) =
    sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1))
  ```

- Introduce `sqr` to simplify expressions
  ```

Local definitions

- ```
  sqr :: Float -> Float
  sqr x = x*x

  type Point2D = (Float,Float)

  distance :: Point2D -> Point2D -> Float
  distance (x1,y1) (x2,y2) =
    sqrt(sqr (x2-x1) + sqr (y2-y1))
  ```
- But now, auxiliary function `sqr` is globally available
  ```

Local definitions

```
type Point2D = (Float,Float)

distance :: Point2D -> Point2D -> Float
distance (x1,y1) (x2,y2) =
  sqrt(sqr (x2-x1) + sqr (y2-y1))
  where
  sqr :: Float -> Float
  sqr x = x*x
```

- Definition of `sqr` is now local to `distance`

Local definitions

- Another motivation

```
type Point2D = (Float,Float)

distance :: Point2D -> Point2D -> Float
distance (x1,y1) (x2,y2) =
  sqrt(xdiff*xdiff + ydiff*ydiff)
  where
  xdiff :: Float
  xdiff = x2 - x1
  ydiff :: Float
  ydiff = y2 - y1
```

Local definition

- `xdiff*xdiff` vs `(x2-x1)*(x2-x1)`
- With `xdiff*xdiff`, (x2-x1) is only computed once
- In general, ensure that common subexpressions are evaluated only once

Summary

- Tuples allow different types to come together in a single unit
    - Pattern matching to extract individual components
- `type` statement creates type aliases
- `where` allows local definitions