CS 457/557: Functional Languages

Lecture 2: First Examples

Mark P Jones
Portland State University
Expressions Have Types

The *type* of an expression tells you what kind of value you might expect to see if you evaluate that expression.

In Haskell, read “::” as “has type”

Examples:

- 1 :: Int, 'a' :: Char, True :: Bool, 1.2 :: Float, ...

You can ask ghci for the type of an expression: :t expr
Pairs

- A pair packages two values into one
  - (1, 2)    ('a', 'z')    (True, False)

- Components can have different types
  - (1, 'z')    ('a', False)    (True, 2)

- The type of a pair whose first component is of type A and second component is of type B is written \((A,B)\)

- What are the types of the pairs above?
Operating on Pairs

There are built-in functions for extracting the first and second component of a pair:

- \( \text{fst} \) (True, 2) = True
- \( \text{snd} \) (0, 7) = 7

Is the following property true?
For any pair \( p \), \( (\text{fst} \ p, \text{snd} \ p) = p \)
Lists

Lists can be used to store zero or more elements, in sequence, in a single value:

[]     [1, 2, 3]     ['a', 'z']     [True, True, False]

All of the elements in a list must have the same type.

The type of a list whose elements are of type \(A\) is written as \([A]\).

What are the types of the lists above?
Operating on Lists

- There are built-in functions for extracting the head and the tail components of a list:
  - head [1,2,3,4] = 1
  - tail [1,2,3,4] = [2,3,4]

- Conversely, we can build a list from a given head and tail using the “cons” operator:
  - 1 : [2, 3, 4] = [1, 2, 3, 4]

- Is the following property true?
  For any list xs, head xs : tail xs = xs
More Operations on Lists

- Finding the length of a list: 
  \[ \text{length} \{1,2,3,4,5\} = 5 \]

- Finding the sum of a list: 
  \[ \text{sum} \{1,2,3,4,5\} = 15 \]

- Finding the product of a list: 
  \[ \text{product} \{1,2,3,4,5\} = 120 \]

- Applying a function to the elements of a list: 
  \[ \text{map odd} \{1,2,3,4\} = \{\text{True}, \text{False}, \text{True}, \text{False}\} \]
Continued ...

Selecting an element (by position):
\[1,2,3,4,5\] !! 3 = 4

Taking an initial prefix (by number):
take 3 \([1,2,3,4,5]\) = \([1,2,3]\)

Taking an initial prefix (by property):
takeWhile odd \([1,2,3,4,5]\) = \([1]\)

Checking for an empty list:
null \([1,2,3,4,5]\) = False
More ways to Construct Lists

- **Concatenation:**
  \[ [1,2,3] ++ [4,5] = [1,2,3,4,5] \]

- **Arithmetic sequences:**
  \[ [1..10] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] \]
  \[ [1,3..10] = [1, 3, 5, 7, 9] \]

- **Comprehensions:**
  \[ [ 2 * x | x <- [1,2,3,4,5] ] = [2, 4, 6, 8, 10] \]
  \[ [ y | y <- [1,2,3,4], odd y ] = [ 1, 3 ] \]
Strings are Lists:

A String is just a list of Characters

['w', 'o', 'w', '!'] = "wow!"
['a'..'j'] = "abcdefghij"
"hello, world" !! 7 = 'w'
length "abcdef" = 6
"hello, " ++ "world" = "hello, world"
take 3 "functional" = "fun"
Functions

The type of a function that maps values of type A to values of type B is written A -> B

Examples:

- odd :: Int -> Bool
- fst :: (a, b) -> a (a,b are type variables)
- length :: [a] -> Int
Operations on Functions

- **Function application.** If \( f :: A \to B \) and \( x :: A \), then \( f \, x :: B \)

- Notice that function application associates more tightly than any infix operator:
  \[
  f \, x + y = (f \, x) + y
  \]

- **In types, arrows associate to the right:**
  \[
  A \to B \to C = A \to (B \to C)
  \]

Example: \( \text{take} :: \text{Int} \to [a] \to [a] \)
  \[
  \text{take 2} \, [1, 2, 3, 4] = (\text{take 2}) \, [1, 2, 3, 4]
  \]
Sections

If $\oplus$ is a binary op of type $A \to B \to C$, then we can use “sections”:

- $(\oplus) :: A \to B \to C$
- $(\text{expr } \oplus) :: B \to C$ (assuming expr::A)
- $(\oplus \text{ expr}) :: A \to C$ (assuming expr::B)

Examples:
- $(1+)$, $(2*)$, $(1/)$, $(<10)$, ...
Higher-order Functions

\[ \text{map} :: (a -> b) -> [a] -> [b] \]
- \[ \text{map} (1+) [1..5] = [2,3,4,5,6] \]

\[ \text{takeWhile} :: (a -> \text{Bool}) -> [a] -> [a] \]
- \[ \text{takeWhile} (<5) [1..10] = [1,2,3,4] \]

\[ (.) :: (a -> b) -> (c -> a) -> c -> b \]
- \[ (\text{odd . } (1+)) 2 = \text{True} \]

"composition"
Example: Calculating Fractals
Calculating Fractals

- Based on Mark Jones’ article “Composing Fractals” that was published as a “functional pearl” in the Journal of functional Programming
- Flexible programs for drawing Mandelbrot and Julia set fractals in different ways
- No claim to be the best/fastest fractal drawing program ever created!
- Illustrates key features of functional programming in an elegant and “calculational” style
- As it happens, no recursion!
Mandelbrot Sequences

type Point = (Float, Float)

next :: Point -> Point -> Point
next (u,v) (x,y) = (x*x-y*y+u, 2*x*y+v)

The source of all that beauty & complexity!

like complex numbers
p = u+iv  z = x+iy

mandelbrot :: Point -> [Point]
mandelbrot p  = iterate (next p) (0,0)

Apply function repeatedly, producing as many elements as we like ...
Converge or Diverge?

Fractals> `mandelbrot (0,0)`

[[0.0, 0.0], [0.0, 0.0], [0.0, 0.0], [0.0, 0.0], [0.0, 0.0], [0.0, 0.0],
[0.0, 0.0], Interrupted]

Fractals> `mandelbrot (0.1,0)`

[[0.0, 0.0], [0.1, 0.0], [0.11, 0.0], [0.1121, 0.0], [0.1125664, 0.0],
[0.1126712, 0.0], [0.1126948, 0.0], Interrupted]

Fractals> `mandelbrot (0.5,0)`

[[0.0, 0.0], [0.5, 0.0], [0.75, 0.0], [1.0625, 0.0], [1.628906, 0.0],
[3.153336, 0.0], [10.44353, 0.0], Interrupted]

Fractals> `mandelbrot (1,0)`

[[0.0, 0.0], [1.0, 0.0], [2.0, 0.0], [5.0, 0.0], [26.0, 0.0], [677.0, 0.0],
[458330.0, 0.0], Interrupted]

Fractals>
The Mandelbrot Set

- The Mandelbrot Set is the set of all points for which the corresponding Mandelbrot sequence converges

- How can we test for this?

- How can we visualize the results?
Testing for Membership

fairlyClose :: Point -> Bool
fairlyClose (u,v) = (u*u + v*v) < 100

An almost arbitrary constant

inMandelbrotSet :: Point -> Bool
inMandelbrotSet p = all fairlyClose (mandelbrot p)

This could take a long time ...
**Pragmatics**

- For points very close to the edge, it may take many steps to determine whether the sequence will converge or not.

- It is impossible to determine membership with complete accuracy because of rounding errors.

- And besides, the resulting diagram is really dull!

- If life gives you lemons ... make lemonade!
Approximating Membership

fracImage :: [color] -> Point -> color
fracImage palette = (palette{!!))
  . length
  . take n
  . takeWhile fairlyClose
  . mandelbrot
where n = length palette - 1

Now we’re using a palette of multiple colors instead of a monochrome membership!

But how are we going to render this?
\[ \delta_y = \frac{(y_{\text{max}} - y_{\text{min}})}{(r-1)} \]

\[ \delta_x = \frac{(x_{\text{max}} - x_{\text{min}})}{(c-1)} \]
Grids

type Grid a = [[a]]

grid :: Int -> Int -> Point -> Point -> Grid Point
grid c r (xmin,ymin) (xmax,ymax)
  = [[ (x, y) | x <- for c xmin xmax ]
      | y <- for r ymin ymax ]

for :: Int -> Float -> Float -> [Float]
for n min max = take n [min, min+delta ..]
  where delta = (max-min) / fromIntegral (n-1)
Some Sample Grids

mandGrid = grid 79 37 (-2.25, -1.5) (0.75, 1.5)

juliaGrid = grid 79 37 (-1.5, -1.5) (1.5, 1.5)

Names make it easier to refer to previously defined values!
Images

type Image color = Point -> color

sample :: Grid Point -> Image color -> Grid color
sample points image
    = map (map image) points

Allow for different types of “color”

Functions are just regular values ...
Putting it all together

draw :: [color] ->
  Grid Point ->
  (Grid color -> pic) -> pic
draw palette grid render
  = render (sample grid (fracImage palette))
Example 1

```
charPalette :: [Char]
charPalette  = "    ,.`"~:;o-!|<?/<>X+=\{^O#%&@8*$"

charRender :: Grid Char -> IO ()
charRender  = putStrLn . unlines

eexample1 = draw charPalette mandGrid charRender
```
Example 2

type PPMcolor = (Int, Int, Int)

ppmPalette :: [PPMcolor]
ppmPalette = [ ((2*i) `mod` (ppmMax+1)), i, ppmMax-i) |
| i <- [0..ppmMax] ]
ppmMax = 31 :: Int

ppmRender :: Grid PPMcolor -> [String]
ppmRender g = ["P3", show w ++ " " ++ show h, show ppmMax]
   ++ [ show r ++ " " ++ show g ++ " " ++ show b |
        | row <- g, (r,g,b) <- row ]
where w = length (head g)
       h = length g
draw ppmPalette mandGridHi ppmRender
An Imperative Approach

deltax = (xmax-xmin)/cols;
deltay = (ymax-ymin)/rows;
for (x=xmin; x<=xmax; x+=xdelta) {
    for (y=ymin; y<=ymax; y+=ydelta) {
        float px = 0, py = 0;
        for (i=1; i<colorsMax; i++) {
            (px, py) = (px*px-py*py+x, 2*px*py+y)
            if (px*px + py*py >= 100)
                break;
        }
        putchar(colors[i-1]);
    }
    putchar(’\n’);
}

An Imperative Approach

deltax = (xmax-xmin)/cols;
deltay = (ymax-ymin)/rows;
for (x=xmin; x<=xmax; x+=xdelta) {
    for (y=ymin; y<=ymax; y+=ydelta) {
        float px = 0, py = 0;
        for (i=1; i<colorsMax; i++) {
            newpx = px*px-py*py+x;
            newpy = 2*px*py+v;
            px    = newpx;
            py    = newpy;
            if (px*px + py*py >= 100)
                break;
        }
        putchar(colors[i-1]);
    }
    putchar(’\n’);
}
Down with Tangling!

- Changes to a program may require modifications of the source code in multiple places.
- The implementation of a program feature may be “tangled” through the code.
- Programs are easier to understand and maintain when important changes can be isolated to a single point in the code (and, perhaps, turned into a parameter).
- A simpler example:
  - Calculate the sum of the squares of the numbers from 1 to 10
  - `sum (map square [1..10])`
Summary

- An appealing, high-level approach to program construction in which independent aspects of program behavior are neatly separated.

- It is possible to program in a similar compositional / calculational manner in other languages ...

- ... but it seems particularly natural in a functional language like Haskell ...