CS 457/557: Functional Languages

Lecture 1: Introduction

Mark P Jones
Portland State University
What is Functional Programming?
What is Functional Programming?

- An alternative to dysfunctional programming?
- Programming with functions?
- Programming without side-effects?
What is Functional Programming?

- **Functional programming** is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands.

- Expressions are formed by using **functions** to combine **basic values**.

- A **functional language** is a language that supports and encourages programming in a functional style.
Functions:

In a pure functional language:

- The result of a function depends *only* on the values of its inputs:
  - Like functions in mathematics
  - No global variables / side-effects

- Functions are first-class values:
  - They can be stored in data structures
  - They can be passed as arguments or returned as results of other functions
Functional Languages:

- Pure, lazy evaluation, strong typing:
  - Haskell, Miranda, Orwell, ...

- Impure, strict evaluation, strong typing:
  - Standard ML (SML), Objective CAML (OCaml), F#, ...

- Impure, strict evaluation, dynamic typing:
  - Lisp, Scheme, Erlang, ...

- Pure, strict evaluation, strong typing:
  - Relatively unexplored (Timber, Habit, ...)
Good News, Bad News:

- **Good News:** You can write Functional Programs in almost any language

- **Bad News:** You can write “C code” in a functional language ...
Example:

Write a program to add up the numbers from 1 to 10
In C, C++, Java, C#, ...:

```c
int tot = 0;
for (int i=1; i<11; i++)
    tot = tot + i;
```

implicit result returned in the variable `tot`
In ML:

```ml
let fun sum i tot 
    = if i>10 
    then tot 
    else sum (i+1) (tot+i) 
in sum 1 0 
end
```

(result is the value of this expression)

- **accumulating parameter**
- **initialization**
- **(tail) recursion**
In Haskell:

\[
\text{sum } [1..10]
\]

result is the value of this expression
Reflections:

- I’ve tried to use “idiomatic” solutions in each language
- This example makes Haskell look good
- But it wouldn’t be too difficult to adapt any one solution to any of the other languages
- An imperative version of the Haskell solution would require linked list code that is built-in to Haskell
- An objective comparison between languages should account for library code as well as the main program
Reflections (continued):

What makes a good program?

- correctness
- clarity
- conciseness (none of my solutions are optimally concise!)
- Performance (not really an issue here)
Raising the Level of Abstraction:

"If you want to reduce [design time], you have to stop thinking about something you used to have to think about." (Joe Stoy, quoted on the Haskell mailing list)

- Example: memory allocation
- Example: data representation
- Example: order of evaluation
- Example: (restrictive) type annotations
Computing by Calculating:

- Calculators are a great tool for manipulating numbers.

- Buttons for:
  - entering digits
  - combining values
  - using stored values

- Not so good for manipulating large quantities of data.

- Not good for manipulating other types of data.
Computing by Calculating:

- What if we could “calculate” with other types of value?

- Buttons for:
  - entering pixels
  - combining pictures
  - using stored pictures

- I wouldn’t want to calculate a whole picture this way!

- I probably want to deal with several different types of data at the same time
Computing by Calculating:

- Spreadsheets are better suited for dealing with larger quantities of data.
- Values can be named (but not operations).
- Calculations (i.e., programs) are recorded so that they can be repeated, inspected, modified.
- Good if data fits an “array”.
- Not so good for multiple types of data.
Functional Languages:

- Multiple types of data
  - Primitive types, lists, functions, ...
  - Flexible user defined types ...

- Operations for combining values to build new values (combinators)

- Ability to name values and operations (abstraction)

- Scale to arbitrary size and shape data

- “Algebra of programming” supports reasoning
Quick Introductions
Starting Hugs:

```
user$ hugs

Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-2005
World Wide Web: http://haskell.org/hugs
Bugs: http://hackage.haskell.org/trac/hugs
Version: September 2006

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs>
```

The most important commands:

- `:q` quit
- `:l file` load file
- `:e file` edit file
- `expr` evaluate expression
The read-eval-print loop:

1. Enter expression at the prompt
2. Hit return
3. *The expression is read, checked, and evaluated*
4. *Result is displayed*
5. Repeat at Step 1
Simple Expressions:

Expressions can be constructed using:

- The usual arithmetic operations:
  \[ 1 + 2 \times 3 \]

- Comparisons:
  \[ 1 == 2 \quad \text{and} \quad 'a' < 'z' \]

- Boolean operators:
  \[ \text{True} \ && \ False \quad \text{and} \quad \text{not False} \]

- Built-in primitives:
  \[ \text{odd 2} \quad \text{and} \quad \text{sin 0.5} \]

- Parentheses:
  \[ \text{odd} \ (2 + 1) \quad \text{and} \quad (1 + 2) \times 3 \]

- Etc ...
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 even ask Hugs for the type of an expression: `:t expr`
Type Errors:

Hugs> 'a' && True
ERROR - Type error in application
*** Expression : 'a' && True
*** Term : 'a'
*** Type : Char
*** Does not match : Bool

Hugs> odd 1 + 2
ERROR - Cannot infer instance
*** Instance : Num Bool
*** Expression : odd 1 + 2

Hugs>
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:

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

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:
  - \( \text{head } [1,2,3,4] = 1 \)
  - \( \text{tail } [1,2,3,4] = [2,3,4] \)

- 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 \( xs \), \( \text{head } xs : \text{tail } xs = xs \)
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:
  \[
  \text{map odd } [1,2,3,4] = [\text{True, False, True, False}]
  \]
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 \( A \rightarrow B \).

Examples:
- \( \text{odd} :: \text{Int} \rightarrow \text{Bool} \)
- \( \text{fst} :: (a, b) \rightarrow a \) (\( a,b \) are type variables)
- \( \text{length} :: [a] \rightarrow \text{Int} \)
Operations on Functions:

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

- Notice that function application associates more tightly than any infix operator:

\[
\text{f x + y = (f x) + y}
\]

- In types, arrows associate to the right:

\[
A \rightarrow B \rightarrow C = A \rightarrow (B \rightarrow C)
\]

**Example:**

\[
\text{take :: Int -> [a] -> [a]}
\]

\[
\text{take 2 [1,2,3,4] = (take 2) [1,2,3,4]}
\]
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 \(\text{expr} :: A\))
- \((\oplus \text{ expr}) \:: A \to C\) (assuming \(\text{expr} :: B\))

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

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

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

- `(.) :: (a -> b) -> (c -> a) -> c -> b`
  - `(odd . (1+)) 2 = True

"composition"
Definitions:

- So far, we’ve been focusing on expressions that we might want to evaluate.

- What if we wanted to:
  - Define a new constant (i.e., Give a name to the result of an expression)?
  - Define a new function?

- Definitions are placed in files with a .hs suffix that can be loaded into the interpreter.
Simple Definitions:

Put the following text in a file “defs.hs”:

greet name = "hello " ++ name

square x = x * x

fact n = product [1..n]
Loading Defined Values:

Pass the filename as a command line argument to Hugs, or use the :l command from inside Hugs:

Main> :l defs
Main> greet "everybody"
"hello everybody"
Main> square 12
144
Main> fact 32
263130836933693530167218012160000000
Main>
Example: Calculating Fractals
Calculating Fractals:

- Based on my 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:

```haskell
type Point = (Float, Float)

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

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

The source of all that beauty & complexity!

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), ^C {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) ^C {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) ^C {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) ^C {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?
Grids:

\[ \delta_y = \frac{(y_{\text{max}} - y_{\text{min}})}{(r-1)} \]

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

```haskell
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:

\[
\text{draw} :: [\text{color}] \rightarrow \\
\quad \text{Grid Point} \rightarrow \\
\quad (\text{Grid color} \rightarrow \text{pic}) \rightarrow \text{pic}
\]
\[
\text{draw palette grid render} = \text{render (sample grid (fracImage palette))}
\]
Example 1:

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

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

example1 = 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
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 ...