Languages and Compilers
Functional Programming

Adam Sampson
School of Design and Informatics
Abertay University

Overview

- Today I'll cover:
  - What **functional programming** is
  - Why functional programming is interesting
  - Features functional languages typically have
- This will be a bit more discursive than previous lectures...
- … because I'd rather you tried some of these things in action in this week's practical exercise

What is functional programming?

- A style of programming, and a family of languages designed to support it
- Massively influential on modern programming languages – most novel features were introduced first in functional programming languages
- FP is...
  - “programming without variables”?
  - “computing by applying functions to arguments”?
  - “constructing programs by composing functions”?
  - “making programming more like mathematics”?
- None of those really capture it entirely

What is functional programming?

- A set of principles which
  - make it easier to analyse and reason about code
  - but require you to program in a different way
  - … and it turns out this is often easier/more concise

- Declarative rather than imperative
- Compositional – both types and functions
- Getting away from the machine model
- Reasoning about the correctness of your code

History

- No single starting point – FP has developed continuously from mid-60s to present

- Lisp > Common Lisp/Scheme > Racket/Clojure
  - recursion, data structures, first-class functions
- ML > Standard ML/OCaml > F#/Reason
  - static typing, type inference, generic types, formal semantics, correctness proofs
- KRC > Miranda > Haskell > Idris
  - purity, indentation, list comprehensions, lazy evaluation, typeclasses, dependent types

Functional features

- Different functional languages have wildly different sets of features – and even completely different models of evaluation (see next week)
- I'll give you a tour of the kinds of features that a typical **purely functional** language might have
  - Most languages are “impure”, or give you a choice – i.e. you can combine the functional features with more traditional programming approaches
  - Many of these features have been adopted by imperative/OO languages since the 1990s

Pure FP: immutable variables

- Once a variable's been given a value, it can't be changed – i.e. there's no assignment statement
- This means variables are *variables* in the mathematical sense: after you've said

  ```
  let x = 42
  ```

  then you can substitute 42 for x (or vice versa) at any point where x is in scope
- This is called **referential transparency**

Pure FP: pure functions

- Functions are actually *functions*, mathematically
- A function always gives the same result when given the same input – no hidden state or side effects
- After you've defined a function:

```
let square x = x * x
```

  then you can substitute `square x` with `x * x` (or vice versa) anywhere later in the program

Why's purity useful?

- It means you can effectively do algebra with bits of code – reason about them, rearrange them
- You can do this **in isolation** – and have the properties compose – so good modularity
- … and the compiler can do the same when checking and optimising the code

- The compiler can evaluate expressions in any order it likes... or in parallel... or not evaluate bits at all if they're not needed

How do you get anything done?

- No side effects? How do you read input, print results, interact with the user?
- Two approaches:
  - **Maintain purity** (e.g. Haskell):
    At the top level, your program is a function from its inputs to its outputs – running the program is evaluating the function
  - **Impure FP** (e.g. OCaml):
    Allow both pure and impure (side-effecting) functions – so you get the purity benefits for most of your code – but you lose some reasoning power

How do you get anything done?

- No mutable variables? How do I store data, accumulate results, let the user control things...?
- There are different design patterns for doing all these things in pure functional languages – often making use of functions and recursion
- e.g. a game where the player presses keys to control an object moving around the screen
  - In an imperative language: “if (key == 'd') x++;”
  - In a purely functional language, we could write a function that takes a list of object positions and a list of keypresses, and returns a new list of positions (and the code would look much the same)

First-class functions

- Functions are first-class values – you can write a function that takes another function as an argument (a **higher-order function**)

```
double x = x * 2
evens = map double numbers
```

- … or return a function from a function, or have a list of functions, etc. etc. like any other data type

- Convenient syntax for **lambda functions** (anonymous functions, “function constants”)

```
evens = map (\x -> x*2) numbers
```

Recursion being idiomatic

- In imperative languages, we're used to doing repetition using **iteration** (i.e. for/while loops)
- Functional languages use **recursion** to get the same effect without needing mutable variables
- e.g. computing the Nth Fibonacci number:

```
let rec fib n =
  if n < 2 then 1
    else fib (n - 1) + fib (n - 2)
```

- Design by inductive reasoning – base case first
- Tail recursion – doesn't blow up the stack
- (Lots more examples in paper linked at end)

Decent type systems, type inference

- We discussed this last week...
- Functional languages usually have all the type system features I described – and more
- In particular, they use Hindley-Milner (or more modern) inference systems that can infer **generic types** for functions

  ```
  let rec map f (first :: rest) = (f first) :: map f rest
  ```

  ```
  >> val map : ('a -> 'b) -> 'a list -> 'b list
  ```

- And they usually provide excellent facilities for sum types – **algebraic data types**

Cons cells

- The linked list is usually a fundamental data type, defined recursively
- The empty list:

```
[]
```

- The cons operator adds an item to the start:

```
x :: list

3 :: 4 :: 5 :: [] = [3; 4; 5]
```

Cons cells and recursion

- We can easily write functions that operate upon linked lists using recursion, especially in combination with **pattern matching**:

```
let rec sum ls =
  match ls with
  | [] -> 0
  | first :: rest -> first + sum rest
```

- i.e. the sum of the empty list is zero, and the sum of a non-empty list is the first item plus the sum of the rest of the list

Pattern-matching

- Think of pattern-matching as being like switch, but more powerful – you can use it to pull apart a data structure
- let action = match inputEvent with

```
    | Key 'a' -> movePlayerLeft
    | Key 'd' -> movePlayerRight
    | Key _ -> showMessage “unknown key”
    | Quit -> quitGame
```

- let simplify expr = match expr with

```
    | Plus x 0 -> simplify x
    | Times x 1 -> simplify x
    | x -> x
```

List comprehensions

- Another replacement for iteration – effectively a more convenient replacement for map/filter
- The syntax follows mathematical set notation: compute an expression for each item in a list, or just items matching a particular condition
- `let evens = [ 2*x | x <- nums ]`
- ```
  let squaredPrimes =
    [ x*x | x <- nums, isPrime x ]
  ```
  ```

Partial application (or **currying**)

- If I've got a function that takes more than one argument...

  let makeMessage a b = a ++ “: “ ++ b

  makeMessage “status” “out of bananas”
  >> “status: out of bananas”

- … then I can supply *some* of the arguments to get a partially-applied function:

  let makeError = makeMessage “error”

  makeError “printer on fire”
  >> “error: printer on fire”

Partial application (or **currying**)

- If I've got a function that takes more than one argument...

  let makeMessage a b = a ++ “: “ ++ b

  makeMessage “status” “out of bananas”
  >> “status: out of bananas”

- This makes sense in terms of substitution...

  makeMessage “status” “out of bananas”
  = (makeMessage “status”) “out of bananas”

Concise syntax

- Functional languages tend to have concise syntax – type inference and pattern matching help with this
- Often inspired by mathematical notation
- Indentation-based syntax was initially popularised by functional languages
  - The “offside rule” – “Next 700” paper
  - Python isn't a functional language... but it has taken a lot of ideas from functional languages over the years!

Any questions?

- If you'd like more background, read...
- John Hughes, “Why Functional Programming Matters” (1990) – good summary, many examples
  http://worrydream.com/refs/Hughes-WhyFunctionalProgrammingMatters.pdf
- Peter Landin, “The Next 700 Programming Languages” (1966) – introduced key FP ideas
  http://dl.acm.org/citation.cfm?id=365257

- **Practical**: try out OCaml – a functional language
- **Next lecture**: implementing parsing