Functional Programming: Prestudy

Jeremy Gibbons

Before you attend the functional programming course, you should familiarize yourself with some of the background and fundamentals. These prestudy notes will help you with this.

This course will use the medium of functional programming to introduce and explore some techniques that make for good programming style and increased programming effectiveness. These techniques are generally applicable: the skills developed during the course can be used in mainstream languages such as C and Java.

One essential aspect of modern functional programming is *compositionality*. Programs can be constructed out of numerous small, self-contained components that can be combined in various ways. This leads to more flexible programs, and more reusable code. Of course, compositionality is encouraged in most programming languages; the contribution of functional programming is a more, and more powerful, means of combining components. We will see many examples of this throughout the course.

Another essential aspect of modern functional programming is the ease with which complex data structures may be manipulated. The course will focus upon the use of recursive data structures such as lists and trees in the solution of a variety of problems: we might look at the implementation of a compression-encoding algorithm, the simulation of a stack-based microprocessor, or the design of algorithms for edge-detection.

The course will consist of lectures and practical sessions. The first day will be almost entirely lecture-based, and will cover the fundamentals of functional programming. The remainder of the week will involve hands-on practical sessions, and lectures in more advanced topics.

We will use *Thinking Functionally with Haskell* [2] as the course text. This is a greatly-revised second edition of the classic books *Introduction to Functional Programming Using Haskell* [3] and *Introduction to Functional Programming* [4], which have been the course texts in the past. If you would prefer a chattier, less formal introduction, you might look at *Learn You a Haskell for Great Good!: A Beginner's Guide* [14]. Also recommended are *Programming in Haskell* [11] and *Haskell: The Craft of Functional Programming* [15], which are simpler and less mathematical than Bird's book, but aimed more at undergraduate computing students

who are still learning to program. For more advanced study, you may want to look at *The Haskell School of Expression: Learning Functional Programming through Multimedia* [8] and *The Fun of Programming* [7], which introduce several interesting applications of functional programming.

The programming language used is Haskell [13], the standard lazy functional programming language. We will be using the Glasgow Haskell Compiler (GHC) and Interpreter (GHCi) [6], a state-of-the-art open-source implementation of the language that includes a wide variety of extensions. The easiest way to get started with programmming Haskell in to use the *Haskell Platform*. You will find installation instructions and a lot more about Haskell in general at the Haskell web site `haskell.org` [1].

1 Functional programming: What?

Functional programming, as the name suggests, is programming with functions. In this context, a *function* is a mapping from values (arguments) to values (results), like the mathematical square function.

Scripts and sessions

A functional program consists of a collection of equations defining values for identifiers. For example, the following simple program ('script', in functional programming parlance)

```
answer :: Integer
answer = 42

name   :: String
name   = "Jeremy"

mylove :: Bool
mylove = True
```

defines values for three identifiers: the integer value 42 for the identifier *answer*, the string value `"Jeremy"` for the identifier *name*, and the Boolean value *True* for the identifier *mylove* (see file `example.hs` in the accompanying material).

'Executing' a functional program in an interpreted system like GHCi consists of reading in a script like this, then evaluating some expressions; these expressions may use the identifiers defined in the script.

The interpreter prints the value of each expression evaluated. For example, if the above script is stored in a file `example.hs`, then the dialogue ('session', in functional programming parlance) might proceed as follows:

```
Prelude⟩ :load example.hs
[1 of 1] Compiling Main ( example.hs, interpreted )
Ok, modules loaded: Main.
*Main⟩ 1 + 2
3
*Main⟩ div answer 6
7
*Main⟩ reverse name
"ymereJ"
```

(The parts typed by the user are in italics; everything else is generated by the interpreter. The function *reverse* is defined in the standard prelude, a collection of useful definitions loaded along with the script.)

Functions

Things get more interesting when the script defines not only constant values, but also function values. For example, the script

$$square :: Integer \to Integer$$
$$square\ x = x * x$$

defines a value for the identifier *square*. The value given to this identifier is a function from integers to integers; the particular function of that type is the one that takes in a value x and returns the value $x * x$. If we ask for the value of *square* in a session, we don't get a very useful answer: the interpreter cannot print out a functional values. However, we can *use* that function in the session, and get a result that we can print out:

```
*Main⟩ square 3
9
```

What more is there?

That is pretty much all there is to it. There are more datatypes than just the usual numeric, character and logical primitives; in particular, there

are *lists* of values, and a mechanism for defining new datatypes. Nevertheless, a script still defines values for identifiers (and perhaps some new datatypes), and the session consists of evaluating some expressions using those identifiers.

'Programming' in the interactive environment is reduced to expression evaluation; in pure functional programming, there are no statements such as assignments, loops, input/output and so on. The surprising thing is that this is sufficient to write any computation — any computable task that can be expressed as a function, taking in some arguments and return a result. As well as the obvious numerical examples like *square*, this includes also examples like a compiler (a function from source code to object code), graph algorithms like route finding (a function from a graph, a source and a destination to a route), report generation (a function from a database to a textual report) and so on. Even compiled programs are pure functions.

2 Functional programming: How?

Now read Chapter 1 of the textbook [3], which covers the fundamental concepts of functional programming. Attempt some of the exercises that are scattered throughout the chapter. Use pen and paper, or if you prefer, download a copy of GHCi from the Haskell web site [6] and try your programs out for real. If you'd like more guidance before the course starts, you might want to look at the various online tutorials available [9, 5, 16].

3 Functional programming: Why?

From what we have seen so far, programming in a functional language sounds quite restrictive — rather like programming in any other kind of language, but ignoring most of the features of that language. It is not clear from what we have seen what the benefits are that functional programming provides.

Referential transparency

One significant benefit is known as *referential transparency*. Note that the absence of all 'statements' in the traditional sense means that functions necessarily have no side-effects. Thus, the expression *square* 3

is exactly equivalent in observable behaviour to its reduced form 9, or indeed to any other expression that reduces to the same value, such as $6 + 3$ or $5 +$ *square* 2. To say that a language is referentially transparent is to say that the only thing that matters of an expression is its value, not any other characteristic such as the manner in which it is defined, the time it takes to execute in a certain interpreter, the effect it has on the output stream, and so on.

A pure functional language is referentially transparent. Most other languages are not. For example, an imperative language such as Modula-2 allows the definition of functions such as the following:

```
PROCEDURE Square(x : INTEGER) : INTEGER;
BEGIN
  y := y + 1;
  RETURN x * x
END Square
```

Modula-2 and similar languages are not referentially transparent, because with such a definition the expression *Square*(3) is not equivalent to its result 9: the former additionally increments the global variable y, whereas the latter does not.

Referential transparency has a number of benefits. One is for the language implementer, who has a lot more freedom in implementation strategies. For example, in the expression *square* 3 + *square* 4, the implementer can choose to evaluate either argument of the addition first, or indeed to evaluate both arguments in parallel on two processors — the overall result will be the same. Smart compilers for traditional languages can also do this, but they need to perform elaborate checks to deduce that the expressions concerned are free of side-effects in order to ensure that the transformation is safe. The implementer of a pure functional language need perform no such check.

Another benefit of referential transparency is for the programmer. The absence of side-effects means that variables in a functional program behave just like variables in algebra — in particular, they do not vary, but rather, maintain their value throughout their existence. Thus, in functional programming, as in algebra, the two occurrences of the variable x in the expression

$$ax^2 + bx + c$$

necessarily refer to the same value; in C, on the other hand, the two occurrences of the variable x in the expression

$$(x{+}{+}) \; + \; (x{+}{+})$$

refer to different values. This means that the standard straightforward equational techniques of algebra suffice for reasoning about functional programs too; in contrast, other kinds of language require more powerful reasoning techniques.

Moreover, *lazy evaluation* [3, p. 6] means that a definition such as

$$square\ x = x * x$$

really does consist of the equation that it appears to be: in any expression, it is safe to replace an occurrence of $square\ x$ for some x with $x * x$, or vice versa. Such equations both simplify reasoning about programs and transforming programs. Without lazy evaluation, function definitions like this do not always consist of the equation they appear to be, but may require additional side-conditions about subexpressions being error free. (Can you think of an example of a function definition that requires such a side-condition?)

Compositionality

A more pragmatic benefit of functional programming languages over other kinds of language is sheer expressivity. Of course, anything that can be written in a functional language can be simulated in a more traditional language; that is how the functional language is implemented on traditional hardware, after all. However, there are many programs that are much simpler to write in a functional language than in other kinds of language (and conversely, some that are more difficult). Hughes [10] argues that functional programming offers more means of gluing programs together than traditional languages do. This means that the programmer has more choices for decomposing a program into components, and hence more flexibility in designing the program.

Larger examples

We conclude this prestudy with two more substantial examples of functional programs. These examples use features of Haskell that we have not yet covered, so don't worry if you don't follow the details now. They are intended simply to give you a feel for things to come.

Simon Peyton Jones [12] compares a functional implementation of the Quicksort sorting algorithm

$$\begin{aligned} &qsort :: (Ord\ a) \Rightarrow [a] \to [a] \\ &qsort\ [\,] \quad\ = [\,] \end{aligned}$$

```
inline void swap(int *x, int* y)
{
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

void quicksort(int a[], int l, int r)
{
  if (l < r)
    {
      int i = l;
      int j = r;
      int p = a[(l + r) / 2];
      for (;;) {
        while (a[i] < p) i++;
        while (a[j] > p) j--;
        if (i > j)
          break;
        swap(&a[i++], &a[j--]);
      };
      quicksort(a, l, j);
      quicksort(a, i, r);
    }
}
```

Figure 1: Typical C implementation of Quicksort

$$\begin{aligned} &qsort\ (a : x) = qsort\ littles +\!\!+ [\,a\,] +\!\!+ qsort\ bigs \\ &\quad \textbf{where}\ littles = [\,b \mid b \leftarrow x, b < a\,] \\ &\qquad\qquad bigs\ \ = [\,b \mid b \leftarrow x, b \geqslant a\,] \end{aligned}$$

with the typical C implementation in Figure 1. The difference in size and clarity is striking. Similar differences arise with most programs that emphasize manipulation of data structures.

Bird [3, Section 5.2] develops a program for constructing a mark list of the form

```
Name         Mark   Rank
ANDERSON       30      5
BAYLIS         62      2
```

```
CARTER          75      1
DENNIS          62      2
EDWARDS         50      4
```

from a class list of the form

```
ANDERSON  101372
BAYLIS    101369
CARTER    101370
DENNIS    101371
EDWARDS   101373
```

and exam results of the form

```
101369   62
101370   75
101371   62
101372   30
101373   50
```

The program is quite short, and is given in Figures 2 and 3 (see the file `classlist.hs` in the accompanying material).

The comparison with the equivalent program in a more traditional language is left as a final exercise.

```
-- some type abbreviations
type Name = String
type GUId  = Integer
type Mark  = Int
type Rank  = Int

type Codes = [ (Name, GUId) ]
type Marks = [ (GUId, Mark) ]
type Ranks = [ (Name, Mark, Rank) ]

-- the main function
process :: (Codes, Marks) → String
process = display · classlist

-- this lays out the table nicely
display :: Ranks → String
display = unlines · (heading :) · map line

heading          :: String
heading          = layout ("Name", "Mark", "Rank")
line             :: (Name, Mark, Rank) → String
line (xn, xm, xr) = layout (xn, show xm, show xr)

layout                 :: (String, String, String) → String
layout (xn, xm, xr) = ljust 12 xn ++ rjust 4 xm ++ rjust 6 xr

ljust, rjust :: Int → String → String
ljust n xs  = xs ++ replicate (n − length xs) ' '
rjust n xs  = replicate (n − length xs) ' ' ++ xs

-- this computes the results
classlist :: (Codes, Marks) → Ranks
classlist = rank · collate

collate :: (Codes, Marks) → [ (Name, Mark) ]
collate (cs, ms) =
  sortby fst (zip (map fst (sortby snd cs)) (map snd ms))
```

Figure 2: Functional program for computing a mark list (Part 1)

```
sortby :: (Ord b) ⇒ (a → b) → [a] → [a]
sortby f = foldr (insertby f) []

insertby :: (Ord b) ⇒ (a → b) → a → [a] → [a]
insertby f x = insert · span test
  where
  insert (xs, ys) = xs ++ [x] ++ ys
  test y = (f y < f x)

-- Candidate A has rank n + 1 if and only if there are exactly
-- n candidates with higher marks.

rank :: [(Name, Mark)] → Ranks
rank xs = map score xs
  where
  score (xn, xm) = (xn, xm, 1 + length (filter (beats xm) xs))
  beats xm (_, ym) = xm < ym

-- test data
main :: IO ()
main = putStr (process (codes, marks))
  where
  codes  = [("ANDERSON", 101372),
            ("BAYLIS",   101369),
            ("CARTER",   101370),
            ("DENNIS",   101371),
            ("EDWARDS",  101373)]
  marks = [(101369, 62),
           (101370, 75),
           (101371, 62),
           (101372, 30),
           (101373, 50)]
```

Figure 3: Functional program for computing a mark list (Part 2)

References

[1] Haskell web site. `http://www.haskell.org/`.

[2] Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015.

[3] Richard S. Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall, 1998.

[4] Richard S. Bird and Philip L. Wadler. *An Introduction to Functional Programming*. Prentice-Hall, 1988.

[5] Eric Etheridge. Haskell tutorial for C programmers. `http://www.haskell.org/haskellwiki/Haskell_Tutorial_for_C_Programmers`.

[6] GHC web site. `http://www.haskell.org/ghc/`.

[7] Jeremy Gibbons and Oege de Moor, editors. *The Fun of Programming*. Cornerstones in Computing. Palgrave, 2003. ISBN 1-4039-0772-2.

[8] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, New York, 2000. `http://www.cs.yale.edu/homes/hudak/SOE/`.

[9] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to Haskell. `http://www.haskell.org/tutorial/`.

[10] John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, April 1989. Available online at `http://www.cse.chalmers.se/~rjmh/Papers/whyfp.html`.

[11] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007. Support materials at `http://www.cs.nott.ac.uk/~gmh/book.html`.

[12] Simon Peyton Jones. On the importance of being the right size: The challenge of conducting realistic experiments. In Ian Wand and Robin Milner, editors, *Computing Tomorrow*, chapter 16. Cambridge University Press, 1996.

[13] Simon Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98: A non-strict, purely functional language. `http://www.haskell.org/onlinereport/`, February 1999.

[14] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011. `http://learnyouahaskell.com`.

[15] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, second edition, 1999. Support materials at `http://www.cs.ukc.ac.uk/people/staff/sjt/craft2e/`.

[16] Arjan van IJzendoorn. Tour of the Haskell syntax. `http://www.cs.utep.edu/cheon/cs3360/pages/haskell-syntax.html`.