CSE301(Functional Programming)

Lecture 0: Introduction to functional programming

Noam Zeilberger

September 4, 2023

1 What is functional programming?

It is difficult to give a precise definition of "functional programming", but a rough approximation is that it is

> *a style of programming that emphasizes* ***function application*** *and* ***function composition****.*

Here, application refers to the act of evaluating a function on some argument to compute a result. For example, the application of the square root function $f(x) = \sqrt{x}$ to the number two evaluates to about $f(2) \approx 1.41421$. Depending on the implementation of a programming language, performing a function application may involve machine-level operations like pushing arguments onto a stack, jumping to the function's entry point, and popping arguments off the stack when the function returns – although a functional programmer certainly has the right (and at times the obligation) to consider such low-level details, at least to a first approximation they can treat function application as a high-level abstraction, which behaves more or less like the evaluation of a mathematical function on its argument.

Function composition is the act of combining two or more functions to define a new function. For example, given three functions from numbers to numbers, say $f(x) = \sqrt{x}$, $g(x) = \sin x$, and $h(x) = e^x$, we can define another function from numbers to numbers

$$i(x) = h(f(x) + g(x))$$

which applies both f and g to its argument and then passes the sum of the two results to h. Here we observe that functions are being composed both "in sequence" (the outputs of f and g are summed and then passed as an input to h) and "in parallel" (the outputs of f and g on the input x can be computed independently). Again, although a low-level implementation of i may have to make some additional choices (such as to store the value of $f(x)$ in memory before computing $g(x)$, or conversely to store $g(x)$ before computing $f(x)$, or alternatively to compute them both in parallel on a multicore machine), the functional notation nicely captures just the logical dependencies of the computation.

It is possible to program in a functional style in almost any programming language, but a *functional programming language* is one that makes this easier, typically by including at least some of the following language features:

- **pattern-matching**, which enables the programmer to define functions more concisely by case-analysis on input values;

- **higher-order functions**, which enable the programmer to define functions that take functions as inputs and return functions as outputs; and

- a **rigorous type system**, which enables the programmer to write interfaces to functions and ensure that they are respected.

Certainly, a language need not have all of these features to count as a functional programming language. For example, untyped lambda calculus, a minimalistic language that serves as a kind of *lingua franca* of functional programming, has higher-order functions and nothing else! Still, established functional programming languages like Haskell and OCaml, as well as newer languages such as Rust (popular for systems programming) and Lean (increasingly popular for theorem proving), all include some support for pattern-matching and higher-order functions, as well as rigorous type systems with varying degrees of expressive power.

2 Why learn functional programming?

For a while, functional programming had a reputation of being mostly confined to academia, perhaps in part due to the perception of being too abstract, and in part due to real performance issues. Functional programming has been gaining in popularity, however, and by now functional languages like Haskell and OCaml have been deployed in industry for at least two decades, with a growing list of users.[1] Moreover, concepts of functional programming have been gradually incorporated into more mainstream programming languages.[2]

One reason why functional programming languages are used in both academia and industry is that, in general, they facilitate the task of passing from an abstract description of a problem to an implementation in code, more quickly and reliably. In part, this has to do with the fact that the notations of functional programming (such as pattern-matching and types) are often inspired by notations of mathematics and logic. But it also has to do with the fact that functional programming languages permit more careful control of so-called "side-effects", which roughly speaking are all those aspects of a function's behavior that one may need to take into account *other* than its input-output behavior, such as reading and writing global variables, displaying graphics to the screen, jumping to an error-handling routine, etc. Functions without any side-effects are said to be "pure", and admit much simpler reasoning principles than impure functions, for instance because a pure function will always return the same output when applied to a given input.

Closer control over side-effects also means that functional code is often easier to parallelize and run on multicore architectures than non-functional code. This property, combined with strong type systems that enable a compiler to be more aggressive when optimizing code, has reduced much of the performance gap between general-purpose functional programming languages like OCaml and Haskell and languages like C and C++. Although functional programming is

[1] See wiki.haskell.org/Haskell_in_industry and ocaml.org/industrial-users.

[2] Most recently, with Microsoft's announcement of support for lambda notation in Excel!

not yet widely used in the domain of high-performance computing, there is hope that these theoretical advantages could eventually lead to such uses.[3]

Beyond its significant practical applications, another reason to learn functional programming is simply because it is *beautiful*. The use of functional programming techniques (including pattern-matching, higher-order functions, types, and beyond) can often lead to extremely concise and elegant encodings of algorithms, and understanding how such terse, almost tautological descriptions of a mathematical problem translate into runnable code can be a thrilling intellectual exercise. In the larger scheme of things, functional programming is just a manifestation of the deep links that exist between mathematics and computation, but it is an important example from which much can be learned.

3 An example

To get a more concrete sense of what functional programming is about, let's consider a small example in Haskell, implementing the QuickSort algorithm:

$$
\begin{aligned}
&\mathit{partition} :: (a \to \mathit{Bool}) \to [a] \to ([a], [a]) \\
&\mathit{partition}\ p\ [\,] = ([\,], [\,]) \\
&\mathit{partition}\ p\ (x : xs) = \textbf{if}\ p\ x\ \textbf{then}\ (x : ts, fs)\ \textbf{else}\ (ts, x : fs) \\
&\quad \textbf{where} \\
&\qquad (ts, fs) = \mathit{partition}\ p\ xs \\
&\mathit{qsort} :: \mathit{Ord}\ a \Rightarrow [a] \to [a] \\
&\mathit{qsort}\ [\,] = [\,] \\
&\mathit{qsort}\ (x : xs) = \mathit{qsort}\ \mathit{left} +\!\!+ [x] +\!\!+ \mathit{qsort}\ \mathit{right} \\
&\quad \textbf{where} \\
&\qquad (\mathit{left}, \mathit{right}) = \mathit{partition}\ (\backslash y \to y < x)\ xs
\end{aligned}
$$

Without getting too preoccupied in the details of Haskell syntax (which you will eventually learn), here are a few comments to help understand this code:

- The program contains the definitions of two functions: *partition*[4] which takes a predicate p and a list as input, and returns a pair of lists by partitioning the input into the elements that satisfy p and the elements that don't; and *qsort*, a recursive implementation of QuickSort that uses the first element of the list as a "pivot".
- The line $\mathit{partition} :: (a \to \mathit{Bool}) \to [a] \to ([a], [a])$ is a *type signature*, declaring that *partition* is a higher-order function which takes a boolean-valued function (i.e., a predicate) as input, and that returns a function from lists to pairs of lists. We say that this type is "polymorphic" in the variable a, since the implementation of *partition* does not care about the underlying type of values stored in the list.
- The type signature $\mathit{qsort} :: \mathit{Ord}\ a \Rightarrow [a] \to [a]$ likewise declares that *qsort* is a function from lists to lists, which again is polymorphic in the type of list values so long as it supports a comparison operation (this is

[3] See, e.g., Sven-Bodo Scholz's talk at FHPNC 2021, "Is Functional HPC the Key to Low Carbon Computing?".

[4] This function is already defined in the Haskell standard library, but we include the definition here for expository reasons.

indicated by the condition *Ord a*, which is called a "type class" constraint in Haskell).

- The definitions of both functions are given by pattern-matching on the input list, with one case for the empty list [] and one case for a non-empty list. In the latter case, the pattern $x : xs$ indicates both the "head" x and the "tail" xs of the list.
- The predicate $(\backslash y \to y < x)$ used in the call to *partition* from *qsort* is an example of "lambda" notation (backslash '\' is Haskell's stand-in for the Greek letter λ). Here we define a function which, for any given input value y, returns the result of comparing y with x, where x was previously defined by pattern-matching.
- In the expression $qsort\ left +\!\!+ [x] +\!\!+ qsort\ right$, the binary operator $(+\!\!+)$ denotes list concatenation.

With those explanations in mind, do you find the Haskell code above to be simpler or more complicated than the way you would go about implementing QuickSort in a language you are more familiar with? (Certainly the answer to this question is a matter of taste, and your taste may evolve as you become more versed in functional programming.)

One nice aspect of the functional approach is that we can often determine the value of a function on a given input by hand simply by referring to its defining equations, without having to actually run the program, or even know much about the precise semantics of the programming language in which it was written. This is called *equational reasoning,* and it resembles closely to the kinds of reasoning one might find in a proof from an algebra textbook. Let's try a small example:

$$qsort\ [3, 1, 4, 2]$$

Since the input list is of the form $x : xs$, with head $x = 3$ and tail $xs = [1, 4, 2]$, the second clause in the definition of *qsort* applies, and we have that

$$qsort\ [3, 1, 4, 2] = qsort\ left +\!\!+ [3] +\!\!+ qsort\ right$$

where *left* and *right* are defined by:

$$(left, right) = partition\ (\backslash y \to y < 3)\ [1, 4, 2]$$

From the definition of *partition*, we can derive in a few steps[5] that

$$(left, right) = ([1, 2], [4])$$

[5]Repeatedly applying the definition of *partition* and introducing fresh variable names for the intermediate results, we derive that

$$\begin{aligned}
(left, right) &= partition\ (\backslash y \to y < 3)\ [1, 4, 2] = (1 : ts, fs) && \text{(since } 1 < 3\text{)} \\
\textbf{where } (ts, fs) &= partition\ (\backslash y \to y < 3)\ [4, 2] = (ts', 4 : fs') && \text{(since } 4 \not< 3\text{)} \\
\textbf{where } (ts', fs') &= partition\ (\backslash y \to y < 3)\ [2] = (2 : ts'', fs'') && \text{(since } 2 < 3\text{)} \\
\textbf{where } (ts'', fs'') &= partition\ (\backslash y \to y < 3)\ [] = ([], [])
\end{aligned}$$

and hence $(ts', fs') = ([2], [])$, $(ts, fs) = ([2], [4])$, $(left, right) = ([1, 2], [4])$.

and so it remains to recursively compute $qsort\ [1,2]$ and $qsort\ [4]$. By similar reasoning as above, we have that

$$\begin{aligned} qsort\ [1,2] &= qsort\ [] +\!\!+ [1] +\!\!+ qsort\ [2] \\ &= qsort\ [] +\!\!+ [1] +\!\!+ (qsort\ [] +\!\!+ [2] +\!\!+ qsort\ []) \\ qsort\ [4] &= qsort\ [] +\!\!+ [4] +\!\!+ qsort\ [] \end{aligned}$$

Finally, using the base case $qsort\ [] = []$ and appealing to properties of list concatenation, we derive:

$$\begin{aligned} qsort\ [3,1,4,2] &= qsort\ [1,2] +\!\!+ [3] +\!\!+ qsort\ [4] \\ &= ([] +\!\!+ [1] +\!\!+ [] +\!\!+ [2] +\!\!+ []) +\!\!+ [3] +\!\!+ ([] +\!\!+ [4] +\!\!+ []) \\ &= [1,2,3,4] \end{aligned}$$

which is indeed the result we expect of a sorting function!

4 A brief (pre-)history

Since its origins almost a hundred years ago, the history of functional programming has been filled with fruitful interactions between logicians, computer scientists, and mathematicians. Here we mention but a few key milestones.

The roots of functional programming are often traced back to the lambda calculus, which was developed by the American logician Alonzo Church starting in the late 1920s.[6] Although Church was originally interested in building a foundation for logical reasoning (analogous to, but intended to be more natural than, the system of *Principia Mathematica* introduced by Russell and Whitehead in the previous decade), collaborations with his students Kleene and Rosser helped reveal the deep computational nature of lambda calculus, and eventually Church (1936) isolated a minimal system for defining functions that is nowadays called the untyped lambda calculus. Soon afterwards, Turing (1937) proved the equivalence between the now standard notion of computability based on Turing machines (from his famous 1936 paper "On computable numbers, with an application to the Enscheidungsproblem") and Church's notion of λ-definability based on untyped lambda calculus. In a sense, we can see Turing's 1937 paper as providing the first formal link between functional programming and imperative programming[7]...although of course this was before the first computer programming languages were implemented, and even before the first digital computers!

LISP was the first functional programming language designed to be run on real computers, developed by John McCarthy in the late 1950s, with influences from lambda calculus although by no means a direct implementation of it. LISP stands for "LIS-t P-rocessing", and one of the language's important conceptual contributions was to show how much of computation could be elegantly expressed just in terms of simple manipulation of lists. Another innovation was *garbage collection,* which freed the programmer from having to deal with

[6] See Cardone and Hindley (2006) for a longer and fascinating account of the history of lambda calculus.

[7] Like Kleene and Rosser, Alan Turing was also a PhD student of Church at Princeton University, moving back to England shortly before the start of World War II. However, anecdotes about Church's advising style and about Turing's interests (see, e.g., Dana Scott's account (Shustek 2022)) suggest it is unlikely that the two ever had a close collaboration.

low-level memory management, albeit at the cost of sometimes unpredictable performance. On the more theoretical side, Peter Landin wrote a series of papers in the mid-1960s where he promoted lambda calculus as a conceptual tool for reasoning about the syntax and semantics of programming languages. In his influential article on "The next 700 programming languages", he also directly tackled the question of articulating what exactly is the difference between imperative programming and functional programming, suggesting that it has to do with the distinction "between indicating what behavior, step-by-step, you want the machine to perform, and merely indicating what outcome you want" (Landin 1966, p.162).[8]

The 1970s (and late '60s) saw some striking connections forged between programming, logic, and mathematics. One important development was the rise of *domain theory* as a mathematical framework for giving rigorous semantics to recursive programs, originating in a deep insight by Dana Scott that computer programs exhibit a form of *continuity* in the topological sense (intuitively, the more information you provide to a function about its input, the more information it can return about its output). Another important development was the *polymorphic lambda calculus*, which coincidentally was reinvented twice within the span of a few years: first by the logician Jean-Yves Girard, and then independently by the computer scientist John Reynolds. A few years later Robin Milner described a practical algorithm for performing *polymorphic type inference*, which, in another coincidence, had also been discovered a decade earlier by the logician J. Roger Hindley. These kinds of coincidences – with logicians and computer scientists bumping into each other in the realm of ideas – occur remarkably often in the history of functional programming, and are manifestations of what is sometimes called "propositions as types", or "proofs as programs", or the "Curry-Howard correspondence". Other instances of this correspondence include, for example:

- *dependent type theory*, introduced by the philosopher Per Martin-Löf as a generalization of first-order logic, and which has since served as a foundation for both proof assistants and functional programming languages;
- *control operators* such as Scheme's call/cc, which it turns out are deeply related to tautologies like Peirce's law and the law of the excluded middle;
- *modal logic*, whose modern formulation dates back to a set of axioms written down by C. I. Lewis in 1912, but which has since found applications in the analysis of staged computation.

[8] A bit further in the article, Landin continues (p.163):

> It follows that functional programming has little to do with functional notation. It is a trivial and pointless task to rearrange some piece of symbolism into prefixed operators and heavy bracketing. It is an intellectually demanding activity to characterize some physical or logical system as a set of entities and functional relations among them. However, it may be less demanding and more revealing than characterizing the system by a conventional program, and it may serve the same purpose. Having formulated the model, a specific desired feature of the system can be systematically expressed in functional notation. But other notations may be better human engineering. So the role of functional notation is a standard by which to describe others, and a standby when they fail.

Yet another important development in the 1970s was Joachim Lambek's work on *cartesian closed categories*, or "ccc"s, establishing a close connection between cccs and Church's simply-typed lambda calculus. Lambek built on Bill Lawvere's earlier work on categorical logic, as well as his own older work exploring connections between mathematical linguistics, deductive systems, and canonical morphisms in certain categories.

Over time, these theoretical advances came to be incorporated into real functional programming languages. Still in the 1970s, Sussman and Steele developed the Scheme language at MIT, which used a LISP syntax but with a semantics closer to lambda calculus, while at the same time incorporating some imperative features. During this time they also wrote the famous "Lambda: The Ultimate" series of papers. Around the same time, Milner and others at Edinburgh began work on the ML language, which continued through the 1980s and eventually evolved into Standard ML. In parallel, Gérard Huet and others at INRIA in France, influenced by ML, developed the CAML language, which later evolved into OCaml. Like Scheme, the ML family of languages combine a lambda calculus-like functional core with imperative features such as mutable references and control operators.

Meanwhile, David Turner experimented with a series of languages based on another model of evaluating functional programs, so-called "lazy" evaluation, in which different parts of the program are only evaluated as they are needed to make progress with the overall computation. His work culminated in the Miranda system, which combined laziness with polymorphism and type inference, and was distributed commercially by Turner's company Research Software Ltd. Miranda in turn was a big influence on Haskell, which originally started as a standardization project by an international committee in 1987, with the goal of designing a pure, lazy language that could be used to better communicate ideas within the functional programming community, and as a stable foundation for applications (Hudak et al. 2007). Two important early language innovations were *type classes,* introduced by Philip Wadler as a way to give a principled solution to operator overloading, and *monads*, also promoted by Wadler based on earlier ideas of Eugenio Moggi as an approach to the problem of side-effects. The publication of the Haskell 98 standard (in 1999) as well as the development of the Glasgow Haskell Compiler (GHC) contributed to the success of the language in gaining academic and industrial users, and the language has continued to evolve over the past two decades.

Today, functional programming remains an active topic of research, intersecting with an ever-widening swath of mathematics while also gaining more and more mainstream traction.

References

Cardone, Felice and J. R. Hindley (2006). "History of Lambda-Calculus and Combinatory Logic". In *Handbook of the History of Logic*. Ed. by D. M. Gabbay and J. Woods. Vol. 5. Elsevier. URL: `http://www.users.waitrose.com/~hindley/SomePapers_PDFs/2006CarHin,HistlamRp.pdf`.

Church, Alonzo (1936). "An Unsolvable Problem of Elementary Number Theory". *American J. Math.* 58:2, pp. 345–363.

Hudak, Paul et al. (2007). "A history of Haskell: being lazy with class". In *Proceedings of the 3rd ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, pp. 1–55.

Landin, P. J. (1966). "The next 700 Programming Languages". *Communications of the ACM* 9:3, pp. 157–166.

Shustek, Len (2022). "An Interview with Dana Scott". *Communications of the ACM* 65:8, pp. 25–29. Condensed transcript of a four-part series of interviews of Scott by Gordon Plotkin from November 2020 to February 2021.

Turing, Alan (1937). "Computability and λ-definability". *J. Symbolic Logic* 2: pp. 153–163.