Why Is Functional Programming Important?

Robert "Corky" Cartwright

Department of Computer Science
Rice University
Houston, Texas 77005
USA
cork@rice.edu

24 November, 2015

Models of Computation

Many different models computation but they fall into two broad categories

- Physical: computations are performed by machines that change state

  Turing Machines
  RAM models
  string rewriting systems
  unrestricted grammars
  term rewriting systems

- Mathematical: computations are proofs in a theory of program data that reduce mathematical expressions to a canonical form. These expressions are constructed from functions and constants such that reduction can be performed by a mechanical process (typically a term rewriting system). See http://epubs.siam.org/doi/abs/10.1137/0213026 or `www.cs.rice.edu/~javaplt/411/12-fall/Readings/RecPrograms.pdf`.

Which form of model is primary?

An indirect answer

- Program verification for a physical program shows that it computes an abstract function (or satisfies an abstract input-output prediate) *written in a functional language*!
- Functional programs can be formalized as definitional extensions of a logical theory of program data. For simple programs, these logical theories are akin to Peano's axioms for the natural numbers. The principal axiom (scheme) is structural induction.
- Logicians have studied the idea of definitional extensions to a logical theory but they typically focus on much uglier forms of definition than simple recursion equations.
- To formalize higher order data (functions, infinite trees and streams, constructive real numbers), we can generalize these simple logical theories (based on structural induction) by adding the notion of lazy (non-strict) evaluation. The theories become more subtle; they assert that every ascending chain of elements (under the approximation ordering introduced by Dana Scott) has a least upper bound.
- Every computable sequential function has a canonical representation as an infinite (lazy) tree. This a deep result that is unfamiliar to most computer scientists.

What is the role of functional progamming in software engineering?

- Functional programming is comparatively easy. When an efficient functional programming solution is available, seize it! Comp 140 should focus on functional programming (and not in Python!).
- Critical conceptual tool in high-level program design. Even when we implement a program in an ugly imperative source language (C++, Python, JavaScript?), we still think in terms of contracts (specifications) that are purely functional. In some cases, these contracts are easy to write; in others they are tedious and messy. In the latter case, we usually settle for informal incomplete specifications written in natural language (javadoc, scaladoc, ...) but executable specifications built from primitives defined in an accompanying functional program is the emerging model.
- Many subcomputations in real systems are best formulated as functional programs or mostly functional programs. We often make modest use of state for improved performance; my favorite optimization technique is memoization.
- The only easy way to write a parallel program to solve a problem is to decompose the problem into independent subproblems and glue the answers together using a functional program. The sweet spot for Cilk, HJ, Habanero Scala is parallel functional programming where atomic tasks may be locally imperative. (When functional code is compiled, the internals of the compiled code are imperative.)
- Functional programming can be lightweight. Unfortunately, Scala is not. Racket (Scheme) is very good for small problems. Swift looks promising but I don't have any experience yet with the language. I was a Scala evangelist until I wrote some solutions to Comp 411 assignments in Scala.

Foundation of Functional Programming: the λ-calculus WITH CONSTANTS

Every functional language has the λ-calculus or a corresponding combinatory language at its core.

Why? The λ-calculus only includes the bare essentials of functional abstraction (expressed using local variables) and application. Nothing more! But these simple ideas are extremely powerful. In principle, we do not need explicit recursion or special binding forms like `let`. This topic is covered in detail in Comp 411, but let me give you a glimpse.

Board Topics Lifted from Comp 411

- Call-by-name vs call-by-value. (Note: in imperative languages, call-by-name becomes very ugly.)
- Expanding `let` into λ.
- Eliminating explicit recursion.
- Eliminating variables.