Dr. Dobb's Portal
The World of Software Development

It's Time to Get Good at Functional Programming

If you've been wondering what functional programming is all about, don't wait any longer. Michael examines functional languages like Scala, F#, Erlang, and Haskell, and Mike Riley adds a note about functional programming with Mathematica.

By Michael Swaine, Dr. Dobb's Journal
Dec 03, 2008
URL:http://www.ddj.com/development-tools/212201710

We're approaching one of those paradigm shifts again. Knowledge of functional programming (FP) is on the verge of becoming a must-have skill. Now, we're not talking about a shift on the scale of the tectonic movement that object-oriented programming brought on. Functional programming has been around since the creation of LISP in 1958 and hasn't taken over application development, nor is it about to. FP won't make your word processing program faster or better. But there are domains where it is highly useful, and in particular FP looks like the paradigm of choice for unlocking the power of multicore processors. Functional programming languages are ideally suited to, as one developer succinctly puts it, "solving the multicore problem."

Which is a big deal, because the chipmakers have essentially said that the job of enforcing Moore's Law is now a software problem. They will concentrate on putting more and more cores on a die, and it's up to you to recraft your software to take advantage of the parallel-processing capabilities of their chips.

And that means that there are real benefits in becoming proficient in parallel functional programming, and, as will be argued momentarily, that means functional programming. The bad news? Getting good at functional programming is hard, harder than moving from iterative Pascal or Basic or C coding to object-oriented development. It's an exaggeration but a useful one: When you move to FP, all your algorithms break.

FP: What It Is

Most modern programming languages, including object-oriented languages, are imperative: A program is essentially a list of instructions intended to be executed in order. Functional programming languages derive from the declarative style branch of the language tree, where there need be no explicit flow of control. This branch bifurcates further, with one branch leading to the logic-flavored languages, primarily Prolog, and one branch leading to the functional languages, the prime historical example being Lisp. Lisp and many other FP languages are based on, and are essentially implementations of, Alonzo Church's lambda calculus, a formal model for computation.

Two defining features of functional languages are that all computations are treated as the evaluation of a function and that functional language programs avoid state and mutable data. Every variable is really a constant. You can't change the state of anything, and no function can have side effects, which is the reason why FP is ideal for distributing algorithms over multiple cores. You never have to worry about some other thread modifying a memory location where you've stored some value. You don't have to bother with locks and deadlocks and race conditions and all that mess. Some companies, Ericsson, for example, are making good money exploiting this virtue of functional programming.

This is too good to be true, of course. No side effects means no I/O, for example. You can't really do anything if you can't modify any values or change state. Yet it can be shown that the lambda calculus is equivalent to a Turing machine, so you must be able to perform calculations using FP. How?

The answer is that FP languages store state in function parameters—that is, on the stack. They also cheat when it's practical to do so, breaking the pure FP paradigm in carefully controlled ways. Monads are one such trick: Impure functions that have side effects and that can call pure FP functions but can't be called by them.

That other defining feature? Everything is a function. When I learned Lisp, my professor taught that no program should be longer than three lines, four in a pinch. The idea is that the entire program is one function, and that this function simply calls two or three other functions that, if they existed, would do the job. Then you write those functions, and the functions they require, until eventually you are writing exclusively in primitive functions, at which point you're done.

In any functional programming language, you are likely to encounter these features:

- First-class functions, or higher-order functions: Functions can serve as arguments and results of functions.
- Recursion as the primary tool for iteration.
- Heavy use of pattern matching, although technically it is not a defining feature of FP.
- Lazy evaluation, which makes possible the creation of infinite sequences and other data structures.

Here's the canonical example of a functional program, the factorial function, in Haskell:

```
factorial n = if n > 0 then n * factorial (n-1) else 1
```

Scala: A Hybrid Language

Even if you've used a functional programming language, say in college, unless you've done functional programming on the job to get real work done, you're likely to find that your experience in imperative object-oriented programming will lead you astray in embracing a functional programming mindset. The learning curve is steep.

So what's the best language for exploring FP's benefits? It depends.

Let's say you're heavily invested in Java skills and tools, and don't want to toss aside the code libraries, the skills, and tools you count on. Then you should take a look at Scala.

Scala is a multi-paradigm language that integrates essential features of iterative object-oriented programming with the FP paradigm. It compiles to JVM bytecodes—you can use Java libraries in your Scala programs and inherit from Java classes. You depend on Eclipse? There's a plug-in. Scala lets you isolate those parts of your code that truly need the benefits of FP and write everything else in Java.

Although there is a .NET version, Scala is really married to Java, having been developed by Martin Obersky, one of the designers of Java generics and the author of the current javac reference compiler. It's the Java route to FP. A good Scala site is www.scala-lang.org.

F#: A Functional Language for .NET

Or let's say you're a .NET developer, live and breath .NET, go to all the Microsoft tech conferences, and want to explore FP within the security and convenience of all your familiar tools. You'll want to investigate Microsoft's F#, recently released from the labs and rapidly transitioning to a fully-supported first-class .NET citizen. Like Scala, F# is an OOP/FP hybrid that lets you slip into FP code where you need it and stick with familiar approaches elsewhere.

F# was designed specifically for .NET based on the OCaml FP language; Don Sype led the development in the Microsoft Research Group. It gives your FP code easy access to .NET libraries and tools and is integrated well with Visual Studio.

Both Scala and F# have all the defining features of FP languages: Intense use of pattern matching, functions as first-class values, lazy evaluation. Each gives your code that immutability and statelessness that you need to exploit the multicore opportunity.

Erlang: A No-Compromises Approach

Scala and F# have the virtue of combining OOP and FP paradigms and letting you wade into the FP waters only as deeply as you need to go. But what if you want to dive right in? Then you may want to explore Erlang.

Erlang is a general-purpose language and runtime environment specifically designed by Joe Armstrong at Ericsson for building highly parallel, distributed, fault-tolerant systems. Joe, who might be prejudiced, seems to thinks it's the next Ruby, and claims that if you write an application in Erlang to run on a single-core processor, it will run four times as fast on a 4-core processor without any modification in the code. And he presents data to (almost) confirm that.

Erlang is not a hybrid like Scala and F#, but a no-compromises pure FP language. It does not have "a C-like syntax to make it easy to learn." And you won't find it all that easy to learn. One Erlang programmer who claims to love the language has nevertheless blogged his frustration at the fact that Erlang has three different expression terminators that are context-dependent.

But for the applications for which it was designed, Erlang is delivering the goods. Ericsson uses it for telecom systems work like controlling a switch or telecom apps, database apps, or Internet apps. Ericsson's AXD301 project, a couple million lines of Erlang code, has achieved 99.9999999% reliability. How? "No shared state and a sophisticated error-recovery model," Joe says.

Haskell: A Foundation for Research

Finally, what if you just want to learn this different way of programming? You understand that to get really good at thinking in FP mode you'll need to go back to school (figuratively). You want a pure (not hybrid) FP experience and you aren't expecting to get productive work out of the exploration. Is there a best language for simply learning functional programming?

A good case can be made for Haskell.

At a meeting in Portland in 1987, a group of programmers decided that the FP landscape was getting littered with divergent approaches and minor, dead-end implementations, and that it was necessary to form a committee to design a common functional programming language with the goal of having a single foundation for FP research, education, and promotion of functional programming. That language was Haskell, so yes, Haskell is a camel (a language designed by a committee), and it is also a good language for learning all the FP essentials and the style of thinking needed for functional programming.

Haskell is a no-compromises functional programming language like Erlang. It presently has no commercial implementations. But while its aspirations are more academic than commercial, there are some interesting applications of Haskell. Linspire uses Haskell for system tools development, for example, and XMonad, a window manager for the X Window System, is written entirely in Haskell. Haskell's influence can be seen in the LINQ features of C# 3.0 and elsewhere.

So you have some choices, and I haven't even mentioned Intel's language Ct or Caml or Mathematica (but see the sidebar). The sharpest distinction among FP languages is between the hybrid approaches that let you extend your current skillsets and toolkits to encompass FP and the pure FP languages that require you to move fully into their world, at least for a while. Probably both approaches are necessary, and it will be interesting to watch how this all plays out. Because one way or another, functional programming is on the rise.

Functional Programming in Mathematica

by Mike Riley

In addition to being a rich mathematical expressions parser, Wolfram Research's Mathematica delivers a complete programming environment. Mathematica's programming model is extremely flexible, letting you create math-centric applications using standard procedural, object-oriented, and functional programming approaches. As Wolfram's Jon McLoone explains, "In the most simplistic sense, we can consider functional programming to be when we pass functions as arguments into other functions" (www.wolfram.com/broadcast/screencasts/elementaryprogramming). He then goes on to show how easy it is to leverage the utility of this powerful programming approach. In McLoone's example, a compound accumulation of interest is calculated in two lines of code. The first defines an addinterest function using Mathematica's function syntax:

```
addinterest[n_]:= n*1.04
```

The syntax is straightforward: addinterest is the name of the function, the brackets contain the parameter(s) to be passed into the function, the := is Mathematica's function-assignment operator, followed by the actual function itself. The second wraps this addinterest into a defined Mathematica nested list function called, appropriately enough, NestList:

```
NestList[addinterest, 100, 20]
```

This generates the accumulation of 20 compound interest results starting with the capital amount of 100. Using this facility, highly sophisticated functional relationships can be constructed in minutes.

Once the essentials of functional programming using Mathematica are grasped, using Mathematica as a primary programming environment is a natural progression. At the International Mathematica User Conference (www.wolfram.com/news/events/userconf2008), I interviewed several professors, engineers, and scientists on their use of the program, all of which were strong proponents of Mathematica's functional programming capabilities. For example, mathematician Eric Schulz said that "over the years, I've coded in Fortran, I've coded a little in C, I've coded in Perl. I've done quite a bit of Visual Basic in the Windows world. My favorite language by far is the Mathematica language as a programming environment." Schulz's own programming work will be prominently featured in one of the "assistants" in the upcoming release of Mathematica 7. Assistants are a new extension capability that let Mathematica developers enhance the application's environment with custom features. Schulz's assistant exposes numerous visual constructs and functions in an easily accessible palette, and was written to scratch his own itch of visually interacting with Mathematica when teaching students using SmartBoard (smarttech.com).

At the same conference, Continuity Logic's Kris Carlson demonstrated conference a simulation he wrote that modeled some of the signaling pathways that govern aging. "I could program in Perl, Visual Basic, and SQL, and I was getting interested in Python. Frankly, I was a jack-of-all-trades...but I decided to drop those and focus just on Mathematica...If I had to go back to those other languages, I would need psychotherapy because I really believe Mathematica is such a superior language."

For more on these and other interviews, including an engaging conversation with Wolfram Research's founder Stephen Wolfram, check out Dr. Dobb's TV (ddj.com/ddjtvlounge.jhtml).

Mike is a DDJ contributing editor. He can be contacted at mike@mikeriley.com.