The promises of functional programming

Konrad Hinsen

THE PROMISES OF FUNCTIONAL PROGRAMMING

Konrad Hinsen

Adopting a functional programming style could make your programs more robust, more compact, and more easily parallelizable. However, mastering it requires some effort.

Since the early days of computing, software development techniques have changed almost as much as computer technology itself. Ever more powerful hardware made it possible to write ever more complex software, which both required ever better development tools and techniques and made their implementation possible. Programmers thus moved from machine code to assembly languages and then problem-oriented programming languages, which have evolved to integrate techniques such as structural and object-oriented programming. Another evolution went from monolithic programs via separately compilable modules and libraries to software component technologies. However, in one respect, today's popular programming techniques are still the same as those the pioneers used: our programs consist of statements that modify data stored in the computer's memory until that memory contains the desired result. This approach closely resembles how a computer works at the hardware level: the processor fetches data from memory, performs elementary operations on it, and writes the result back to a memory cell.

In fact, this approach is so common that most of you have probably never questioned it. And yet, an alternative approach was developed as mathematical theory in the 1930s (Alonzo Church's λ-calculus) and as a programming technique in the 1950s (John McCarthy's Lisp language)—*functional programming*. Although functional programming has been very popular in computer science research for several decades, its use for writing real-life programs is relatively recent. Of the many reasons for this, the two most important are that functional programming is very different from traditional programming (also referred to as *imperative programming*) and thus requires a lot of learning and unlearning, and that computer hardware implements the imperative programming model, so imperative programs are easier to compile into efficient machine code than functional programs. However, several clear signs indicate a growing interest in functional programming techniques—recent programming languages (such as Sun's Fortress or Microsoft's F#) have the explicit goal of supporting it. The reason is that functional programming has several advantages for concurrent and parallel programming. Experience also suggests that functional programs are more robust and easier to test than imperative ones.

In scientific computing, researchers have mainly used functional programming for symbolic processing. In fact, the most widely used functional programming language in science is probably Mathematica, although most Mathematica users don't write complex functional programs. Another example of symbolic processing is the widely used fast Fourier transform library, FFTW,[1] which uses a functional code optimizer written in OCaml to produce optimized C code for a Fourier transform of a given length. In numerical applications, functional programming doesn't yet play an important role. Perhaps the most ambitious project to introduce it into the number-crunching world was Lawrence Livermore National Laboratory's Streams and Iteration in a Single Assignment Language (SISAL) project, which started in 1983. SISAL is a functional language with parallelization support, designed specifically for the needs of scientific applications. Unfortunately, it didn't attract the attention it deserved, and funding stopped in 1996. Today, SISAL lives on as an open source project (http://sorceforge.net/projects/sisal).

This article's purpose is to explain what functional programming is and how it differs from traditional imperative programming. I also explain how functional programming helps with concurrent and parallel programming. The language I use in the examples is Clojure, a modern dialect of Lisp (see the sidebar "Clojure and the Lisp Family"), but everything said here applies equally to other function-

Clojure and the Lisp Family

The language used in the main text's examples is Clojure, a modern dialect of the Lisp language family. Lisp stands for "list processing," which hints at the motivations of John McCarthy, who developed the first Lisp language in the late 1950s for use in artificial intelligence research. Today, the most widely used Lisp dialects are Scheme and Common Lisp.

The Lisp language family's distinctive feature is the principle that "code is data." Lisp provides a syntax for a simple yet flexible data structure—nested lists. It then defines how to interpret nested lists that follow specific conventions as executable code. The advantage of expressing code in a data structure is that Lisp code can easily generate (and then execute) other Lisp code, a fact that programmers have exploited from the beginning through Lisp macros, the world's first meta-programming system and probably still its most powerful one.

Lisp has the reputation of being slow, but that isn't true anymore. Many modern compilers can produce code that can compete with C in performance when given appropriate optimization hints. However, it remains very difficult to produce programs that are both efficient and portable between compilers and platforms.

Clojure is a recent Lisp dialect that sets itself apart by three features: it has four highly optimized data structures (lists, vectors, maps, and sets) designed for pure functional programming, it offers extensive support for concurrency, and it was designed for the Java Virtual Machine with the goal of easy interoperability with other JVM languages, including Java itself. For more information about Clojure, see its Web site at http://clojure.org.

al languages, only the syntax will be different. If you're interested in functional programming, you should also read (or reread) Jerzy Karczmarczuk's 1999 article in this magazine,[2] which illustrates how functional programming can yield elegant solutions to problems in mathematical modeling.

The Basics

The fundamental principle of functional programming is that you realize a computation by composing functions. The word "function" is used here in the mathematical sense—a mapping from input values to output values—whereas what most programming languages call "functions" are subroutines that return a value. One important difference is that a function in the mathematical sense always produces the same output when given the same input. An operation such as "get the next line from a file" isn't a function because each time you call, it produces a different return value. Another important difference is that a mathematical function doesn't "do" anything other than return a value. It isn't supposed to have side effects—for example, it shouldn't write anything to a file or change a variable in memory.

If a program is composed of functions, and functions aren't supposed to change any variables, then what are variables good for? Nothing, and that's why functional programming doesn't have variables. This is probably the biggest surprise to those who discover functional programming because variables are so very fundamental to our traditional ways of writing programs. The other missing fundamental concept is loops. After all, what's the point of a loop if nothing can change between iterations because there are no variables? By now, you might be convinced that it's absolutely impossible to write a useful program in a functional style, but keep reading.

What replaces loops in functional programs is *recursion*. A function is called recursive if it calls itself—directly or indirectly. Of course, calling itself makes sense only if the arguments change between calls. Moreover, if you ever want to get out of the call-yourself chain, a recursive function must return without calling itself for some arguments. Let's look at a simple example of recursion:

```
(defn countdown [n]
  (if (zero? n)
    (list 0)
    (cons n
          (countdown
            (- n 1)))))
```

These six lines define a function `countdown` of a single argument `n`, which should be an integer (although this is neither said nor enforced, Clojure being a dynamicallly typed language like Python and JavaScript). If `n` is zero, the return value of `countdown` is `(list 0)`, a list containing the single element `0`. Otherwise (we're looking at lines four to six now), the return value is a list constructed by prepending `n` to the return value of `countdown` for `(- n 1)`, which is Clojure's way of writing `n-1`. The function call `(countdown 1)` thus returns `(cons 1 (countdown 0))`, which, after executing the recursive function call, becomes `(cons 1 (list 0))`. Looking up the definitions of the functions `cons` and `list` will tell you that the final result is the two-element list `(1 0)`. This simple example illustrates how to use recursion for looping: with each recursive call, the argument becomes smaller, up to the point where it's handled directly without any further recursion.

A closer look at the chain of recursive calls to `countdown` also reveals why functional programming can live without variables. At each recursive

call, `n` decreases by one, which looks a bit as if `n` were a variable decremented from its initial value to zero; indeed, a compiler could transform the recursive function into a subroutine with a loop over `n` for efficiency reasons. The crucial difference is that `n` isn't a reference to a piece of memory that could be modified at will, intentionally or by mistake. In fact, as anyone with debugging experience has learned the hard way, the problem with variables is often that looking at a variable's value doesn't tell you where that value came from. In contrast, you can always trace the call chain that leads to `n` having a specific value at a specific point in the function `countdown` back to its beginning. The function call chain is a complete description of the data flow through the program, and it's a very useful feature for verifying a program's correctness.

I hope I've convinced you that variables and loops aren't as essential as you might have thought. But what about other side effects? Is it really practical to work with programs that can't write data to a file? Or, in fact, produce any output? Of course, the answer is no: the pure functional programs I've described to this point will just heat up your computer. You need side effects if you want your program to have any interaction with the real world. But you can—and should—limit side effects to very few places in a program.

We can categorize functional programming languages by their attitude to unavoidable side effects. Pure languages (such as Haskell) allow them only inside special language constructs, permitting the compiler to verify the absence of accidental side effects everywhere else. Impure languages (the majority) leave the responsibility for the use of side effects fully to the programmer. As is so often the case, both approaches have their good and bad sides.

Functional Abstractions

Abstractions are fundamental to writing nontrivial programs. They permit expressing an algorithm in terms of concepts that are useful in its context, rather than in terms of operations that the computer or the programming language already happens to provide. A programmer would write a least-squares fit problem, for example, in terms of linear-algebra operations—such as matrix multiplication and solving linear systems of equations—that work on an array data structure. Compilers and libraries (written by you or by someone else) then transform the algorithm into something that a computer can handle.

The abstractions provided by popular programming languages for scientific computing include basic data structures (integer, real and complex numbers, arrays, and structures) and a notation for numerical expressions that's similar to mathematical notation. Programmer-provided abstractions are mainly subroutines. Object-oriented languages add powerful constructs for data abstraction: the programmer can add problem-specific data types to the general ones provided by compiler and runtime systems.

In functional programming, algorithmic abstractions are the most prominent. A developer identifies and implements patterns that occur repeatedly in algorithms in the form of functions, which is made possible by the fact that functional programming languages consider functions to be data. It's possible, and even very common, to write functions that take other functions as parameters, returning yet another function. Such functions are called *higher-order functions*, as opposed to first-order functions whose arguments and return values are all standard data items. In mathematics, you would use the term *operator*, an example being the derivative operator that maps a function to its derivative, which is itself a function.

As a simple example, let's consider the following operations: calculating the sum of a list of numbers, calculating the product of a list of numbers, and finding the set of all items that occur in a list of values. What these (and many more) operations have in common is a simple algorithmic pattern: you start with an initially "empty" accumulator value (0, 1, the empty set) and then iterate over a list, combining at each step the current accumulator value with one list element. The combination operations for the three examples given are addition, multiplication, and adding an item to a set. This algorithmic pattern is known as *reduction*; it's implemented in Clojure via the function `reduce`. We can thus write our three examples as

```
(defn sum [numbers]
  (reduce + 0 numbers))
(defn product [numbers]
  (reduce * 1 numbers))
(defn set-of-items [items]
  (reduce conj #{} items))
```

The first argument to `reduce` is the combination operation, which is a function of two arguments. In Clojure, we can simply use `+` and `*` for addition and multiplication because they're plain functions—there's no special notation for mathematical operators. In the last example, `conj` is a function that adds an item to a collection.

It's interesting to see how we could implement `reduce` if it weren't provided already. Here's one way to write it:

Other Functional Languages

We can group the most popular languages that support functional programming into just a few families. The oldest one, the Lisp family, is described in the "Clojure and the Lisp Family" sidebar.

The second group is the ML family, which first appeared in the 1970s. Today, its most prominent members are Standard ML and OCaml, but Microsoft's recently published F# language might soon catch up with them. The ML languages differ from the Lisp family in two respects: they're statically typed, using the Hindley-Milner inference algorithm to permit the compiler to deduce the types of most functions from the way they're used, and they propose a pattern-matching syntax for defining functions that makes many definitions look similar to common mathematical notation.

The Haskell language is the result of a collective effort to define a common functional language for use in programming language research. It's similar in many respects to the ML family, the most important difference being lazy evaluation: a data item (such as a list element) is evaluated only when its value is actually required in a computation. This is possible only in a pure functional setting because side effects would otherwise occur at completely unpredictable times. Lazy evaluation makes it possible to work with infinite data structures, avoid unnecessary computations, and define control structures as simple functions. However, lazy evaluation also makes a program's CPU time and memory usage profile more difficult to understand and generally leads to slower programs overall because of unavoidable bookkeeping overhead.

Two recent languages, Scala (for the Java Virtual Machine) and Nemerle (for the .NET platform), are hybrid languages that add functional programming features to otherwise quite standard object-oriented languages. Sun's new Fortress language, whose main intended application domain is high-performance computing, also proposes a mixture of functional and object-oriented features.

Moving on to special-purpose languages, Wolfram's computer algebra system Mathematica is based on a proprietary functional programming language. Other computer algebra systems also propose functional features, although not always to the point of allowing a full functional programming style.

```
(defn my-reduce
  [op initial items]
  (if (empty? items)
    initial
    (my-reduce
      op
      (op initial
          (first items))
      (rest items))))
```

This is again a recursive function. If its input list `items` is empty, it just returns the initial value. This is the recursion's exit; otherwise, it applies the function `op` to the initial value and the first element of the list `((op initial (first items)))` and feeds the result to a recursive call on what's left of the list after removing the first element `((rest items))`. Note that Clojure doesn't treat the argument `op` in any special way merely because it's a function. Functions are perfectly ordinary data items, just like integers and text strings.

Functions can also create and return other functions, as illustrated in the following (somewhat contrived) example, which defines a function `make-adder` that takes a numerical argument `x` and returns a function of another numerical argument `y` that adds `x` and `y`:

```
(defn make-adder [x]
  (fn [y] (+ x y)))
```

Calling `(make-adder 2)` returns a function that adds 2 to its argument. We can use this function just like any other one, such as

```
((make-adder 2) 3)
```

which yields 5. Note that the result of `(make-adder 2)` is a function that stores an integer value (2) internally that was passed as an argument to `make-adder`. A function that captures a value in this way is called a *closure*; it's a widely used programming technique in functional programs. As you might have guessed, this is where the Clojure language derived its name, with the "j" hinting at Java.

Concurrency and Parallelism

Concurrency (the existence of several execution threads operating on the same data) and parallelism (the division of a computational task into multiple communicating processes running in parallel) are two aspects of computing rapidly gaining in importance. The main reason is that single-processor performance is no longer improving at the pace it used to; instead, computers are becoming more powerful by integrating more computational cores. Exploiting such machines requires concurrency, parallelism, or both. Unfortunately, today's mainstream techniques for concurrent and parallel programming are difficult to learn and quite error-prone in practice.

The big issue with concurrency is the difficulty of maintaining the data in a coherent state: it must be impossible for one execution thread to modify data that another one is accessing (reading or writing) at the same time. So, if several threads need to modify a data item, they must do so in a coordinated way. This is currently achieved via locks, but they're difficult to use, and their incorrect use can go un-

noticed for a long time before errors show up. As for parallelism, the main difficulties are identifying independent computations inside a program and coordinating them with the required communication operations such that the resulting program always produces the correct result and does so efficient-ly for typical input parameters.

Fu. ctional programming is frequently cited as a promising tech-nique in this context. Pure functional code has no variables and thus no data coherence issues or need for locking. Moreover, all data dependencies are explicit, making it possible to apply a large number of program transforma-tions (with the goal of parallelization) while guaranteeing that the program's result won't change.

If this makes you hope that automatic parallelization will be your welcome gift once you succeed in entering the world of functional programming, you're in for disappointment. Although it's true that compilers for functional languages could in principle transform a serial into an equivalent parallel program automatically, there remains the problem of finding such a transformation that actually yields an efficient program for a given parallel computer and given input data. Compiler technology isn't yet up to this task, although this could well change in the future, in particular with parallelizing just-in-time compilers that have access to a program's execution time profile. For the near future, it's reasonable to expect compilers that create parallel programs semiautomatically based on programmer-provided performance hints. This would already be a significant step forward compared to today's parallel programming techniques.

References

1. M. Frigo and S.G. Johnson, "The Design and Implementation of FFTW3," *Proc. IEEE*, vol. 93, no. 2, 2005, pp. 216–231.
2. J. Karczmarczuk, "Scientific Computation and Functional Programming," *Computing in Science & Eng.*, vol. 1, no. 3, 1999, pp. 64–72.