Why no one uses functional languages[1]

Editor: *Philip Wadler, Bell Laboratories, Lucent Technologies*; *wadler@research.bell-labs.com*

Philip Wadler

To say that no one uses functional languages is an exaggeration. Phone calls in the European Parliament are routed by programs written in Ericsson's functional language Erlang. Virtual CDs are distributed on Cornell's network via the Ensemble system written in INRIA's CAML, and real CDs are shipped by Polygram in Europe using Software AG's Natural Expert. Functional languages are the language of choice for writing theorem provers, including the HOL system which helped debug the design of the HP 9000 line of multiprocessors. These applications and others are described in a previous column [1].

Still ... I work at Bell Labs, where C and C++ were invented. Compared to users of C, "no one" is a tolerably accurate count of the users of functional languages.

Advocates of functional languages claim they produce an order of magnitude improvement in productivity. Experiments don't always verify that figure — sometimes they show an improvement of only a factor of four. Still, code that's four times as short, four times as quick to write, or four times easier to maintain is not to be sniffed at. So why aren't functional languages more widely used?

1 Reasons

Here is a list of some of the factors that inhibit adoption of functional languages. I'll note some research aimed at ameliorating these factors. If you know of relevant projects that I've failed to mention, please bring them to my attention.

Most of these factors remain serious impediments for most systems. Notable exceptions are Ericsson's Erlang (`www.erlang.se`) and Harlequin's ML Works (`www.harlequin.com`), two industrial-grade systems with extensive user environments and support.

Compatibility Computing has matured to the point where systems are often assembled from components rather than built from scratch. Many of these components are written in C or C++, so a foreign function interface to C is essential, and interfaces to other languages can be useful.

The isolationist nature of functional languages is beginning to give way to a spirit of open interchange. Serious implementations now routinely provide interfaces to C, and sometimes other languages. Interworking with the imperative world is straightforward for strict languages like ML or Erlang, but trickier for lazy languages like Haskell or Clean, since laziness makes the order of evaluation difficult to predict. However, through a pleasing interplay of theory and practice, recent research has shown how abstract concepts such as monads or linear logic can be applied to smoothly interface lazy functional languages to the real world [2, 3].

Conquering isolationism is a task for everyone, not just functional programmers. The computing industry is now beginning to deploy standards, such as CORBA and COM, that support the construction of software from reusable components. Recent work allows any Haskell program to be packaged as a COM component, and any COM component to be called from Haskell. Among other applications, this allows Haskell to be used as a scripting language for Microsoft's Internet Explorer web browser [4].

Libraries The fashionable idea of software reuse has been around for ages in the form of software libraries. A good library can make or break a language. Users are attracted to Tcl primarily on the strength of the Tk graphics library. Much of the attractiveness of Java has little to do with the language itself, but with the associated libraries for graphics, networking, databases, telephony, and enterprise servers. (Much of the unattractiveness of Java is due to the same libraries.)

Considerable effort has been extended on developing graphic user interface libraries for functional languages. Haskell boasts a plethora: Fudgets, Gadgets, Haggis, and Hugs Tk. SML/NJ has two, eXene and SML Tk. The SML language comes with a powerful module system,

[1] To appear in *ACM SIGPLAN Notices*.

which makes flexible libraries easier to construct. One example of such a library is ML RISC, a retargetable back end that has been used for SML and C compilers and has been adopted to a number of architectures [5].

Portability I have heard of numerous projects where C won out over a functional language, not because C runs faster (although often it does), but because the hegemony of C guarantees that it is widely portable. For example, researchers at Lucent would have preferred to build the PRL database language using SML, but chose C++ because SML was not available on the Amdahl mainframe they were required to use. On the other hand, abstract machines are a popular implementation technique, for both functional languages and for Java, in part because writing the machine in C makes it is easy to port to a wide variety of architectures.

Availability Even when a functional language has been ported to the machine and operating system at hand, it may not be easy to use. For example, a typical response from a user of Glasgow Haskell is that installing it was an "adventure".

Large projects are understandably reluctant to commit to a language unless it comes with a guarantee of continuing support. A few functional languages are available commercially: Research Software markets Miranda, Abstract Hardware markets Poly ML, ISL markets Poplog/SML, Harlequin markets ML Works, and Ericsson has a division devoted to support of Erlang. Nonetheless, for many functional languages, it remains difficult to ensure a stable source and reliable support.

An additional problem arises because functional languages are often under active development, creating tension between the needs of stability and research. The Haskell community is attempting to resolve these by defining Standard Haskell, a version of the language that will remain stable and supported while other versions of Haskell continue to evolve [6].

Packagability Following the LISP tradition, many functional language implementations offer a read-eval-print loop. While convenient, it is also essential to provide some way to convert a functional program into a standalone application program. Most systems now offer this. However, these systems often incorporate the entire runtime package for the library, and thus have unacceptably large memory footprints. An ability to develop compact standalone applications is essential.

Tools To be usable, a language system must be accompanied by a debugger and a profiler. Just as with interlanguage working, designing such tools is straightforward for strict languages, but trickier for lazy languages. However, there are few debuggers or profilers for strict languages, perhaps because constructing them is not perceived as research. That is a shame, since such tools are sorely needed, and there remains much of interest to learn about their construction and use.

Constructing debuggers and profilers for lazy languages is recognized as difficult. Fortunately, there have been great strides in profiler research, and most implementations of Haskell are now accompanied by usable time and space profiling tools. But the slow rate of progress on debuggers for lazy languages makes us researchers look, well, lazy.

At a larger scale, one wants integrated development environments and software engineering methodologies. Building an integrated development environment is a lot of work with little research content, so it is not surprising that this has attracted little attention. But there is plenty of interesting work to be done in applying software methodologies to functional languages, and it is disappointing that there is virtually no effort in this area.

Training To programmers practiced in C, C++, or Java, functional programs look odd. It takes a while to come to grips with writing `f(x,y)` as `f x y`. Curried food and curried functions are both acquired tastes.

Programmers practiced in imperative languages are used to a certain style of programming. For a given task, the imperative solution may leap immediately to mind or be found in a handy textbook, while a comparable functional solution may require considerable effort to find (even if once found it is more elegant). And though there are a large range of problems that possess efficient solutions in a functional language, there remain some tough nuts for which the best known solutions are imperative in style. (For these reasons, many functional languages provide an escape to the imperative style, for instance SML includes updateable references as a basic data type, and Haskell provides them via monads [7].)

The training problem is not intractable. Software AG found they could train industrial programmers to use Natural Expert in a one-week course that included lazy evaluation, polymorphic types, and higher-order functions. Typically, students were miffed when the compiler would repeatedly reject programs for type errors, but pleasantly surprised when their programs finally passed the type checker and ran correctly on the first try [8].

Popularity If a manager chooses to use a functional language for a project and the project fails, then he or she will certainly be fired. If a manager chooses C++ and the project fails, then he or she has the defense that the same thing has happened to everyone else.

While management problems are a significant barrier, the flipside is a significant opportunity: a large project that is in trouble may be willing to consider switching to a functional language because the increase in productivity may get them out of a jam. An effective way in can be to offer to prototype the solution in a functional language, and once the prototype is running show how to scale it to a full solution.

While managers have their worries, so too do managees. Experience with C++ or Java will buff up your resume nicely, while Haskell or SML will do you little good. Lucent's Pdiff system, written in SML, is a key tool in maintaining database software for the 5ESS switch. No developer could be found willing take on the role of maintaining the system, and eventually a physicist looking to switch fields was hired.

2 Non-reasons

On the other hand, there are two pieces of common cant as to why people don't use functional languages to which I do not subscribe.

Performance This might have been a reason a decade ago, but these days the performance of functional languages often rivals C. That's a rough estimate. Performance can be significantly inferior to C for some applications, and a wee bit better for others. But as a rough starting point, within a factor of two of C seems fair.

More importantly, experience shows that while performance that rivals C helps, it is not a requirement for success. Tcl/Tk, Perl, and Visual Basic all rose to prominence with implementations that are interpreted. Java has become enormously successful with performance significantly short of C. In the functional world, Erlang achieved its first successes as an interpreted language.

One has languages with high performance that are not widely used, and languages with middling performance that are widely used. Performance is sometimes an issue, but it is rare for it to be the deciding factor. It is imprudent to expect that all we need do is make functional languages run blindingly fast in order for them to become immensely popular.

"They don't get it" Functional programming is beautiful, a joy to behold. Once someone understands functional programming, he or she will switch to it immediately. The masses that stick with outmoded imperative and object-oriented programming do so out of blind prejudice. They just don't get it.

The above paragraph echoes beliefs deeply held by many researchers. But the long list in the preceding section should make it clear that it may be possible to be attracted by functional programming, but still find it unusable.

For instance, here is a posting to the Haskell mailing list.

> I have been trying to learn Haskell and have been impressed with both its elegance and the way it allows me to write code that works on the first try (or two). However, I am not a researcher. I do commercial software development and need some documentation and stability. [9]

Mailing lists related to functional languages are rife with requests for foreign function interfaces, libraries, and tools.

Doubtless, there are prejudiced individuals out there, accustomed to C and its variants and dismissive of alternatives. But many out there do "get it", and eschew functional programming for other reasons.

3 Lessons

To summarize, there are a large number of factors that hinder the widespread adoption of functional languages. To be widely used, a language should support interlanguage working, possess extensive libraries, be highly portable, have a stable and easy to install implementation, come with debuggers and profilers, be accompanied by training courses, and have a good track record on previous projects. It helps if the implementation is efficient, but this is not an absolute requirement. Potential users may find the language attractive, but reject it because of some or all of the preceding factors. Here are the lessons I draw from this exercise.

Killer App The factors listed constitute a significant barrier to use of functional languages, but not an absolute barrier. A user will forego many conveniences if given a compelling reason to do so. Tcl/Tk and Perl rose to prominence without benefit of debuggers or profilers.

Some researchers hope that the high-level nature of functional languages will prove compelling on its own, but experience to date suggests this hope is misplaced. Instead, experience shows that users will be drawn to a language if it lets them conveniently do something that otherwise is difficult to achieve. Like other new technologies, functional languages must seek their killer app.

A previous column listed a number of such applications, stressing how each exploited some strength of functional languages [1]. Telecommunications developers are drawn to Erlang by its support for concurrency and distribution; the latter is tied directly to the fact that functional data, being immutable, is well suited for transmission across a network. Creators of theorem provers are drawn to ML by its support for symbolic computations. Geneticists are drawn to CPL/Kleisli because its type system supports access to heterogeneous databases, and because the mathematical properties of functional languages can be exploited in query optimization. Expert system developers are drawn to Natural Expert because lazy evaluation resembles reasoning by backward chaining, and because lazy evaluation enables a space-efficient interface to databases.

Top-notch functional programming research is often tied to applications. Carnegie-Mellon grounds its functional programming work in the Fox project, which aims to build network drivers in SML. Chalmers researchers have close relations with Carlstedt and Logikkonsult, and among other things have applied partial evaluation to airline scheduling. Glasgow teamed up with York to produce a whole book of applications. The Oregon Graduate Institute is teaming up with Intel to look at hardware design. Yale researchers have applied functional programming to music performance and natural language understanding, and are teaming up with Microsoft to look at animation. However, most of this research has not centered around application libraries or packages that might attract significant user communities.

Applications have unexplored depths. Jump in, the water's fine!

Research emphasis Despite the applications work listed above, functional programming researchers place far more emphasis on developing systems than on applying those systems. Further, the bulk of effort is devoted to language design, program analysis, and the construction of optimizing compilers, with far less to debuggers, profilers, and software engineering tools and methodologies.

Shifts in research emphasis may require shifts in the reward structure. As Kuhn noted in *The Structure of Scientific Revolutions*, the mainstream of academic work consists of incremental contributions to existing paradigms. Within functional programming, the mainstream is program analysis and compiler development. Leaders in the field need to move into the new areas of tools and applications, and conferences and journals need to explicitly welcome contributions in these areas. Gopal Gupta is organizing the PADL 99, the First International Conference on Practical Aspects of Declarative Languages [10]. To aid a paradigm shift, a field may set out new criteria for judging new work. Simon Peyton Jones and myself have just completed an editorial for the *Journal of Functional Programming* that welcomes papers on functional programming practice and experience, and sets out the criteria we apply to judge them [11].

A modest proposal Even a modest implementation of a functional language should provide a foreign function interface, a debugger, and a profiler. By this measure, I know of only a few modest implementations of functional languages, including Ericsson's Erlang, Harlequin's ML Works, and INRIA's CAML.

Andrew Tolmach and Andrew Appel devised an ingenious debugger for the SML/NJ implementation [12], but as the implementation evolved the debugger was not maintained, and there is no debugger available for the current release of SML/NJ.

There is a tension between building useful systems and extending the frontiers of research, and functional language researchers can pride themselves on having found the resources to build some excellent systems. We now need to take the next step, and ensure these systems include essential interfaces and tools. We should no longer settle for implementations that are not even modest.

Hope This long list of reasons why no one uses functional languages may look depressing, but I prefer to look on the bright side. People do not reject functional languages because of stupidity, rather they reject them for a variety of good reasons. Stupidity is famously resistant to attack — these other problems are something we can tackle.

References

[1] Philip Wadler, An angry half-dozen, *ACM SIGPLAN Notices* 33(2):25-30, February 1998. [NB.

Table of contents on the cover of this issue is wrong.]

[2] Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, September 1997.

[3] Rinus Plasmeijer and Marko van Eekelen, Pure and efficient functional programming using the "unique" features of Clean. *ACM SIGPLAN Notices*, to appear.

[4] Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM components in Haskell. *IEEE Fifth International Conference on Software Reuse*, Vancouver, BC, June 1998.
`www.haskell.org/active/activehaskell.html`

[5] Lal George, MLRISC: Customizable and Reusable Code Generators, Bell Labs technical report, May 1997.
`www.cs.bell-labs.com/cm/cs/what/smlnj/doc/MLRISC/`

[6] John Hughes, editor, Standard Haskell.
`www.cs.chalmers.se/~rjmh/Haskell/`

[7] J. Launchbury and S. L. Peyton Jones, Lazy functional state threads. In *ACM Conference on Programming Language Design and Implementation*, Orlando, Florida, 1994.

[8] Nigel W. O. Hutchison, Ute Neuhaus, Manfred Schmidt-Schauss, and Cordy Hall. Natural Expert: a commercial functional programming environment. *Journal of Functional Programming*, 7(2):163–182, March 1997.

[9] S. Alexander Jacobson `alex@i2x.com`, letter to Haskell mailing list, 3 May 1998.

[10] Gopal Gupta, chair, First International Conference on Practical Aspects.
`www.cs.nsmsu.edu/~complog/conferences/padl99/`

[11] Simon Peyton Jones and Philip Wadler. Editorial: Practice and experience papers, *Journal of Functional Programming*, to appear.
`www.dcs.glasgow.ac.uk/jfp/`

[12] Andrew Tolmach and Andrew Appel, A Debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, April 1995.

Philip Wadler works with the Unix and ML groups at Bell Labs. He must like working with others: he is co-designer of the Haskell and GJ languages, co-author of Introduction to Functional Programming*, and co-Editor-in-Chief of the* Journal of Functional Programming.