How enterprises use functional languages, and why they don't

Philip Wadler Bell Labs, Lucent Technologies wadler@research.bell-labs.com

Logic programming and functional programming row in the same boat. Methods used to achieve success with one often transpose to the other, and both face similar obstacles. Here I offer a compendium of success stories for functional programs, followed by a list of obstacles to more widespread use of functional programming, in the belief that much of this experience is relevant to logic programmers. This material first appeared as columns in ACM SIGPLAN Notices [29, 30]. The final section contains a few remarks specific to the relations between functional and logic programming.

1 An angry half dozen

"Have you used it in anger yet?"

The time is a dozen years ago, the place is Oxford, and my fellow postdoc has just scrutinized my new bike. He's admired the chrome, checked the gears, noted the Kryptonite lock. Now he wants to know if I've used it to serious purpose. Gleaming chrome is well and good, but will it run you through the woods?

"Have you used it in anger yet?"

Seeing that my topic is functional languages, you may have just asked the same question, though perhaps in different words. You've scrutinized functional languages. You've admired the elegance of lambda calculus, checked the benchmarks from the compilers, noted the security provided by strong typing. Now you want to know if they have been used to serious purpose. Mathematical elegance is well and good, but will it run that mission-critical system?

Here are a half-dozen examplars of functional programs used in anger.

1.0 Compilers

This one's a freebie. I won't count it toward the six, as it is obvious and incestuous.

Most compilers for functional languages are implemented in the language they compile. The Standard ML of New Jersey compiler (SML/NJ) is about 130K lines of Standard ML. The Glasgow Haskell compiler is about 90K lines of Haskell. Caml, another dialect of ML, is implemented in Caml. Erlang is implemented in Erlang, and some versions of Scheme in Scheme.

In some corners, functional languages bear a reputation for gross inefficiency, but this reputation is out of date. Code quality ranges from a shade better than C to an order of magnitude worse, with the typical case hovering at a factor of two or so slower. One example is the Pseudoknot benchmark, based on an application that uses backtracking search to determine three-dimensional protein structure. A large number of functional languages were benchmarked against this program, the best running two to three times slower than the equivalent C [15].

The functional community splits into two camps. Lazy languages evaluate arguments on demand, and so require highly disciplined use of side effects; strict languages evaluate arguments eagerly, but make it easier to exploit side effects. Haskell, Miranda, and Clean are lazy; Standard ML, Caml, Erlang, and Scheme are strict. Over the past few years there has been remarkable convergence between the two communities, and the Pseudoknot tests show lazy and strict languages have comparable performance.

Most functional languages now provide some means of interworking with programs written in C or other imperative languages. Interworking is straightforward for a strict language, but trickier for a lazy language. A key advance of recent years, achieved by a pleasing interplay of theory and practice, is to obtain interworking for lazy languages via such abstract concepts as monads and linear logic [28, 23]. Profiling systems for functional languages have also improved vastly, and the usual code-measure-improve cycle is now routinely applied to improve the time and space behaviour of functional programs [26]. However, there are still few good debuggers for functional languages.

1.1 HOL and Isabelle

Hewlett-Packard's Runway multiprocessor bus underlies the architecture of the HP 9000 line of servers and multiprocessors. Hewlett-Packard applied the HOL (Higher-Order Logic) theorem prover to verify liveness properties of the arbitration protocols in Runway. Verification was achieved by a hybrid of theoremproving in HOL and model-checking in SMV. This approach uncovered errors that had not been revealed by several months of simulation [6].

The Defence Science and Technology Organization, a branch of the Department of Defence in Salisbury, South Australia, is applying the Isabelle theorem prover to verify arming conditions for missile decoys. A graphical front-end has been added to Isabelle for this purpose, humorously called DOVE (Design-Oriented Verification and Evaluation) [21].

Both HOL and Isabelle are implemented in Standard ML. Standard ML is a descendant of ML, the *metal*anguage of the ground-breaking LCF theorem prover. LCF in turn is an ancestor of both HOL and Isabelle. This circle reflects the intertwined history of theorem provers and functional languages [12, 22, 13].

ML/LCF exploited two central features of functional languages, higher-order functions and types. A proof tactic was a function taking a goal formula to be proved and returning a list of subgoals paired with a justification. A justification, in turn, was a function from proofs of the subgoals to a proof of the goal. A tactical was a function that combined small tactics into larger tactics. The type system was a great boon in managing the resulting tangle of functions, where some functions accept functions as arguments, and some of these return functions as results. Further, the type discipline ensured soundness, since values of the abstract data type "theorem" could only be created by a specified set of functions, each one of which corresponded to an inference rule of the logic. The type system Robin Milner devised for ML remains a cornerstone of work in functional languages.

HOL and Isabelle are just two of the many theorem provers that draw on the ideas developed in LCF, just as Standard ML is only one of the many languages that draw on the ideas developed in ML. Among others, Coq is implemented in Caml, Veritas in Miranda, Yarrow in Haskell, and Alf, Elf, and Lego in Standard ML again. An upcoming issue of the *Journal of Functional Programming* is devoted to the interplay between functional languages and theorem provers. Most theorem provers are written in functional languages, with the exception of a few systems written in Lisp (the granddaddy of functional languages).

1.2 Erlang

Ericsson's Mobility Server is marketed in twelve countries. Among other things, it controls some mobile phones for the European Parliament in Strasbourg. The Mobility Server is one of a range of Ericsson products implemented using Erlang, a functional language designed by Ericsson for telecommunications applications. At last count, Ericsson marketed eight products based on Erlang [1].

Ericsson has a separate division, Erlang Systems, that handles marketing, training, and consulting for Erlang. Over one thousand Ericsson employees have attended Erlang course and over five hundred are currently involved in product development using Erlang. The Mobility Server contains hundreds of thousands of lines of Erlang code, and products written in Erlang have earned Ericsson millions of kronor.

You might guess Erlang stands for "Ericsson Language", but actually it is named for A. K. Erlang, a Danish mathematician who also lent his name to a unit of bandwidth. (A phone system designed to bear 0.33 erlang will work even if one-third of its phones are in use at the same time.)

Erlang is dynamically typed in the same sense as Lisp, Scheme, or Smalltalk, which makes it one of the few modern languages to eschew ML's heritage of static typing. The basic data types are integers (with arbitrary precision, so overflow is not a problem), floats, atoms, tuples, lists, and process identifiers.

Primitives allow one to spawn a process, send a message to a process, or receive a message. Any data value may be sent as a message, and processes may be located on any machine. Erlang uses compression techniques to minimize the bandwidth required to transmit a value. Thus it is both trivial and efficient to send, say, a tree from one machine to another. Compare this with the work required in a language such as C, C++, or Java, where one must separately establish a connection, serialize the tree for transmission, and apply compression. To support robust systems, one process can register to receive a message if another process fails.

Ever since Guy Steele's pioneering work on Scheme, tail-recursion has been a mainstay of functional languages, and it is put to good use in Erlang. A server in Erlang is typically written as a small function, with arguments representing the state of the server. The function body receives a message, performs the computation it requests, sends back the result, and makes a tail-call with parameters representing the new state. Finite state machines are easily represented: just have one function for each state, with state transitions represented by tail calls. The daunting tasks of changing running code on the fly is solved by a surprisingly simple use of higher-order functions and tail-calls: design the server to receive a message containing a new function for the server, which is applied with a tail-call; a new variable can be added to the server state by a tail-call to a function with an added parameter.

Functional programmers often claim that the use of higher-order functions promotes reuse. The classic examples are the *map* and *fold* functions, which encapsulate common forms of list traversal, and just need to be instantiated with an action to perform for each element. Most, but not quite all, list processing can be easily expressed in terms of these functions. The Erlang experience suggests this notion of reuse scales up to support concurrent client-server architectures. A set of libraries encapsulate common server requirements, and just need to be instantiated with the action to be performed for each request. Most, but not quite all, required servers can be easily expressed in terms of these libraries.

Erlang bears a striking resemblance to another modern phenomenon, Java. Like Java, Erlang (along with all other functional languages) uses heap allocation and garbage collection, and ensures safe execution that never corrupts memory. Like Java, Erlang comes with a library that provides functionality independent of a particular operating system. Like Java, Erlang compiles to a virtual machine, ensuring portability across a wide range of architectures. And like Java, Erlang achieved its first success based on interpreters for the virtual machine, with faster compilers an afterthought.

Erlang succeeded not just because it was a good language design, but because its designers took the right steps to promote its growth. They evolved the language in tandem with its applications, worked closely with developers, and provided documentation, courses, hot-lines, and consultants. A foreign-language interface was essential to allow interworking with existing software in C. Users were often attracted to Erlang by the availability of tools and packages, such as an interface compiler for the ASN.1 exchange standard, and a real-time highlyreliable distributed database system called Mnesia, both implemented entirely in Erlang.

1.3 Pdiff

If you've ever made a long-distance phone call in the US, you've probably used a Lucent 5ESS phone switch. Each 5ESS contains an embedded, relational database to maintain information about customers, network topology, rates, features such as call waiting, and so on. The database is complex, containing nearly a thousand relations, and there are tens of thousands of consistency constraints (also called *population rules*) that the data must satisfy [11, 7]. As new features are added to the switch, new transactions are required; for instance, one may need a new transaction to register a customer for call waiting. Each transaction should be *safe* in that it should leave the database in a consistent state. Ensuring safety was difficult and error prone, especially since the constraints were embedded in C programs that audit the database for consistency, and transactions were performed by other C programs.

The first step was to introduce PRL (Population Rule Language) to describe constraints and transactions. This marked a vast improvement over the use of C, but left the problem of determining for each transaction what conditions must be satisfied to ensure safety.

The next step was to introduce Pdiff (PRL differentiator). The input to Pdiff is the safety constraint for the database and an unsafe transaction, both written in PRL. Pdiff computes what condition must hold in advance of the transaction to ensure the database is consistent afterward. (This is similar to Dijkstra's computation of the weakest precondition that must hold in advance of a command to ensure a given predicate holds afterward.) Additional steps simplify this condition on the assumption that the database is consistent before the transaction. The output is a safe transaction in PRL, which checks all the necessary constraints for validity before allowing any change to the database.

Pdiff consists of about 30K lines of code written in Standard ML, written by researchers at Bell Labs. Pdiff improves the quality and reliability of switches, reduces the time to deploy new features, and has saved Lucent millions of dollars in development costs.

The Pdiff history reveals some problems of using a functional language in practice. When the time came to hand off maintenance of Pdiff to the 5ESS staff, no internal candidate could be found for the role. Developers prefer to have C++ or Java on their resume, and balk at languages perceived as "weird". Eventually a physicist looking to change fields was hired for the purpose. An opportunity was missed when the 5ESS team considered using Standard ML to write the PRL compiler. Since Standard ML wasn't available for their machine (an Amdahl), they used C++ instead.

1.4 CPL/Kleisli

In April 1993, a workshop organized by the US Department of Energy considered the database requirements of the Human Genome Project. An appendix of the workshop report listed twelve queries that would be difficult or impossible to answer with current database systems, because they require combining information from two or more databases in disparate formats [8].

All twelve of these queries have been answered using CPL/Kleisli. CPL (Collection Programming Language) is a high-level language for formulating queries. Kleisli, the system that implements CPL, translates CPL into SQL for querying relational databases, or runs the queries against data in ASN.1, ACE, or other formats. CPL/Kleisli is in active use at the Philadelphia Center for Chromosome 22 and at the BioInformatics Centre of the Institute for Systems Science in Singapore [4].

Functional programming plays two roles here: CPL is a functional language, and Kleisli is written in Standard ML. The basic data types of CPL are sets, bags, lists, and records. The first three of these may be processed using a comprehension notation familiar to mathematicians and functional programmers. For instance, a mathematician may write $\{x^2 \mid x \in Nat, x < 10\}$ for the set of squares of natural numbers less than ten. Similarly, the CPL query

```
{ [ Name = p.Name, Mgr = d.Mgr ] |
    \p <- Emp, \d <- Dept,
    p.DNum = d.DNum }</pre>
```

returns a set of records pairing employees with their managers. The comprehension notation is reminiscent of SQL, where one may write

```
SELECT Name = p.Name, Mgr = d.Mgr
FROM Emp p, Dept d
WHERE p.DNum = d.DNum
```

for the same query. But CPL allows sets, bags, lists, and records to be arbitrarily nested, whereas SQL can only process "flat" relations, consisting of sets of records. The extra nesting in CPL helps one formulate queries for databases that don't fit the relational model. CPL also allows sums (similar to variant records in Pascal, and the datatypes used in most functional languages), making it easy to process data with alternative formats, such as books and journals in a bibliographic database.

A standard technique in functional programming is to apply mathematical laws to transform an elegant but slow program into an efficient equivalent. This technique is applied to good effect in CPL/Kleisli. The standard laws for transforming comprehensions can be viewed as generalizing well-known optimizations for relational algebra. For instance, a CPL query may depend on two relational databases held on different servers. The Kleisli optimizer will transform this into two SQL queries to be sent to the servers (performing as much work as possible locally at the server), and a remaining CPL program at the query site to combine the results. Lazy evaluation and concurrency allow SQL computation at the database sites and CPL processing at the query site to overlap.

CPL/Kleisli also exploits record subtyping. In the example above, Emp represents employees by a set of records. Each record must contain a Name and DNum field, but may contain other fields as well. The type system that permits this flexibility and the technique for implementing it efficiently were both adopted directly from research in the functional community.

1.5 Natural Expert

Every flight through Orly and Roissy airports in Paris is processed by an expert system called Ivanhoe, which generates invoices and explanations for the services used. Ivanhoe is written in Natural Expert, an expert system shell, formerly marketed by the German firm Software AG [18]. Polygram in France controls about one-third of the European market for CDs and cassettes. The Colisage expert system plans packing schedules to minimize empty space and routes to minimize numbers of stops (somewhat like simultaneously solving the Bin Packing and Traveling Salesman problems). Colisage was originally written in a production rule system called GURU, but was ported to Natural Expert when the GURU version proved hard to maintain. Polygram praised the Natural Expert system as shorter and easier to maintain.

Dozens of other applications have been programmed in Natural Expert, including a management support system, a system for assessing bank loans, a tool to plan hospital menus, and a natural-language front end to a database.

Natural Expert integrates an entity-attribute database management system with NEL (Natural Expert Language), a higher-order, statically typed, lazy functional language, roughly similar to Haskell.

One of the selling points of Natural Expert is its user environment. The database is used not only to manipulate user data, but also to store the NEL program itself, which is structured as a number of rules. The database records what rules refer to what other rules, aiding program maintenance. A simple hypertext facility lets the reader jump from use of a rule or attribute to its definition.

The result returned from a database access is typically a list of entity indexes. Lazy evaluation processes entities one at a time, reducing the amount of store required. This is important, because Natural Expert runs on mainframes. Surprisingly, mainframes often provide fewer resources than a personal computer: Natural Expert typically uses only 80K for the heap, and even then some clients complain it is too large.

Traditionally, lazy languages disallow side effects, because the order in which the effects occur would be difficult to predict. NEL, however, permits one use of side effects, a primitive that prints a given question on a terminal and returns the answer typed by the user. Questions are printed in an arbitrary order, but that's no problem for this domain. More importantly, thanks to lazy evaluation, a question is asked at most once, and only if it's relevant to the task at hand. Expert systems people call this "backwards chaining".

Training is key to industrial use of any system. Natural Expert is taught in a one-week course, which includes polymorphic types and higher-order functions. Typically, students grumble about all the compile-time error messages generated by the unfamiliar type system, but are pleased to discover that once a program passes the compiler it often runs correctly on the first try. Nonetheless, clients still point to lack of familiarity with functional languages as a bar to wider acceptance.

Although many of the applications built with Natural Expert are successful and in current use, sales of the system generated insufficient revenue, and Software AG has dropped it as a product.

1.6 Ensemble

Ensemble is a library of protocols that can be used to quickly build distributed applications. Ensemble is in daily use at Cornell to coordinate sharing of keys in a secure network, and to support a distributed CD audio storage and playback service. A number of commercial concerns have begun projects with Ensemble, including BBN, Lockheed Martin, and Microsoft [16].

Ensemble protocol stacks typically have ten or more layers. Highly-layered stacks are flexible, but can be slow. Ensemble regains speed by a series of optimizations. The protocol designer segments the code in each the layer, marking common cases. It is a simple matter (currently performed by hand, but easily automated) to trace which segments execute together, and collect these into optimized *trace handlers*. They also cache information to minimize header size and reorder computations to preserve latency. The result is a win-win architecture, offering both modularity and performance.

Ensemble is written entirely in Objective Caml, a dialect of ML. Ensemble beats the performance of its predecessor, Horus, by a wide margin, even though Horus is written in C. To quote the designers, "The use of ML does mean that our current implementation of Ensemble is somewhat slower than it could be, but this has been more than made up for by the ability to rapidly experiment with structural changes, and thereby increase performance through improved design rather than through long hours of hand-coding the entire system in C" [16].

The designers took care to restrict the use of features of ML they deemed expensive. Higher-order functions are used only in stylized ways that can be compiled efficiently. Exception handling and garbage-collected objects are avoided in the trace handlers. To squeeze the most out of Ensemble, a final step is to translate the trace handlers (which constitute only a small part of the code) into C by hand. This achieves a further improvement of about a factor of two.

A related effort is the Fox Project at Carnegie-Mellon University, which first demonstrated that systems software can be written in functional languages. You can access the FoxNet Web Server at foxnet.cs.cmu.edu. The HTTPD server, the TCP/IP stack, and everything down to the driver protocol is implemented in the Fox variant of Standard ML [3].

Ensemble has produced the fastest product of its kind, while the Fox Project stack runs at speeds varying from 8% faster to 100% slower than commercial implementations. However, this is comparing apples with oranges: Ensemble gains speed by experimenting with new protocols, while Fox measures its achievements against the fixed target of the TCP/IP protocol.

The Fox Project was an important precursor to Ensemble, in that it demonstrated functional languages could be used to build systems programs with reasonable performance. Nonetheless, in my opinion Ensemble marks a more important milestone for functional programming: FoxNet was created by researchers primarily interested in languages, while Ensemble was created by researchers primarily interested in networking.

1.7 Conclusions

So there you have it, six instances of functional languages used *in anger*. Or rather more than six, depending on how you count.

Perhaps some disclaimers are in order. I'm one of the designers of Haskell. Glasgow Haskell is due to my former colleagues, SML/NJ is due to my current colleagues, HOL is largely due to another former colleague, and Pdiff is due to other current colleagues. I consulted for Ericsson on the design of a type system for Erlang. CPL/Kleisli is partly based on my research into comprehensions. So I may be biased.

The list of applications given here is far from exhaustive. I've omitted Microsoft's Fran animation library for Haskell [9], Lufthansa's combination of a simple functional language with partial evaluation to speed up crew scheduling [2], Hewlett Packard's ECDL network control language, the Lolita natural language understanding system, and Mitre's speech recognition system, to name a few. Some of these are listed at a web page for Functional Programming in the Real World [31].

2 Why no one uses functional languages

As we have just seen, to say that no one uses functional languages is an exaggeration. Calls to the European Parliament are routed by programs written in Erlang, virtual CDs are distributed on Cornell's network via Ensemble, real CDs are shipped by Polygram in Europe via Natural Expert.

Still \dots 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?

2.1 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, but make no pretence of completeness.

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. As mentioned above, this is straightforward for strict languages, and recent advances with monads and linear logic has made it possible for lazy languages.

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 [24]. Among other applications, this allows Haskell to be used as a scripting language for Microsoft's Internet Explorer web browser. My colleagues at Lucent are currently applying similar ideas to the SML/NJ compiler.

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 those 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, 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 [10].

Portability and installation. 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. As mentioned above, Lucent researchers would have preferred to build the PRL database language using SML, but chose C++ because SML was not available on the Amdahl mainframe they used.

On the other hand, abstract machines are a popular implementation technique, for both functional languages and for Java, in part because writing an emulator for the machine in C makes it is easy to port to a wide variety of architectures. The Hugs interpreter for Haskell is written in C, and fairly easy to port.

Even when a functional language has been ported to the machine and operating system at hand, it may not be easy to install. While Hugs is easy to install (one mouse click on my Windows box), installing the Glasgow Haskell compiler is something of an adventure (I've so far failed to install it on my local Irix machine).

Availability. 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, 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 [17].

Footprint. 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 stand-alone application program, that can be invoked directly without the intervention of a read-eval-print loop. 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 stand-alone 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 even 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 (though once found it may be more elegant). And while many problems do have efficient functional solutions, there remain some tough nuts for which the best known solutions are imperative in style. Some functional languages lessen these problems by providing an escape to imperative style: SML includes updateable references as a basic data type, and Haskell provides them via monads [20].

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 [18].

Popularity. If a manager chooses to use a functional language for a project and the project fails, then he or she is out on a limb with little support. 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.

Though 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.

No less than managers, employees, too, have their worries. Experience with C++ or Java will buff up your resume nicely, while Haskell or SML will do you little good. Recall, for instance, that no developer could be found to maintain Lucent's Pdiff system, written in SML, so a physicist looking to switch fields was hired (as mentioned in Section 1.3).

2.2 Non-reasons

Having listed many good reasons why people avoid functional languages, let me now rebut two pieces of common cant as to why people don't use functional languages to which I do not subscribe.

Performance. A decade ago people might have reasonably rejected functional languages for poor performance, but these days the performance of functional languages is often within the same ballpark as C. Performance can vary widely, but for the symbolic manipulation to which functional languages are well suited, a rough estimate of within a factor of two of C seems fair.

More important, experience shows that performance that rivals C is not a requirement for success. Java has become enormously successful with performance significantly short of C. Tcl/Tk, Perl, and Visual Basic all rose to prominence with implementations that are interpreted. In the functional world, Erlang achieved its success as an interpreted language, and the Hugs interpreter for Haskell is more widely used than the Glasgow Haskell compiler.

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 objectoriented 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. [19]

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.

2.3 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.

Each of the "angry half-dozen" exploits some strength of functional languages. 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. 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. To aid a paradigm shift, a field may set out new criteria for judging work in new areas.

As I write, Gopal Gupta is organizing PADL 99, the First International Conference on Practical Aspects of Declarative Languages [14]. And Simon Peyton Jones and myself have just completed an editorial for the *Journal of Functional Programming* welcoming papers on functional programming practice and experience, and setting out the criteria we apply to judge them [25]. 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 [27], 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.

3 Functional and logic programming

There are many overlaps between logic programming and functional programming, and some intriguing differences.

I argued above that a "killer app" is essential for success. While functional languages have had moderate successes in a few areas, they have only become perceived as the language of choice in one fairly narrow area: theorem provers. On the other hand, logic programming languages have three "killer apps" where they have achieved widespread success: deductive databases, artificial intelligence, and constraint programming.

Arguably, this is why one can point to hundreds of industrial applications of logic programming (I saw one list with nearly one thousand entries), whereas equivalent applications of functional programming number in the dozens.

Unfortunately, in both the functional and logic camps, rather than play to our strengths we try to be all things to all people. With the exception of the original ML, which aimed squarely at theorem proving (and where its descendants have dominated that area), relatively few functional or logic languages aim at conquering one application area. Instead, advocates promote functional and logic languages as general purpose, good for whatever ails you. Perhaps a more specific emollient is in order. Designers of logic languages might have an easier job filling this prescription than those of functional languages, as logic languages already have a better track record with "killer apps".

The name *declarative* was coined to cover both functional and logic programming, recognizing that they have much in common. Despite this, the two communities meet together rarely, having separate conferences and journals. One area of overlap is the attempt to build languages that combine the features of functional and logic languages. Most of the attempts in this direction seem to come from members of the logic programming community. There have been some technically strong achievements in this area, but as yet no "killer app" has emerged.

As mentioned above, Gopal Gupta is organizing PADL 99, the First International Conference on Practical Aspects of Declarative Languages. This aims to draw together two communities that should interact more, and to focus their attention on applications — a double win!

Acknowledgements. My thanks to Brian Kernighan, Doug McIlroy, and Jon Riecke for comments on earlier versions of this paper, and to all those who listened to and commented on my talks on this topic. Special thanks to the organizers, Krzysztof Apt, David Warren, Vitek Marek and Mirek Truszczyński, and the participants of the workshop on the Logic Programming Paradigm held in Shakertown, Kentucky, a great group from whom I learned a great deal.

References

- 1. Joe Armstrong. The development of Erlang. ACM SIGPLAN International Conference on Functional Programming, June 1997; SIGPLAN Notices 32(8):196–203, August 1997. Also see the Erlang page:
 - http://www.erlang.se
- Lennart Augustsson. Partial evaluation in aircraft crew planning. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, June 1997; SIGPLAN Notices 32(12):127–136, December 1997.
- 3. Edoardo Biagioni, Robert Harper, Peter Lee, and Brian G. Milnes. Signatures for a network protocol stack: A systems application of Standard ML. *ACM Conference* on Lisp and Functional Programming, 1994. Also see the Fox Project page: http://foxnet.cs.cmu.edu
- 4. P. Buneman, S. B. Davidson, K. Hart, C. Overton, and L. Wong. A Data Transformation System for Biological Data Sources. *Proceedings of 21st International Conference on Very Large Data Bases*, Zurich, Switzerland, September 1995. Also see the Kleisli page:
 - http://sdmc.iss.nus.sg/kleisli/MoreInfo.html
- 5. P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension Syntax. *ACM SIGMOD Record* 23(1):87-96, March 1994. (Invited paper.)
- Albert J. Camilleri. A hybrid approach to verifying liveness in a symmetric multiprocessor. 10'th International Conference on Theorem Proving in Higher-Order Logics, Elsa Gunter and Amy Felty, editors, Murray Hill, New Jersey, August 1997. Lecture Notes in Computer Science 1275, Springer Verlag, 1997.
- Sandra Corrico, Bryan Ewbank, Tim Griffin, John Meale, and Howard Trickey. A tool for developing safe and efficient database transactions. XV International Switching Symposium of the World Telecommunications Congress, pages 173–177, April 1995.
- Robert J. Robbins, Editor. Report of the Invitational DOE Workshop on Genome Informatics, 26–27 April 1993.

http://www.bis.med.jhmi.edu/Dan/DOE/whitepaper/contents.html

- Conal Elliot and Paul Hudak. Functional reactive animation. ACM SIGPLAN International Conference on Functional Programming, June 1997; SIGPLAN Notices 32(8):196–203, August 1997.
- 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/

- T. Griffin and H. Trickey, Integrity Maintenance in a Telecommunications Switch, *IEEE Data Engineering Bulletin*, Special Issue on Database Constraint Management, 17(2): 43–46, 1994.
- 12. M. J. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher-order logic*. Cambridge University Press, 1993. Also see the HOL page:

http://www.dcs.glasgow.ac.uk/~tfm/fmt/hol.html

- M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*. Lecture Notes in Computer Science, Vol. 78, Springer-Verlag, 1979.
- 14. Gopal Gupta, chair, First International Conference on Practical Aspects of Declarative Languages.

http://www.cs.nsmsu.edu/~complog/conferences/pad199/

- 15. Pieter Hartel, et al. Benchmarking implementations of functional languages with 'Pseudoknot', a float-intensive benchmark. Journal of Functional Programming, 6(4):621–656, July 1996.
- 16. Mark Hayden and Robbert vanRenesse. Optimizing Layered Communication Protocols. Symposium on High Performance Distributed Computing, Portland, Oregon, August 1997. Also see the Ensemble page: http://simon.cs.cornell.edu/Info/Projects/Ensemble/
- 17. J. Hughes and S. Peyton Jones, editors, Standard Haskell.
- www.cs.chalmers.se/~rjmh/Haskell/
 18. 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.
- 19. S. Alexander Jacobson alex@i2x.com, letter to Haskell mailing list, 3 May 1998.
- J. Launchbury and S. L. Peyton Jones, Lazy functional state threads. In ACM Conference on Programming Language Design and Implementation, Orlando, Florida, 1994.
- M. A. Ozols, K. A. Eastaughffe, and A. Cant. DOVE: Design Oriented Verification and Evaluation. *Proceedings of AMAST 97*, M. Johnson, editor, Sydney, Australia. Lecture Notes in Computer Science 1349, Springer Verlag, 1997.
- 22. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994. Also see the Isabelle page:

http://www.cl.cam.ac.uk/Research/HVG/Isabelle/

- 23. Rinus Plasmeijer and Marko van Eekelen, Pure and efficient functional programming using the "unique" features of Clean. ACM SIGPLAN Notices, to appear.
- 24. Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM components in Haskell. *IEEE Fifth International Conference on Software Reuse*, Vancouver, BC, June 1998.

http://www.haskell.org/active/activehaskell.html

25. Simon Peyton Jones and Philip Wadler. Editorial: Practice and experience papers, Journal of Functional Programming, May 1998. Also see the JFP page: http://www.dcs.glasgow.ac.uk/jfp/

- P. Sansom and S. Peyton Jones, Formally-based profiling for higher-order functional languages, ACM Transactions on Programming Languages and Systems, 19(1), January 1997.
- 27. Andrew Tolmach and Andrew Appel, A Debugger for Standard ML. Journal of Functional Programming, 5(2):155–200, April 1995.
- Philip Wadler. How to declare an imperative. ACM Computing Surveys, 29(3):240– 263, September 1997.
- 29. Philip Wadler, An angry half-dozen, *ACM SIGPLAN Notices* 33(2):25-30, February 1998. [N.B. Table of contents on the cover of this issue is wrong.]
- Philip Wadler, Why no one uses functional languages, ACM SIGPLAN Notices 33, 1998.
- 31. Philip Wadler. Functional programming in the real world. http://www.cs.bell-labs.com/~wadler/realword/