Functional Programming: An angry half-dozen*

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

January 8, 1998

"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?"

Having read the title of this column, 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.

0 Compilers

This one's a freebie. I won't count it toward the six, as it is the obvious and incestuous example of functional languages used in anger.

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. The British firms Abstract Hardware Limited and Harlequin both market commercial ML compilers, each bootstrapped in ML.

In some corners, functional languages bear a reputation for gross inefficiency, but this reputation is out of date.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 [12].

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. This is straightfor-

*This will run as the first of a new series of columns on functional programming in *SIGPLAN Notices*, February 1998.

ward in a strict language, but figuring out how to integrate such side effects into a lazy language has been one of the key advances of recent years. Profiling systems for functional languages have also improved vastly over the last few years, and the usual code-measure-improve cycle is now routinely applied to improve the time and space behaviour of functional programs. However, there are still few good debuggers for functional languages.

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 theorem-proving 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 Organisation, 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) [15].

Both HOL and Isabelle are implemented in Standard ML. Standard ML is a descendant of ML, the metalanguage of the groundbreaking LCF theorem prover, which is in turn an ancestor of both HOL and Isabelle. This circle reflects the intertwined history of theorem provers and functional languages [10, 16, 11].

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 nesting of functions that return functions that accept functions. Further, the type discipline ensured soundness, since the only way to create a value of type *Theorem* was by applying a given set of functions, each corresponding to an inference rule. The type system 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.

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 the first of a range of Ericsson products implemented using Erlang, a functional language designed by Ericsson for telecommunications applications [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 currenly 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 minimise 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, serialise 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-calls have been a mainstay of functional languages, and they are 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: just 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 coming along later.

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 the ASN.1 interface compiler and the Mnesia real-time distributed database, both implemented entirely in Erlang.

3 Pdiff

If you've ever made a 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, features such as call waiting, rates, network topology, and so on. The database is complex, containing nearly a thousand relations. There are tens of thousands of consistency constraints (also called *population rules*) that the data must satisfy [7].

As new features are added to the switch, new transactions are required to update the corresponding data, say 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 constaints 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.

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 points out some of the problems of using a functional language in practice. The 5ESS team considered using Standard ML to write the PRL compiler, but since Standard ML wasn't available for their machine (an Amdahl), they used C++ instead. 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.

4 CPL/Kleisli

In April 1993, a workshop organised 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 }
```

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.

A standard technique in functional programming is to apply mathematical laws to transform an elegant but inefficient 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 generalising well-known optimisations for relational algebra. For instance, a CPL query may depend on two relational databases held on different servers. The Kleisli optimiser 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.

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 [14].

Polygram in France controls about one-third of the European market for CDs and cassettes. The Colisage expert system plans packing schedules to minimise empty space and routes to minimise 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. One might expect a mainframe to provide more resources than a personal computer, but 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 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.

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 [13].

Ensemble protocol stacks typically have ten or more layers. Highly-layered stacks are flexible, but can be inefficient. Ensemble avoids these inefficiencies by a series of optimisations. 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 optimised *trace handlers*. They also cache information to minimise 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."

The designers took care to restrict the use of features of ML they deemed expensive. Higher-order functions are used only in stylised 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].

While the Fox Project has done excellent work, Ensemble makes a more convincing case for functional programming: FoxNet was created by researchers primarily interested in languages, while Ensemble was created by researchers primarily interested in networking.

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 [17]. If you know of other applications that belong, please do write.

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`

[2] 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 and the twelve queries:
`http://sdmc.iss.nus.sg/kleisli/`
`MoreInfo.html,misc/doe-queries.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.)

[6] 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.

[7] 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.

[8] 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`

[9] 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.

[10] 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`

[11] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*. Lecture Notes in Computer Science, Vol. 78, Springer-Verlag, 1979.

[12] 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.

[13] Mark Hayden and Robbert vanRenesse. Optimizing Layered Communication Protocols. *Symposium on High Performance Distributed Computing*, Portlan, Oregon, August 1997. Also see the Ensemble page:
`http://simon.cs.cornell.edu/Info/`
`Projects/Ensemble/`

[14] 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.

[15] 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.

[16] 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/`

[17] Philip Wadler. Functional programming in the real world.
`http://www.cs.bell-labs.com/~wadler/`
`realword/`