Experience Report: Haskell in the "Real World"

Writing a Commercial Application in a Lazy Functional Language

Curt J. Sampson
Starling Software, Tokyo, Japan
cjs@starling-software.com

Abstract

I describe the initial attempt of experienced business software developers with minimal functional programming background to write a non-trivial, business-critical application entirely in Haskell. Some parts of the application domain are well suited to a mathematically-oriented language; others are more typically done in languages such as C++.

I discuss the advantages and difficulties of Haskell in these circumstances, with a particular focus on issues that commercial developers find important but that may receive less attention from the academic community.

I conclude that, while academic implementations of "advanced" programming languages arguably may lag somewhat behind implementations of commercial languages in certain ways important to businesses, this appears relatively easy to fix, and that the other advantages that they offer make them a good, albeit long-term, investment for companies where effective IT implementation can offer a crucial advantage to success.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.3 [*General*]: Coding Tools and Techniques; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) Languages

General Terms Experimentation, Human Factors, Languages, Performance

Keywords functional programming, Haskell, commercial programming, financial systems

1. Introduction

1.1 Background and Beginnings

We[1] are reasonably smart but not exceptional programmers with several decades of professional experience amongst us. Our main working languages were C in the 1990s, Java in the very late 1990s and the first part of the 2000s, and Ruby(7) after that. Other developers on the team have experience in similar languages (such as C# and Python). When we embarked on this project, we had no significant functional programming experience.

In the mid-2000s a fellow programmer extolled to me the virtues of Scheme and, following up on this, I found that people such as Paul Graham were also making convincing arguments (in essays such as "Beating the Averages" [2](3)) that functional programming would increase productivity and reduce errors. Working through *The Little Schemer*(1) I was impressed by the concision and adaptability of what was admittedly toy code. Upon founding a software development company soon after this, my co-founder and I agreed that pursuing functional programming was likely to be beneficial, and thus we should learn a functional language and develop a software project in it. There things sat for about two years, until an opportunity arose.

In the spring of 2008 we were approached by a potential client to write an financial application. He had previous experience working on a similar application in Java, and approached us with the idea of doing something similar.

1.2 Selling the Client

We felt that the client's application domain could benefit from using a more powerful language, and we saw this as an opportunity to explore functional programming. Convincing the client to do this took some work, especially since we had no real experience with functional languages at that point. We took a two-pronged approach to selling functional programming for this job.

First, we explained that we had significant programming experience in more traditional languages, especially Java. We pointed out that this made us familiar with the limitations of these languages: so familiar, in fact, that in the case of Java we had already moved away from the language to escape those very limitations. We explained that, for us, switching languages every few years was a normal technology upgrade (much as any other industry will incorporate new advancements in the state of the art) and we had previous experience with working through these sorts of changes.

Second, we argued that we had been for some time looking to move on to a new language, had done significant research in that direction, and had already identified many of the specific things needed to make the change successful and profitable. As well as general promises of better productivity and fewer bugs, we pointed to specific features we wanted in a new language for which the client himself was looking. In particular, he had expressed a preference for a system where he would have the ability to examine and modify the financial algorithms himself; we explained and demonstrated that the languages we were looking at offered much

[1] Despite this paper having one author, I am reporting on the combined experiences of a multi-person developer and customer team. Thus, I use "we" throughout, except when relating distinctly personal experiences.

[2] Particularly fascinating to us was, "In business, there is nothing more valuable than a technical advantage your competitors don't understand." We are still considering the importance of ourselves understanding the technical advantages we've gained.

ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.

more concision and a more mathematical notation than Java, and showed how this would enable him to more easily participate in the programming process.

2. Selection of Language and Tools

2.1 Desired Language Features

The world of functional programming offers a wide, even bewildering variety of features and technologies, spread across many different languages. In some cases a language has has strong support for an imperative or object-oriented coding style, with merely the possibility of doing functional programming (such as JavaScript); in other cases one must abandon wholesale one's non-functional style and adopt a completely different one.

As we had already extensive experience with Java and Ruby and had become unhappy with the level of expressive power that either offered, we chose to pursue what we felt were more "advanced" language features, at the cost of an increased learning curve and greater risk.

Functional language features of particular interest to us were a sophisticated type system (Hindley-Milner is the obvious example here), and advanced, modular structures for control flow. More generally, we were looking for minimal syntax, concise yet expressive code, powerful abstractions, and facilities that would allow us easily to use parallelism.

The "concise yet expressive code" requirement was particularly important to us, both because we've found that reading and understanding existing code is a substantial part of the programming process, and because we wanted to be able to develop code that, at least for the parts in his domain, our client could read and perhaps modify.

2.2 Considerations of Commercial Software Developers

As commercial software developers, there were also several other factors influencing our selection of tools.

A primary criterion is reliability: bugs in the toolchain or runtime system can derail a project irrecoverably, and may not appear until well into development. Good support can help to mitigate these problems, but that still provides no guarantee that any particular issue can be solved in a timely manner. Generally, widespread use on a particular platform provides the best hope that most serious problems will have already been encountered and debugged. For developers with sufficient technical knowledge, having the source for the system can also provide the ability to debug problems in more detail than the systems supporters are perhaps willing to do, increasing the chances of finding a solution, even if the solution is implemented by someone else.

The compiler or interpreter is only one part of a developer's environment; just as important are the methods of code storage (flat files or otherwise), source code control systems, editors, build tools (such as `make`), testing frameworks, profilers, and other tools, many of these often home-grown. These represent a substantial investment in both learning and development time, and are usually important to the success of a project.

Thus, being able to use a new language implementation with existing and familiar tools is a huge benefit. If the implementation is very nearly a drop-in replacement for a language implementation already in use (as with GHC or Objective Caml for GCC in typical Unix development projects, or F# in .NET environments), much of the previously-built development environment can be reused. If, on the other hand, everything needs to be replaced (as with some Smalltalk environments, or Lisp machines), this imposes a tremendously increased burden.

For many commercial developers, integration with other code running on the platform (such as C libraries) and complete access to the platform facilities themselves (system calls and so on) is important. At times being able to do things such as set operating-system-specific socket options for individual network connections can make the difference between an application working well or poorly. In some cases, applications need to be able to control memory layout and alignment in order to marshal specific data structures used by the system or external libraries.

In our case performance was an important consideration. The application reads and parses market data messages that arrive on a network connection and responds with orders. The response must occur within a few dozen milliseconds for the system to work reasonably well, and the faster the response, the greater the chance the system's algorithms will be able to make profitable trades.

Finally, though this is not true of all commercial developers, we prefer to use a free, open source implementation of a language, rather than commercial tools. There are two reasons for this beyond the purchase cost. First, we've found that commercial support for proprietary software is not significantly better, and is often worse, than community support for open source software under active development. Second, the access to source code and ability to rebuild it can be extremely helpful when it comes to debugging problems (the burden of which usually falls on the customer). Third, clients often feel more comfortable with open source when using niche products (as most functional language platforms are) as it ensures that they can have continued access to the tools needed to build their system, as well as the system itself.

2.3 Languages Examined

With the above criteria in mind, we looked at some of the more popular free functional language implementations available. Our ability to compare these was limited: we simply didn't know enough about functional languages in general, and had no extensive experience with any functional language implementation. Thus, we were forced to rely on what we could learn from textbooks and information on the web. (Blog entries documenting interesting features of the various languages were particularly influential.) Especially, we had no way of judging some of the more esoteric and complex language features as we simply didn't understand them.

This no doubt skewed our evaluation process, but we saw no reasonable way of dealing with this other than spending several months building a non-trivial test application in each of several different languages. This point should be kept in mind by language promoters: the information you need to communicate to convince non-functional-programmers to try a language is of a substantially different sort than what sways those who are already familiar with at least one functional programming language.

We looked most seriously at the Glasgow Haskell Compiler(2), Objective Caml(6), and Scala(8). (We considered looking at LISP and Scheme implementations, but these seemed to be both lacking in certain language features and we had (what are in retrospect perhaps undeserved) concerns about the available free implementations.)

All three of these language implementations we examined share some features in common:

- expressive static type systems;
- concise code, and syntax suited to building embedded domain-specific languages without macros;
- compilers known to be fairly reliable and produce reasonably fast code;
- tools that fit with our current Unix development systems;
- the ability to interface with code compiled from other languages; and
- an open source implementation under active development.

2.3.1 The Glasgow Haskell Compiler

Haskell was the language that most appealed to us; it seemed to have the most advanced set of features of any language out there. Especially, being able easily to guarantee the purity of parts of the code promised to make testing easier, which is an important point for us as we rely heavily on automated testing.

The other significantly different language feature of Haskell, lazy evaluation, had mild appeal for potential performance benefits, but we really had no idea what the true ramifications of lazy evaluation were.

The extensive use of monads was interesting to us, but, as with lazy evaluation, we didn't know enough about it to have any sort of informed opinion. We simply believed what we'd read that monads were a good thing.

There were things about Haskell we found easier to understand. Type classes we found relatively simple, and they seemed to offer more flexibility than the standard object-oriented inheritance-based approach. Two things we could understand very well about Haskell were that it was relatively popular, with a strong community, and that the Foreign Function Interface offered us an escape into other languages should we encounter any overwhelming problems, performance or otherwise.

The books available for learning Haskell, though at times mystifying, seemed to provide the promise of wonderful things. *The Haskell School of Expression*(4), in particular, we found fascinating.

There are several implementations of Haskell available; we chose the Glasgow Haskell Compiler(2) as that is widely recognized as the most mature Haskell compiler available.

2.3.2 Objective Caml

To us, Objective Caml(6) appeared to be a more conservative alternative to GHC, offering many of GHC's and Haskell's features, but without being so radical as to introduce enforced purity or lazy evaluation.

Advantages particular to the implementation were that it was well known, mature, reputedly produced very fast code, and we knew it to be used for large amounts of production code by other companies in the financial community.

OCaml did have several things that put us off. The language syntax didn't seem nearly as clean as Haskell, which was partially inherent (such as double-semicolons to end declarations) and partly due to the inability to overload functions (due to the lack of type classes).

As well, the books we had available were just not as good. At that time we'd just finished reading a fairly recent book on OCaml that, unlike the Haskell books we'd read, did not show us any impressive new programming techniques but instead appeared to treat the language in a very imperative fashion.

2.3.3 Scala

We considered Scala(8) mainly because we thought we might have to use Java libraries. This turned out not to be the case, and as we otherwise preferred to run native code, we didn't investigate the language very deeply.

2.4 Selection of Haskell

After examining the three options above, we chose the Glasgow Haskell Compiler, version 6.8, as our development platform. We felt that GHC offered the following advantages:

- Haskell appeared to be a more powerful and more interesting language than Objective Caml.
- The Haskell community offered good support and was (and still is) growing. There were several good books available with more due to appear, blog entries describing interesting uses of Haskell and various techniques were plentiful, and the `#haskell` channel on IRC was particularly responsive to questions.
- The compiler, while perhaps at the time not as good as the Objective Caml compiler, appeared to be under more active development.

3. Successes and Advantages

3.1 Concise and Readable Code

After a week or two of adjustment, we found Haskell syntax to be remarkably clean and expressive. An initial test was to work with the client to code some of the mathematical algorithms used by the trading system; many were straightforward translations of the mathematical equations into similar Haskell syntax. This obvious advantage of Haskell proved a particularly good selling point to the client early on in the process, reinforcing the idea that the client, though not a programmer, would be able easily to understand some of the more important domain-related aspects of his application.

Another early area of work was the parser for the market data feed. Initially we used Parsec for this, and were quite impressed by how expressive Haskell and a combinator-based approach can be. However, we soon decided to experiment with writing our own parser, for two reasons. First, Parsec did not allow us to keep certain items of state that we felt we needed for error-checking and recovery purposes. Second, the version of Parsec that we were using at the time used `String` rather than `ByteString`, and some brief profiling experiments showed that the performance advantages of `ByteString` were considerable.[3]

As it turned out, after some study (the "Functional Parsers" chapter of Hutton's *Programming in Haskell*(5) was particularly helpful here), we found that parsers in Haskell are particularly easy to write.

Learning about control structures beyond simple recursion, particularly monadic ones, took considerably more time, but also proved fertile ground for finding ways to improve our code's clarity and concision. After a year or so we are extremely happy with our improved ability (over object-oriented languages, and well beyond just parsers) to build combinators, manage state, and deal with complex control flow through monadic code.

To summarize, though the learning effort for the various techniques available to us ranged from low (for mathematical formulae) to moderately high (monadic control structures), there seemed to be almost no area in which Haskell code was not more clear and considerably more concise than doing the equivalent in Ruby or, particularly, Java.

3.2 The Type System and Unit Tests

It's not unusual, when writing in languages with little compile-time type-checking such as Ruby, for about a third of our code to be unit tests. We had anticipated that we might write fewer unit tests in Haskell, but the extent to which the type system reduced the need for unit tests surprised us. As a comparison, in reasonably complex Ruby application of similar line count (though rather less

[3] `ByteString` is, I think, from an academic point of view a rather trivial optimization of some fairly ordinary routines. But from our point of view it was a huge win, and we are eternally grateful to Don Stewart and Duncan Coutts for writing this library. This is worth considering when designing a new system: you need not immediately supply efficient things for basic commercial needs, but if you can give others the ability (through mechanisms such as the `OverloadedStrings` language extension) to easily bring those in later, you open up more possibilities for success in the commercial arena. Haskell has not been bad in this respect, but there are many times I've wished for improvements such as making `String` a typeclass.

functionality), we have about 5200 lines of production code and just over 1500 lines of unit tests. In contrast, in the version of the trading system as of early 2009, we had over 5600 lines of production code and less than 500 lines of unit tests. The functional tests show similar differences.

Further, in most areas where we used the type system to show program correctness, we came out with much more confidence that the program was indeed correct than if we had used testing. Unlike tests, a good type system and compiler will often force the developer to deal with cases that might have been forgotten in testing, both during the initial development and especially when later modifying the code.

On one last note, the difference in the amount of test code we used was far larger between Ruby and Haskell than between Ruby and Java, though Java, as with Haskell, also offers static type checking. We attribute this to a large difference in expressive power between the type systems of Haskell and Java.

3.3 Speed

With the exception of space leak problems (see next section), speed was never an issue for us. The application is sensitive to how fast it can do the calculations to build a pricing model for the current market, but the code generated by GHC 6.8.3 turned out to be significantly faster than the client's previous Java application. We've to this point seen no significant difference between the current system's performance and what we believe we could achieve in C or C++.

We use multi-core systems, and make extensive use of multi-threaded code for both concurrency and parallelism. This has worked well for us. However, when used for parallelism, with lazy evaluation one has to be careful about in which thread computations are really occurring: i.e., that one is sending a result, and not an unevaluated thunk, to another thread. This has brought up problems and used solutions similar in style to the space leak issues.

We have yet to make extensive use of explicitly parallel computation based on the `par` function. However, it seems clear that if and when we move in to this area, implementation will be considerably simpler than when using a thread-based model. Haskell's "purity by default" model helps greatly here.

3.4 Portability

One pleasant surprise was that, once we'd made a few modifications to our build system, building and running our application under Microsoft Windows was no trouble at all. While not an initial requirement, this was a nice facility to have as it saved our client setting up a separate Unix machine to run the simulator for himself. Eventually, we ended up being able to write in Haskell a small amount of Windows code we'd initially planned to write in C++, which saved us considerable time and effort (see below).

3.5 The Foreign Function Interface

One part of our application involved interfacing with the Microsoft Windows DDE facility in order to transfer hundreds of data values to Excel to be displayed and updated several times per second. We had originally planned to write this this in C++, but in view of the portability we'd discovered, we decided see how much of this we could do in Haskell.

This involved a non-trivial interface with a Microsoft C library. In addition to simple calls into C code, we had to deal with low-level data structures involving machine words that combined bit fields and integers of non-standard bit sizes, special memory allocation and deallocation schemes specific to the library, and callbacks from the library back into our code.

This might well have been the biggest and most pleasant surprise we encountered: GHC's excellent Foreign Function Interface allowed us to code this entirely in Haskell without having to write a single line of C. Further, we were able to make extensive use of the C header files supplied for the library, resorting only minimally to error-prone supplying of type information by hand.

Being able to use the Haskell type checker to write C-like code was particularly enjoyable. Type-checking memory allocations and the like is a well known and impressive technique in the Hindley-Milner type system community, but it wasn't until we'd used it ourselves in anger that we realized that programming this close to the machine could be quite a relaxing thing.

This was a significant change from any other language we've used, and from what we believe we would have needed to do in, say, Objective Caml. Every other foreign interface we've seen would have required us to write at least some code in C and would have provided us with significantly less type safety.

4. Problems and Disadvantages

No development environment or set of tools is without its problems, and GHC turned out to be no exception. However, while the nature of the problems we ran into was different from other systems, the general number and level of difficulty of the problems was not significantly different from any other system, especially taking into account how different the language is.

The largest problems for us were issues with learning the language itself, dealing with lazy evaluation, and the performance issue of space leaks.

4.1 Language Issues

Haskell's syntax is quite simple and consistent, and this is great advantage. However syntax is a relatively small part of learning a language, even in languages where it is significantly more complex. To use a language well, and take good advantage of it, one must also learn its structures, idioms and style. Haskell is well known as a language that's very different from others in this regard, and even after many months of programming in it, we still don't feel we've progressed particularly far in this direction, aside from the use of monads for handling state and control flow. (For example, we use applicative style in very limited ways, and as yet make no use of `Control.Applicative`.)

We found that to use many common Haskell structures, such as functors, monoids and monads, we needed to think in significantly different ways than we did in other languages, even other functional languages such as Scheme. Further, the level of abstract thought required to understand these structures (in great part due to their generality) is noticeably higher than in other languages, and in our experience not in the background of many typical commercial programmers. This is not an insurmountable problem, given appropriate learning materials: we built our first monadic parser from scratch after only a few days of study. But learning these things, especially on one's own, can be pretty rough going.

This is mitigated to some degree by being able to fall back easily to more commonly known structures from other languages. We spent a couple of days as an extended interview working with a programmer unfamiliar with Haskell but with extensive experience in LISP, and while the code we produced made little use of typical Haskell idioms, it was certainly clear and workable.

4.2 Refactoring

One of our reasons for doing extensive automated testing is to support refactoring, a technique which we use extensively.

As compared to Ruby, refactoring "in the small," that is, within a single module or a small change crossing a few modules, was more difficult. The main issue was that where, with a language such as Ruby, one can change a small handful of functions and leave the

rest "broken" while testing the change, in Haskell one must fix every function in the module before even being able to compile. There are certain techniques to help work around this, such as copying the module and removing the code not currently of interest (which is tedious), or minimizing the use of type declarations (which doesn't always help, and brings its own problems), but we didn't find these to be very satisfactory.

We feel that a good refactoring tool could significantly ease this problem, but there seems nothing like this in common use in the Haskell community at the moment.

That said, the major refactorings we do on a regular basis ("major" ones being restructuring that crosses many modules and moves significant amounts of code between them) were less affected by this problem, and did not seem to take significantly longer than implementing the same sort of change in any other language. As well, being able to do more work with the type checker and less with testing increased our confidence that our refactorings had not broken parts of the system.

4.3 Profiling Tools, and Deficiencies Thereof

Due especially to lazy evaluation (of which more is mentioned later), understanding the runtime behaviour of one's application is rather important when working in Haskell. GHC's profiling tools are good—certainly as good as anything in Java—but unfortunately not good enough that we don't have some significant complaints.

First, they are rather subtle: it takes time and some careful examination of the documentation and the output to learn to use them well (though this is very often true of any profiling tool). We have found that, as of mid-2009, there are no good tutorials available for this; frustrating and time-consuming experimentation has been the rule. Better descriptions of the output of the summarization tools (such as `hp2ps`, which generates a graph of heap usage from raw profile data) may not be the only solution to this: providing more documentation of the raw profiling data itself might not only enable developers to build their own profile analysis tools, but provide more insight as to just what's going on inside the runtime system.

One of the major problems, directly connected to lazy evaluation, is that where the work happens is quite a slippery concept. One module may define and apply all of the functions for the work to be done, but in actuality leave only a few thunks in the heap: the real work of the computation occurs in some other place where the data are used. If several other modules use the results of, say, a parser, which module or function actually ends up doing the computational work of the parsing (evaluating the thunks) can change from run to run of the program.

This problem is only exacerbated by doing work in multiple threads. When one hopes to spread the work more or less evenly amongst all the threads in the application, inter-thread communication mechanisms that as happily transfer thunks as they do data are more of a liability than an asset. It's quite possible to have a final consumer of "work" actually doing all of the work that one wanted to have done in several other threads running on different cores.

Naïve approaches to fixing the problem can easily backfire. At one point we made some of our data structures members of the `NFData` class from `Control.Parallel.Strategies` and used the `rnf` function at appropriate points to ensure that evaluation had been fully forced. This turned out to do more harm than good, causing the program's CPU usage shoot through the roof. While we didn't investigate this in detail, we believe that the time spent repeatedly traversing large data structures, even when already mostly evaluated, was the culprit. (This would certainly play havoc with the CPU's cache.)

We would particularly like to have a way at runtime to monitor the CPU usage of each individual Haskell thread within our application no matter which operating system threads it might run on over time. This would allow us to monitor the distribution of work to see where more strictness needs to be applied in order to make the best use of our multi-core machines. This needs to be available for non-profiled builds so that we can monitor how the application runs in the real environment, as well as when it's specially built for profiling. We feel that this would be a great step forward in fulfilling pure functional programming's promise of much easier use of multi-core CPUs than conventional languages.

A last note about a particular problem for us with GHC: profiling builds must run using the non-parallel runtime. For an application that requires more than a full core even for builds not slowed down by profiling, this can at times be entirely useless: the problems that show up in the profile output are due entirely to the lack of CPU power available to run the application within its normal time constraints. We are contemplating modifying our stock exchange simulator and trading system to run at a small fraction of "real-time" speed in order to see if this might provide more realistic profiling results.

4.4 Lazy Evaluation and Space Leaks

Lazy evaluation turned out to have good and bad points in the end. However, it caused, and continues to cause, us pain in two particular areas.

The first is that, when distributing computation amongst multiple co-operating threads, it's important to ensure that data structures passed between threads are evaluated to head normal form when necessary before being used by the receiving thread. Initially we found figuring out how to do this to be difficult; even determining when this is necessary can be a challenge in itself. (One clue is a large amount of heap use connected with communications queues between threads, as the sending thread overruns the receiver's ability to evaluate the thunks it's receiving.) A profiler that did not slow down the program and that could help with this would be highly welcome.

The second is the dreaded "space leak," when unevaluated computations retain earlier versions of data in the heap and cause it to grow to enormous proportions, slowing and eventually halting the program as it consumes all memory and swap space. This proved to be a particular problem in our application, which is essentially several loops (processing messages and doing calculations) executed millions of times over the course of a many-hour run. The profiler that comes with GHC is of more help here, but we have yet to work out a truly satisfactory way of dealing with this.

A particularly common cause of space leaks for us was when using data structures (such as lists and `Data.Map`) that have many small changes over time. As one inserts and removes data, old, no-longer-referenced versions of the data structure are kept around to enable the evaluation of thunks that will create the new version of the data structure. As mentioned previously, naïve approaches to the problem can backfire, and better approaches were not always obvious. For `Data.Map` and similar structures, we resorted to doing an immediate lookup and forced evaluation of data just inserted; this seemed to fix the problem while avoiding the massive performance penalty of doing an `rnf` on the root of the data structure.

Another issue relates to interactive programs. Certain lazy idioms (such as using the `getContents` function) can effectively deadlock such programs, rendering them unresponsive. This is not immediately obvious to programmers new to lazy evaluation, and caused us early on to spend some figuring out what was going on.

Knowing what we know now, we are still hesitant about the value of lazy evaluation; every time we feel we've finally got a good handle on it, another problem seems to crop up. We feel that the subject needs, at the very least, a good book that would both cover all of the issues well and help the reader develop a good intuition for the behaviour of lazily-evaluated systems.

5. Conclusions

Our experience with GHC has shown that, while it has its quirks and issues, these are no worse than those of other commonly-used programming language implementations when used for commercial software development. However, some of the issues are different enough that one should be prepared to spend a little more time learning how to deal with them than when moving between more similar systems.

We found significant advantages to using Haskell, and it's clear to us that there is much more expressive power available of which we've yet to make use. Learning to take advantage of Haskell takes a fair amount of work, but benefits are seen fairly quickly and continuously as one improves.

We were lucky with our choice of Haskell and GHC and, in light of our experience, we would make the same choice again. However, we note that, given our lack of experience with other functional languages and platforms, we cannot really say whether or not it is significantly better than, say, OCaml or Scala.

It is our opinion that, overall, our switch to a functional language in a commercial environment has been successful, and we are convinced we will continue to see further benefit over the long term.

References

[1] Friedman, Daniel P., and Felleisen, Matthias, *The Little Schemer*, Fourth Edition. MIT Press, 1996. ISBN-13: 978-0-262-56099-3.

[2] The Glasgow Haskell Compiler. http://www.haskell.org/ghc/

[3] Graham, Paul, "Beating the Averages", from *Hackers and Painters* O'Reilly, 2004, ISBN-13: 978-0-596-00662-4. Also available from http://paulgraham.com/avg.html

[4] Hudak, Paul, *The Haskell School of Expression: Learning Functional Programming Through Multimedia*. Cambridge University Press, 2000. ISBN-13: 978-0-521-64408-2.

[5] Hutton, Graham, *Programming in Haskell*. Cambridge University Press, 2007. ISBN-13: 978-0-521-69269-4.

[6] Objective Caml. http://caml.inria.fr/ocaml/index.en.html

[7] Ruby Programming Language. http://www.ruby-lang.org/en/

[8] The Scala Programming Language. http://www.scala-lang.org/

[9] Duncan Coutts, Don Stewart and Roman Leshchinskiy, "Rewriting Haskell Strings." http://www.cse.unsw.edu.au/ dons/papers/CSL06.html