Fourth ACM SIGPLAN Workshop on Commercial Users of Functional Programming

Jeremy Gibbons

November 2007

The goal of the *Commercial Users of Functional Programming* series of workshops is "to build a community for users of functional programming languages and technology". The fourth workshop in the series took place in Freiburg, Germany on 4th October 2007, colocated as usual with the *International Conference on Functional Programming*. The workshop is flourishing, having grown from an intimate gathering of 25 people in Snowbird in 2004, through 40 in Tallinn in 2005 and 57 in Portland in 2006, to 104 registered participants (and more than a handful of casual observers) this time.

For the first time this year, the organisers had the encouraging dilemma of receiving more offers of presentations than would fit in the available time. The eventual schedule included an invited talk by Xavier Leroy, eleven contributed presentations, and an energetic concluding discussion. Brief summaries of each appear below.

Industrial uses of Caml: Examples and lessons learned from the smart card industry (Xavier Leroy, INRIA)

Abstract: The first part of this talk will show some examples of uses of Caml in industrial contexts, especially at companies that are part of the Caml consortium. The second part discusses my personal experience at the Trusted Logic start-up company, developing high-security software components for smart cards. While technological limitations prevent running functional languages on such low-resource systems, the development and certification of smart card software present a number of challenges where functional programming can help.

The workshop started with an invited talk by Xavier Leroy, introduced as "the father of OCaml". Leroy's talk consisted of three sections: a collection of examples of industrial uses of Caml; a exploration of lessons learned from the Caml Consortium; and a brief discussion of personal experiences with a start-up company developing high security software components for smart cards.

Industrial uses of Caml

Leroy presented an extensive list of examples of industrial uses of Caml (some of which are also described at `http://caml.inria.fr/about/successes.en.html`):

- the *SLAM* tool from Microsoft for static verification of Windows device drivers, whose authors observed that "The expressiveness of [Caml] and robustness of its implementation provided a great productivity boost";
- IBM's *Migration Toolkit*, for the static analysis and conversion of database schemas;
- the *CellControl* domain-specific language inspired by Esterel for programming assembly-line automata, a component of Dassault Système's *Delmia* environment for computer-aided manufacturing;
- a reimplementation in progress of Intel's *reFLect* language, used for high-level modelling and verification of circuits;
- LexiFi's *Modeling Language for Finance*, a mild extension of OCaml providing a domain-specific language for the formal specification and pricing of financial products;
- in-house code for financial quantitative research developed by Jane St Capital, a Wall Street trading firm;
- XenSource's *Hypervisor* systems administration tools for virtualisation solutions;
- the *Astrée* static analyser based on abstract interpretation, used by Airbus to verify fly-by-wire software for the A340 and A380 aircraft.

Some general trends that may be observed from this list are that: the majority of industrial applications of Caml revolve around programming language technologies, such as domain-specific languages, static analysis, program verification, compilation and interpretation; there are occasional inroads into system programming, where scripting languages are more common; there are some more unconventional uses of Caml in academic projects, particularly involving network protocols.

The Caml Consortium experiment

As a prelude to his discussion of the Caml Consortium, Leroy outlined some of the perceived needs of industrial users, from a language implementer's viewpoint. These included: Windows support, for commercial reasons; Linux/BSD support, to satisfy "prima donna" programmers; x86 64-bit support; a foreign-function interface; for some uses, such as static analysis, execution speed and moderate memory requirements; and stability, stability and more stability.

Regarding the last point, there is the age-old question "what happens if Xavier/ Simon/... gets run over by a bus?" That was a frequently-asked question in the 1990s, but seems to be less of an issue now. Being open-source helps, as does having been around longer than Java. Could some official commitment from a reputable institution help even more?

On the other hand, certain features seem somewhat unsurprisingly to be unimportant to industrial users. GUI toolkits are not an issue, because GUIs tend to be built using more mainstream tools; it seems that different competencies are involved in Caml and GUI development, and companies "don't want to squander their precious Caml expertise on aligning pixels". Rich libraries don't seem to matter in general; presumably companies are happy to develop these in-house. And no-one wants yet another IDE; the applications of interest are usually built using a variety of languages and tools anyway, so consistency of development environment is a lost cause.

This led on to a description of the Caml Consortium, an attempt to (lightly) formalise relationships between the Caml designers and their industrial users. This initiative was inspired by the Python Consortium, itself inspired (especially legally) by the WWW Consortium. The initial goals of the Consortium were to provide a place for serious industrial users to meet and discuss; to raise funds to support a full-time developer working at INRIA, on topics relevant to consortium members; to organise annual meetings to decide on new directions and developments, and especially what this programmer should work on. Low membership fees (of the order of €2,000 per annum) were set, with the expectation that twenty members would pay for the programmer.

The Consortium ended up with only six members (Dassault Aviation, Dassault Systèmes, Intel, LexiFi, Microsoft, and XenSource), which was not enough to support a full-time programmer; but there turned out to be very few requests for specific features to be developed anyway. (The money from membership fees now goes to funding studentships and internships instead.) The most tangible benefit for consortium members were more liberal licensing conditions: non-members have a somewhat restrictive open-source license, whereas members enjoy a "free for all uses" license.

Two particular lessons were learned from the experience. Dual licensing seems to be a good solution, except when it comes to integrating external contributions: INRIA needs to own copyright on every part of Caml, and apparently the status of copyright transfers is unclear in the French legal system. There was little interest in identifying features useful to several members, and sharing the cost of their development. Leroy would definitely repeat the dual licensing exercise, if he were to be in the same situation again; he found it nice to meet consortium members from time to time, but perhaps this could have been achieved through a less formal mechanism.

A quick look at the smart card industry

Leroy had a little time left for a brief description of his company Trusted Logic, founded in 1999 with the intention of becoming an independent provider of software and solutions for smart cards and other secure embedded systems. A smart card is a tiny computer, usable as a security token: with low resource usage, hence inexpensive, but highly secure against hardware and software attacks.

A decade ago, smart cards had closed, proprietary architectures, and programs were developed in C or assembler by manufacturers. Now there are standardised, mostly open architectures such as MultOS and JavaCard, and programming is no longer restricted to manufacturers.

Leroy viewed as misguided the description of the JavaCard environment as a 'subset' of Java. In particular, storage of objects in persistent memory and the lack of garbage collection mean that all the space needed by an application is allocated at installation time. This leads to a programming style that is more like Pascal than Java. This has consequences for the opportunities for use of functional programming in such embedded systems: FP on the card is not currently possible, due to lack of resources; FP on card terminals is technically feasible, but it isn't clear what applications would benefit; but such devices require custom development tools, and this is a classic match for FP.

In conclusion, functional programming generally, and Caml in particular, are quite successful in the area of programs that manipulate programs, such as theorem provers and bootstrapped compilers; this domain is industrially relevant today. It is a niche domain, but an active and technologically important one; we are still looking for other such niches.

The way it ought to work... and sometimes does (Ulf Wiger, Ericsson)

Abstract: The telecommunications world is now moving rapidly towards SIP-based telephony and multimedia. The vision is to merge mobile and fixed networks into one coherent multimedia network. Ericsson has been a pioneer in SIP technology, and the first Erlang-based SIP stack was presented in 1999. A fortunate turn of events allowed us to revive the early SIP experiments, rewrite the software and experiment to find an optimal architecture, and later verify our implementation with great success in international interop events. We believe this to be a superb example of how a small team of experts, armed with advanced programming tools, can see their ideas through, with prototypes, field trials, and later large-scale industrial development.

The scheduled speaker from Ericsson, Sean Hinde, had been summoned to assist customers in India, but Ulf Wiger kindly agreed to stand in. His topic was what he called "a perpetual problem in industry", namely getting the right people to lay the

foundation for a new product: capturing the innovative work done in research labs, often years before it becomes commercially viable, and injecting this knowledge into the "product development machine". Large companies essentially always employ a bell curve of competencies: some people are a waste of space; the majority are competent if given well-defined parameters within which to work; and some could define the whole of a system, given enough time. Architectural decisions are usually taken by those in the middle of the curve, because there are more of them; how can we arrange for those decisions to be made by the best?

Wiger spoke about his colleague Hans Nilsson and SIP (the *Session Initiation Protocol*, a text-based protocol a bit like SMTP for setting up phone calls) as a case study. Nilsson wrote Ericsson's first working SIP stack in 1998, the year before even the RFC was published. He became the SIP expert in the group in 2005, and adapted his SIP stack to the AXD 301/TAG product; then SIP was removed from the product's feature list. Nilsson continued, but now in 'stealth mode', working in secret to maintain SIP expertise within the project, without having a budget or a mandate to do so. He reworked the stack as a reference implementation, and optimised it for readability; this turned out to *increase* its performance by 50% (other developments at Ericsson had apparently made previous optimisations obsolete, and even detrimental).

At that point, the GSM Association wanted to test a new product idea called an *IPX Proxy*. Since officially Nilsson's design unit "didn't do SIP", they were asked last whether they could build one for Ericsson. This was with only four weeks to go before the "interop" (the working meeting at which participating companies try to get their implementations to interoperate); but Nilsson, aided by Joe Armstrong, completed the implementation in time. The interop was successful, and Nilsson could fix bugs in his reference implementation, or make it "bug compatible" with others', in minutes rather than days. The adventure became known at the highest levels of the company, and although it was not an official product, customers started to ask for the Ericsson IPX Proxy, and Nilsson's implementation is now a vital part of a major product at Ericsson.

Wiger summarised the experience as a lucky combination of gut feeling, hard work, talent and coincidence. Given the right tools, the lone expert can dazzle even the largest company, and sometimes even deliver a real product out of it at the end. But they have difficulty in getting the message through that it is the combination of tools and talent that matters; many are more willing to believe in magic instead.

The default case in Haskell: Counterparty credit risk calculation at ABN AMRO
(Cyril Schmidt and Anne-Elisabeth Tran Qui, ABN AMRO)

Abstract: ABN AMRO is an international bank headquartered in Amsterdam. For its investment banking activities it needs to measure the counterparty risk on portfolios of financial derivatives. We will describe the building of a Monte-Carlo simulation engine for calculating the bank's exposure to risks of losses if the counterparty defaults (e.g., in case of bankruptcy). The engine can be used both as an interactive tool for quantitative analysts and as a batch processor for calculating exposures of the bank's financial portfolios. We will review Haskell's strong and weak points for this task, both from a technical and a business point of view, and discuss some of the lessons we learned.

Tran Qui began this presentation with an explanation of the problem. A financial contract can be valued; if the counterparty on the contract defaults (goes bankrupt), the contract terminates. To quantify the effect of this, ABN AMRO needs to estimate the value of the contract under future market conditions. The intention is to assess the bank's total exposure to the collapse of a single counterparty.

Their solution was to build *Exion*, a Monte Carlo simulation tool for the valuation of contracts. It simulates future market scenarios, calculating the value of the contract for each simulation, thereby obtaining a distribution of future values. Exion has a "core plus plug-in" architecture; it provides a language in which to write plug-ins, each of which is developed by quantitative analysts to value a particular financial contract. For technical reasons, they didn't make use of Peyton Jones' financial combinators.

Then Schmidt took over to describe the implementation. Haskell was chosen as the development language, for somewhat subjective reasons rather than as a consequence of a scientific study. He was familiar with Haskell, and his manager had used Gofer, so he had no difficulty in making the case; as far as he knew, Haskell is not used elsewhere within the company.

The strongest point in favour of Haskell for this problem was its relative similarity to mathematical notation. Support for a new kind of contract typically involves translating a formula from a research paper into code, and a narrower gap simplifies debugging. Haskell's expressiveness and conciseness were valued. Strong typing, automatic memory management, and the absence of side-effects mean that once the program compiles, it is likely to do what you want it to.

The weakest point of Haskell was performance: the system was more than four times slower than the equivalent C++ code; since running times are of the order of ten hours, this is significant. Rewriting to improve performance led to ugly code, losing some of Haskell's advantages; eventually they resorted to rewriting

critical parts in C++. In response to a question about scaling up the hardware to address performance, Schmidt reported that new machines are coming, but not before 2Q2008.

Other issues were the scarcity of documentation and resources: there are many more books on C++ than on Haskell, and much more in-house experience, although this situation is improving. Profiling is more difficult than in C++; fortunately debugging is not needed so often, but there is still a psychological difficulty in starting to use a new language when you know there's no graphical debugger for it. Sometimes the tools get out of sync with each other, for example hs-plugins with GHC; Visual Haskell is nice, but they're using a different version of GHC and couldn't get them to work together. But these were all minor issues compared with performance.

To conclude, Schmidt claimed that Haskell improves development productivity. He was not afraid of breaking the rest of the system: less regression testing was needed than with C++. On the other hand, performance needs to be tracked: a small change in code can induce a large change in running time; but bottlenecks can be rewritten in C++. Rapid progress is being made in development environments. Overall, Schmidt was happy that he chose Haskell. Help from the audience was offered in streamlining the performance-critical core, but because the code is proprietary it could not be distributed.

Ct: Channelling NeSL and SISAL in C++ (Anwar Ghuloum, Intel)

Abstract: I will discuss the design of Ct, an API for nested data parallel programming in C++. Ct uses meta-programming and functional language ideas to essentially embed a pure functional programming language in impure and unsafe languages, like C++. I will discuss the evolution of the design into functional programming ideas, how this was received in the corporate world, and how we plan to proliferate the technology in the next year.

Ct is a deterministic parallel programming model integrating the nested data parallelism ideas of Blelloch and bulk synchronous processing ideas of Valiant. That is, data races are not possible in Ct. Moreover, performance in Ct is relatively predictable. At its inception, Ct was conceived as a simple library implementation behind C++ template magic. However, performance issues quickly forced us to consider some form of compilation. Using template programming was highly undesirable for this purpose as it would have been difficult and overly specific to C++ idiosyncrasies. Moreover, once compilation for performance was considered, we began to consider a language semantics that would enable powerful optimizations like calculational fusion, synchronization barrier elimination, and so on. The end result of this deliberation is an API that exposes a value-oriented, purely functional vector processing language. Additional benefits of this approach are numerous,

> including the important ability to co-exist within legacy threading programming models (because of the data isolation inherent in the model). We will show how the model applies to a wide range of important (at least by cycle count) applications. Ct targets both shipping multi-core architectures from Intel as well as future announced architectures.
>
> The corporate reception to this approach has (pleasantly) surprised us. In the desktop and high-performance computing space, where C, C++, Java, and Fortran are the only programming models people talk about, we have made serious inroads into advocating advanced programming language technologies. The desperate need for productive, scalable, and safe programming languages for multi-core architectures has provided an opening for functional, type-safe languages. We will discuss the struggles of multi-core manufacturers (i.e. Intel) and their software vendors that have created this opening.
>
> For Intel, Ct heralds its first serious effort to champion a technology that borrows functional programming technologies from the research community. Though it is a compromise that accommodates the pure in the impure and safe in the unsafe, this is an important opportunity to demonstrate the power of functional programming to the unconverted. We plan to share the technology selectively with partners and collaborators, and will have a fully functional and parallelizing implementation by year's end. At CUFP, we will be prepared to discuss our long term plans in detail.

Ghuloum started his presentation with a description of the current state of the art in processors. In each new generation, linear dimensions shrink by 30%, capacitance shrinks by 30%, maximum voltage decreases by 10%, switching time shrinks by 30%. Consequently, there is a doubling of feature density and 60% increase in power: performance per area is increasing, but performance per watt is decreasing.

Therefore Intel's current strategy, rather than building bigger, more aggressively speculative cores, is to use more relatively simpler cores, riding the increased density wave of Moore's Law while decreasing per-core power. The "free lunch" is over; applications will not automatically see improvements in performance when a processor is upgraded, and might even see regressions.

Parallel hardware is here today; quad-core processors were available in desktop machines nearly a year ago, and multi- and many-core processors are on the way. But parallel computers are much harder to program than sequential ones, so there is a strong interest from software vendors in a parallel programming model that is easy to use, high performance, and portable. Intel's response to this pressure is two-fold. For the near term, they plan to experiment with parallel programming idioms based on existing prevalent languages, to ease the transition; that was the subject of the remainder of this presentation. For the longer term, they are starting to design the languages of the next decade, using the experience gained with the first approach. Ghuloum believes that one or more mainstream languages for parallel computing will come from the programming languages community; "stay tuned".

Intel is experimenting with *Ct*, a C++ library supporting "throughput computing" (GPGPU-style) using data-parallel ideas. The advantages of data parallelism are that: it provides a deterministic programming model, eliminating data races and maintaining invariance of behaviour on increasing cores; it has a predictable performance model; it is highly portable, to both threaded and SIMD architectures; and it is very expressive, especially when application usage patterns are considered.

There is a deliberate policy to avoid language extensions, and to exploit libraries and runtime- and macro magic instead. So Ct is like a library implementation of STL-style containers. It supports both flat data parallelism (vectors, element-wise operations, reductions, some constrained permutations) and nested data parallelism (irregularly nested and sparse or indexed vectors). It includes an embedded functional mini-language as a template meta-programming library. It uses a dynamically linked runtime to generate and optimise code on the fly.

There is a white paper *Ct: A Flexible Parallel Programming Model for Tera-scale Architectures* about Ct at `http://techresearch.intel.com/UserFiles/en-us/Image/TS-docs/whitepapers/CtWhitepaper_061507.pdf`. Anyone wishing to experiment with Ct 1.0 should contact the speaker (`anwar.ghuloum@intel.com`); they have a free license for academic use, but are still working on the terms for commercial use.

Terrorism Response Training in Scheme (Eric Kidd, Dartmouth Medical School)

Abstract: The Interactive Media Lab (IML) builds shrink-wrapped educational software for medical professionals and first responders. We have teams focusing on media production, script-level authoring, and low-level engine development. Our most recent project is *Virtual Terrorism Response Academy*. VTRA uses 3D simulations to teach students about radiological, chemical and biological weapons. Our software is now undergoing trials at government training centers and metropolitan police departments. VTRA consists of approximately 60,000 lines of Scheme, and a similar amount of C++. All of our product-specific code is in Scheme, and we make extensive use of macros and domain-specific languages.

From 1987 to 2002, we used a C++ multimedia engine scripted in 5L, the "Lisp-Like Learning Lab Language". This was Lisp-like in name only; it used a prefix syntax, but didn't even support looping, recursion, or data structures. We needed something better for our next project! We ultimately chose to use Scheme, because (1) it was a well-known, general-purpose programming language, and (2) we could customize it extensively using macros. Migrating to Scheme proved tricky, because we needed to keep releasing products while we were building the new Scheme environment. We began by carefully refactoring our legacy codebase, allowing us to maintain our old and new interpreters in parallel. We then rewrote the front-end in a single, eight-day

> hacking session. But even once the Scheme environment was ready, few of our employees wanted to use it. In an effort to make Scheme programming more accessible, we invested significant effort in building an IDE. Today, our environment is much more popular—a third of our employees use it on a regular basis, including several professional artists.
>
> After migrating to Scheme, we added support for 3D simulations. And Scheme proved its worth almost immediately: we faced several hard technical problems, which we solved by building domain-specific languages using Scheme macros. First, we needed to simulate radiation meters. For this, we used a reactive programming language to implement a Model–View–Controller system. Second, we needed to guide students through the simulation and make teaching points. For this, we relied on a "goal system", which tracks what students need to accomplish and provides hints along the way. In both these cases, Scheme proved to be a significant competitive advantage. Not all problems have clean imperative solutions. A language which supports functional programming, macros, and combinator libraries allows us to do things our competitors can't.
>
> This summer, we'll be releasing our engine as open source, and starting work on a GUI editor. We welcome users and developers!

Kidd reported on a reimplementation exercise at the Interactive Media Laboratory at Dartmouth Medical School. They have a team of fifteen people producing shrink-wrapped software, funded by both grants and sales. The project, the *Virtual terrorism response activity*, was a simulation tool for training operations level personnel (fire, police, emergency medical service), not technical. The leader of the team, Joseph V. Henderson MD, has been researching such "interactive mentoring" for twenty years.

The application consists of a variety of audio and video, and two- and three-dimensional animations. The major challenge is providing tools for constructing animations. An existing version used a simple scripting language with a Lisp-like syntax, but one even more limited than the Bourne Shell for whitespace and metacharacter handling. It consisted of an interleaved parser, interpreter and GUI, and had independent Mac and Windows code bases. The task was to refactor it.

The ad hoc boundaries between aspects of the code were abstracted into interfaces, which were implemented in Scheme — chosen because it was a small syntactic step from what was already there. That didn't help much; the user base remained at a single person. DrScheme was tried, but was felt (in 2003) to be too slow for large programs. Instead, they customised the *Scintilla* text editor to provide name completion, syntax highlighting, and proper indentation; that was enough to encourage more of the team to become users. Some macro hackery was applied to flatten nested `let`s into apparent sequences of steps.

Negative aspects of the experience included: a lack of backtracers, debuggers or profilers; a feeling that functional constructs such as maps and lambdas become

"hairballs" in the code, resistant to adaptation; a suspicion that they were reinventing objects the hard way; the problems with evolving a domain-specific language after it has started being used for 50,000 lines of code; and still no graphical editor. Big wins included: good libraries and documentation; a nice "state database" of key–value pairs; domain-specific languages for goal description, hints, etc; open source, borrowing ideas from Smalltalk and Ruby; and hope for a nice (perhaps monadic) model of reactive objects.

Learning with F#
(Phil Trelford, Microsoft Research)

Abstract: In this talk, I will describe how the Applied Games Group at Microsoft Research Cambridge uses F#. This group consists of seven people, and specializes in the application of statistical machine learning, especially ranking problems. The ranking systems they have developed are used by the XBox Live team to do server-side analysis of game logs, and they recently entered an internal competition to improve "click-through" prediction rates on Microsoft *adCenter*, a multi-million dollar industry for the company. The amount of data analysed by the tools is astounding: e.g. 3TB in one case, with programs running continuously over four weeks of training data and occupying all the physical memory on the 64-bit 16GB machines we use.

F# plays a crucial role in helping the group process this data efficiently and develop smart algorithms that extract essential features from the data and represent the information using the latest statistical technique called "factor graphs". Our use of F# in conjunction with SQL Server 2005 is especially interesting: we use novel compilation techniques to express the primary schema in F# and then use SQL Server as a data slave.

Trelford reported on work in the Applied Games Group at Microsoft Research in Cambridge; this group of seven (two of whom are programmers) use statistical machine learning techniques to solve ranking problems. The basis of the algorithms is a technique known as *factor graphs*, which provide a finer-grained representation of the structure of a probability distribution than do Bayesian nets.

Two particular projects were described. The first of these was the *TrueSkill Rating Problem*. Every XBox 360 game uses *TrueSkill* to match players fairly. The problem is to compute a global ranking of the six million players around the world, on the basis of a million matches per day. Two pieces of related work were to construct a live activity viewer for XBox 360 games (1,400 lines of F# code to parse log file entries and dump them to a database, and another 1,400 lines for the GUI, handling 2GB of logs in under an hour), and to tune the user experience for the recently released *Halo 3* game by simulating matches in advance of the release (1,800 lines of F# provided a multi-threaded histogram viewer and real-time spline

editor, which worked on 52GB of simulation data generated by a distributed C# application using .Net remoting).

The second problem described was the *adCenter Problem*. This was an internal competition to try to predict the probability of clicks on adverts placed on `www.live.com` and `www.msn.com` over a few days of real data, based on results obtained from several weeks of historical training data. The training data consisted of seven billion page impressions, and the training period was two weeks, which meant that the analyser had to process 5,787 impression updates per second, or one every 172 microseconds. The implementation used Excel, SQL Server, and Visual Studio with the interactive, interpreted version of F#. The most significant part of the solution was a *SQL Schema Generator*: 500 lines of code written in two weeks, managing high-performance database updates via computed bulk insertions.

The benefits of F# were felt to be: that it was a language spoken by both developers and researchers in the group; that it yields correct, succinct and highly performant programs; its interoperability with the .Net framework, including all the Visual Studio tools; and the fun it is to program with.

Productivity Gains with Erlang (Jan Henry Nyström, Erlang Training & Consulting)

Abstract: Currently most distributed telecoms software is engineered using low- and mid-level distributed technologies, but there is a drive to use high-level distribution. This talk reports the first systematic comparison of a high-level distributed programming language in the context of substantial commercial products.

The research clearly demonstrates that Erlang is not only a viable, but also a compelling choice when dealing with high availability systems. This is due to the fact that it comparatively easy to construct systems that are:

- resilient: sustaining throughput at extreme loads and automatically recovering when load drops;
- fault tolerant: remaining available despite repeated and multiple failures;
- dynamically reconfigurable: with throughput scaling, near-linearly, when resources are added or removed.

But most importantly these systems can be delivered at a much higher productivity and with more maintainable deployment than current technology. This is attributed to language features such as automatic memory and process management and high-level communication. Furthermore, Erlang interoperates at low cost with conventional technologies, allowing incremental reengineering of large distributed systems.

Nyström reported on the project *High-Level Techniques for Distributed Telecoms Software*, funded by the UK Engineering and Physical Sciences Research Council as a collaboration between Motorola UK Labs and Heriot-Watt University between 2002 and 2006. The aim of the project was "to produce scientific evidence that high-level distributed languages like Erlang or Glasgow Distributed Haskell can improve distributed software robustness and productivity".

The six specific research questions addressed by the project were: Can robust, configurable systems be readily developed? Can productivity and maintainability be improved? Can the required functionality be specified? Can acceptable performance be achieved? What are the costs of interoperating with conventional technology? Is the technology practical?

The research strategy was to reengineer in GdH and Erlang two existing telecommunications applications, written mostly in C++, and to compare the two versions for performance, robustness, productivity, and the impact of programming language constructs. (This talk subsequently discussed only Erlang, not GdH.) The two applications chosen were a *Data Mobility* component, which communicates with Motorola mobile devices, and a *Dispatch Call Controller*, which handles mobile phone calls.

The results reported were that the Erlang reimplementations were a success: improved resilience; less than a third the size of source code; required functionality is readily specified; two to three times as fast, although with increased memory residency; interoperability with C, albeit at a high cost in execution time; and availability on several platforms. No control experiment was performed, however.

An OCaml-based Network Services Platform (Chris Waterson, Liveops)

Abstract: At Liveops, we've developed a robust network services platform that combines the scalability of event-based I/O with the simplicity of thread-based programming. We've done this using functional programming techniques; namely, by using continuation-passing monads to encapsulate computation state and hide the complexity of the non-blocking I/O layer from the application programmer. Application code is written in a "naive threading" style using primitives that simulate blocking I/O operations.

This network platform serves as the basis for one of the most critical applications in our business — agent scheduling — and has proven to be easy to maintain and extremely scalable. Using commodity server hardware, we are able to support thousands of persistent SSL connections on a single dual-core Pentium-class server, and handle tens of thousands of transactions per minute. The application and platform are implemented in OCaml.

This talk will briefly describe the application domain, discuss the specifics of the monadic I/O library we've built, and describe some of the issues involved. Our hope is that by the time that the conference arrives, the library will be released as open-source software.

> Although developed independently, this work is the same vein as (and, in some ways, validates) Peng Li and Steve Zdancewic's *A Language-based Approach to Unifying Events and Threads*, which appears in the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) in June 2007.

Liveops manage media-driven telephone calls, for example, in response to TV commercials. The company offers agents work when it is needed; the agents schedule themselves to book the work, and execute the work from home. Their first-generation system was Ajax-style, LAMP-based, and stateless. Each interaction generated an HTTPS request, and clients polled the schedule for updates. The system fell over at about 1000 users.

These issues with scaling led to a session-based redesign, whereby the client maintains a persistent SSL connection with server. Now there is no need to re-authorise or to reconstruct the state needed to handle requests, and two-way communication allows events to be propagated to client. But how could they scale to millions of users (and hence millions of SSL connections)? Currently the company has 10,000 to 15,000 agents, but they want to grow.

So they had conflicting requirements to reconcile: a manageable programming mode, and scalable implementation. The approach taken was to convert the initial imperative style of code to use continuations, and then express these monadically in OCaml. The approach was encapsulated in a monadic communication library *Mcom*, layering the CPS monad for propagating state over an I/O monad for the basic operations. The result "feels" like naively threaded code over blocking I/O, but runs with scalable I/O primitives (`kqueue`, `epoll`), and ensures that pre-emption occurs at well-known points. Particular aspects of OCaml exploited were: first-class functions to support continuations; currying to support the monadic style; and the production-quality compiler and runtime.

The resulting system seems to scale well. Liveops use two Xeon-class dual-core servers (the second one to enable live software updates), and can support 5,000 simultaneous SSL connections; they can sustain over 700 transactions per second, with bursts of 1,500. There have been two major releases since the initial deployment in mid-2006, and the system consists of 10,000 lines of code. On the other hand, the approach requires non-blocking protocol implementations (they had to write their own non-blocking SQL driver), and the monadic style doesn't mix well with externally-raised exceptions.

Liveops got into functional programming in the first place through synchronicity: a friend gave the speaker some of Wadler's papers to read at just the right time. Overall, functional programming techniques allowed the company "to have our cake and eat it too"; they hope to release the code as open source soon, once legal issues have been resolved.

Using Functional Techniques to Program a Network Processor
(Lal George, Network Speed Technologies)

Abstract: I will describe technology we built at Network Speed Technologies to program the Intel IXP network processor — a multi-core, multi-threaded, high performance network device.

Ideas from functional programming and language design were key to programming this device. For example, 650 lines of Intel C for the IXP together with embedded assembly are required to just update the TTL and checksum fields of an IPv4 header; in our functional language, it is less than 40 lines!

The functional semantics and novel compilation technology enables us to demonstrably out-perform hand-coded assembly written by experts — a remarkable accomplishment by itself, and even more so in the embedded space. The critical components of the compilation technology are a big part of the puzzle, but are not directly FP related.

The language and technology have been a phenomenal success, and easily surpass conventional approaches. The ease of learning, the dramatically lower cost, and superior performance make this the 'right' choice for deploying these devices. However, there are hard lessons learned from using FPL in the real world...

George's talk was about the functional programming techniques that his company Network Speed Technologies exploited for programming embedded devices, and in particular the Intel IXP network processor. Such network processing units have demanding performance requirements: a packet arrives every 66 cycles, to they have to be multi-core; and with no cache, they have to be multi-threaded too.

Programming such devices is complicated. Intel have provided a modified C compiler for the purpose, with over a hundred specialised intrinsics; but these require register choices to be made early, reducing flexibility and increasing maintenance costs. George gave the example of decrementing the time-to-live field of an IPv4 header. The compound read–modify–write nature of the operation, and the variable alignment of this field within the header, mean that 650 lines of micro-engine C with embedded assembler are needed.

Network Speed Technologies have produced a language μL and corresponding compiler μC specifically designed for IXP applications, which they claim to be "remarkably small, simple and complete, easier to learn, enhancing reliability and robustness, and generating guaranteed optimal code".

The language provides a 'register-centric' view of the processor architecture, hiding the non-orthogonality whereby different register banks are connected to different resources. Data structures are mapped via a 'layout declaration'; unpacking with respect to a layout yields the fields of the data structure, irrespective of alignment, and higher-order features allow the thread-safe write-back process to be returned too.

The compiled code generated by μC, using integer linear programming, apparently beats hand-coded assembly.

Pragmatic issues that arose during the exercise included those of syntax, programmer familiarity with functional concepts (such as lexical scoping, higher-order functions and anonymous lambdas), plug-in capability, and the availability of textbooks. Unfortunately, the company was not a success: they did have customers, but then the market crashed.

Impediments to Wide-Spread Adoption of Functional Languages (Noel Welsh, Untyped)

Abstract: If functional languages are so great, why does virtually no-one use them? More to the point, why have relatively new languages like PHP, Python, and Ruby prospered while functional languages have failed to make inroads in their long and glorious history? I believe the answers are largely cultural, and indeed the academic culture of functional languages is both their greatest strength and biggest barrier to adoption. I'll present a simple model of language adoption, and show specific instances where functional languages fail to support it. I'll also make concrete suggestions for how functional language communities can improve, while still retaining their distinctive strengths.

Welsh's talk was about the lack of commercial success of functional programming: reasons why, and what could be done to foster adoption of functional programming.

Certainly, if you look at the shelves in a computer bookstore, or the skills listed on `monster.com`, it appears that — to a first approximation — no-one uses functional languages. Do we care? We should. Industry benefits from functional programming research (as the Experience Reports at ICFP show), and interesting research (such as continuation-based web servers) has arisen from addressing practical problems.

Welsh's diagnosis was that the essential barrier to the adoption of functional languages was one of risk versus return. Most companies see the potential return from a change of programming language quite small; but the risk of the unknown (associated with training developers, maintaining code, and using an unproven implementation) seems high.

The solution is not to start looking for risk-loving programmers — next year's ICFP location notwithstanding. Rather, we should look for people for whom the risk versus return equation is more favourable: startups, and expert programmers. These people will naturally be interested in new languages; we just have to make it easier for them to gain proficiency.

Welsh presented a simple 'pipeline' model of language acquisition. We start with a universe of *potential programmers* who have heard about and are interested in a language. Of these, some proportion will actually try the language, becoming

novices. Those who don't drop out will become the *hackers*, who make up the majority of the programming community. Finally, some of the hackers who don't get bored and leave will become the *gurus* of the community. The challenge is to minimise attrition along this pipeline. To see how to do that, he considered the needs of each group.

What are the needs of the potential programmer? People program to solve problems; so there had better be a clear statement of what kinds of problem the language is good for. The Python community does a good job of this on `python.org`: "Python is a dynamic object-oriented programming language that can be used for many kinds of software development. It offers strong support for integration with other languages and tools, comes with extensive standard libraries, and can be learned in a few days." Compare this with the equivalent from `haskell.org`: "Haskell is a general purpose, purely functional programming language featuring static typing, higher-order functions, polymorphism, type classes, and monadic effects. Haskell compilers are freely available for almost any computer." If you understand all that, you don't need to be here: you're already a Haskell programmer.

The needs of the novice are fairly well covered. After all, academics have an incentive to write first-year textbooks, and there are many good ones available. They may not be ideal for competent programmers, but they are good enough. Similarly, the gurus are well served by events such as ICFP; they will take care of themselves. It's the hackers — those in the middle of the pipeline — who are currently poorly served. Academics have little incentive to produce material for them, and there are too few other skilled practitioners to carry the load. What can be done to help them?

A tried-and-tested model, used by the Perl, Python and Ruby communities, is to set up a non-profit organisation to coordinate community activities. It doesn't have to do much; all that is needed is for it to have the authority within the community to be seen as definitive. In particular, this organisation should encourage a culture of documentation. Good tools are already available; things will grow incrementally. Similarly, a central repository of libraries is needed, like `planet.scheme.org`: particularly if it provides some kind of automated package management system.

Conferences are important: for meeting up, learning from each other, and generally feeling like a community. But better is worse: academic conferences are intimidating for the less skilled, and originality too high a barrier for getting their own contributions heard. The Ruby community runs about a dozen events a year: mostly small one-day affairs, at which the barrier to entry is low, and the talks are just as likely to be about programming process, or tutorials, or community issues, as about recent advances. More initiatives such as the Bay Area Functional Programmers and London Haskell Users' Group are needed. (It is important for user groups not to be too ambitious; it is tempting to start with big names, but this is difficult to sustain, and discourages subsequent speakers.)

In summary, the suggestions mostly revolved around coordinating existing efforts, and providing a little infrastructure. This is not onerous, and provides an opportunity for enthusiastic non-experts to contribute. Setting up a non-profit organisation is cheap and easy.

Functional Programming in Communications Security
(Timo Lilja, SSH Communications Security)

Abstract: At SSH Communications Security, we've employed functional programming for a long time in some of our projects. Over the years, we've shipped a number of products written mostly in Scheme, and are about to ship some software which is in part written in Standard ML. We have also written several pieces of software for internal use in Haskell, Standard ML, Scheme, and probably others as well.

In this talk, I will describe some useful insights into how these languages have worked for us in developing security software. We had some successes: we've been able to build and ship fairly large software systems rather quickly and with good confidence in certain aspects of their security. We've also experienced some failures. Using functional programming doesn't protect against bad design. Implementations of functional languages are sometimes slow. The user base of many languages is small, and there aren't a whole lot of programmers on the market who can program well in, for example, Scheme. Over the past few years, we've also seen some of the social phenomena related to functional programming: how people feel about it, why they believe it works/doesn't work, and why they are (not) interested in doing it.

Lilja was standing in for the advertised speaker, his colleague Ville Laurikari, who couldn't attend. Their company's most visible product is *SSH Tectia Client/Server*, a remote shell and file transfer tool; this is written in C and C++, because it has to run on and interact closely with many OS platforms. They also produce *SSH Tectia Manager*, a remote administration tool for their client/server products, installations and configurations. This is a three-tier enterprise application; the server is mostly written in Scheme (56% of the code), and the client in C (43%) and a bit of Standard ML (1%).

They have their own in-house Scheme implementation, which is purely interpreted rather than compiling to bytecode. This implements most of R5RS, with various in-house extensions (such as `case-lambda` and `define-struct`). Their C libraries are based on cooperative multitasking with `select()`, event loops and callbacks; primitive Scheme I/O operations always yield to the event loop. There is a C interface, providing bindings for C code through an interface definition language. They have a continuation-based web application framework, similar to that of PLT Scheme. There is a database interface, providing queries through SQL strings and result sets as lists, and a persistent object interface.

Among the problems they experienced with using Scheme were: a lack of standard libraries, leading to lots of in-house ways of doing things; maintaining their own in-house implementation (in fact, two of them) of Scheme, written in C; the

lack of a module system and debugger; the effort required to integrate with their event-driven C libraries; and the temptation to shoot yourself in the foot with various fancy features of Scheme, such as macros and continuations. They had no difficulty in convincing management to adopt an unusual language, because the company founder was a big fan of Scheme. As for building much of the infrastructure in-house, they have been using the language for quite some time, predating systems such as PLT Scheme: their continuation-based web server development was contemporaneous.

The company is currently piloting the use of Standard ML and MLton in their Tectia Manager product; the goal is to replace all of the C code and perhaps some of the Scheme code. They have found SML to be a nice clean language, with a great module system, good support for bit-twiddling, and high performance code generated by MLton. They will continue investing time into SML and MLton, mainly in the area of new ports and libraries, and would welcome help from others.

Among the problems they experienced were: few people know SML, especially in Finland; MLton has not been ported to some platforms important to the company; there is no great IDE, especially for Windows (there is an Eclipse plug-in, but it is only for SML/NJ and is not maintained anyway); again, there is a lack of libraries; and it is not possible to take advantage of multiple cores in one process.

The company chose not to use functional programming in their flagship Client/Server product; they use C instead, for which they have a long history. Compilers are readily available on any platform (they currently use HP-UX, AIX, Solaris, Windows, and z/OS), as are IDEs and other support tools, and it is possible to write high-performance code using multiple cores. Developers with a knowledge of C are readily available, and learning it is not considered a career-limiting move. They could perhaps use a higher-level language for most things, but there is no popular functional language providing all of these benefits.

Cross-Domain WebDAV Server (John Launchbury, Galois)

Abstract: At Galois, we use Haskell extensively in a carefully architected DAV server. Our clients need very strong separation between separate network access points. Haskell gave us critical flexibility to provide major pieces of functionality that enable the separation, including the implementation of a completely new file system. Interestingly, however, we implemented the single most critical component in C! In this talk we will discuss our experiences, and draw lessons regarding the appropriateness—or otherwise—of functional languages for certain security critical tasks.

Galois is currently incubating a cross-domain WebDAV server for the US government. WebDAV stands for *Web-based Distributed Authoring and Versioning*, and provides networked file-mounting across the internet; it is an HTTP-like protocol

that provides writes, versioning, and metadata as well as simple reads. In some environments, different security domains are physically separated, to prevent information leakage. A cross-domain WebDAV server must breach this air-gap, providing validated reads from a high-security to a low-security domain, but not writes, or reads in the opposite direction. (High-security writes to low-security data are prohibited, largely to prevent information leakage in the case of malicious software in the high-security domain. Moreover, it is not enough just to prohibit explicit information transfer; implicit 'covert channels' must be prohibited too, such as the appearance of locks on files.) Lacking such a facility, many high-security users have multiple computers on separate networks.

Launchbury described Galois' *Trusted Services Engine* architecture, based on *Multiple Independent Levels of Security*. This involves factorising the security architecture, minimising the number of components requiring high assurance. They keep each component as simple as possible, and use formal methods in critical places. In particular, Launchbury described the *Wait-Free File System* (WFFS) and *Block Access Controller* (BAC) components.

The WFFS ensures that read access from a high-security domain is invisible to a low-security domain. This was implemented in a functional style: "We modelled it in Java, then for efficiency we translated it into Haskell." The BAC is a high-security device, and so had to be written in a (simple subset of) C, taking about 1000 lines. They used the theorem prover Isabelle on the code. The model-to-code correspondence was initially a little difficult to establish, so they rewrote the functional code in a monadic style to make it easier.

Launchbury reported that Haskell gave Galois "the engineering freedom to build the system right". The Foreign Function Interface worked well and reliably; the module system mostly worked well, although they would have liked a way to manage imports more flexibly. Various Haskell tools (profiling, testing with QuickCheck and HUnit, Cabal for packaging, Haddock for documentation) were extremely helpful. Performance was great, and concurrency really easy to use; most early performance problems were with library calls to manipulate binary objects. Andy Gill, a key developer, observed that "to a first approximation, strictness versus laziness didn't matter squat". But on the down side, one cannot build high-security components directly in Haskell: allowing the Haskell heap and run-time systems to span security domains would require the validation of 30,000 lines of C.

Discussion
(chaired by Don Syme, Microsoft Research)

Syme ended the day by chairing a discussion on hiring functional programmers. What skills should we look for? Where should we look? Is retraining feasible? What forums and teaching qualifications do we need? Can we help schools 'sell' functional programming? He sensed a sea change over the last few years: it used to be the case that just one or two companies were hiring functional programmers, but

now he could list four or five such companies off the top of his head, and a quick poll indicated about ten companies represented at the meeting were looking to hire functional programmers.

One representative stated that their company is not looking for just Haskell programmers, although most of its work is done in Haskell; rather, they are looking for the functional way of thinking—for abstraction, and the algorithmic view—and functional programmers are self-selecting for that approach. Another used to ask how recruits liked the functional programming course at UNSW as an indicator; he hires Java programmers, but looks for Haskell on their CV. Someone said that he doesn't look for functional programmers — he thinks there are simply good programmers and bad programmers. Someone else agreed: they are just looking for smart people.

Regarding training programmers in functional languages, someone reckoned that it was easier to teach those straight out of college. Others agreed: older programmers sometimes have difficulty switching. Someone reported that they have educated a lot of average programmers in functional programming, enabling them to become reasonably productive; putting them in C++ development would have been a disaster, requiring experts continually to clean up their messes, but they can't do as much damage in a functional language.

How can we spread the word about functional programming? It is promising that it seems no longer to be purely a geeky academic subject: the Association of C and C++ Users conference in 2008 has a functional programming stream, and Joe Armstrong has been doing keynote presentations and Simon Peyton Jones doing tutorials at practitioners' conferences. Some students just don't get Haskell early on—but they don't get Java either, and they just shouldn't be learning CS; others are really good, but they're motivated more by cool stuff than by jobs; for those in the middle, it's useful to see evidence of the importance of functional programming. It's much more convincing to see that evidence from people in industry than from academics.

CUFP in the future (Kathleen Fisher, AT&T)

Participants were encouraged to sign up to the mailing list `cufp@googlegroups.com` and the Google Group `http://groups.google.com/group/cufp`. The *Journal of Functional Programming* is now soliciting articles on 'Commercial Uses'. ICFP this year had a special category of 'Experience Reports': short papers adding to the body of evidence (as opposed to the body of knowledge) about using functional programming, which hopefully will be continued.

The next CUFP meeting takes place in late September 2008 in Victoria, BC, associated with ICFP as usual. What should be done differently during CUFP next year? Tutorials? Information or demonstration booths? An "interop" exercise? 1-minute introductions by participants? A venture capitalist? Please send suggestions to Jim Hook, the general chair of ICFP2008.