JFP **22** (1): 1–8, 2012. © Cambridge University Press 2012
doi:10.1017/S0956796812000020 First published online 6 March 2012

CUFP 2011 Workshop Report

ANIL MADHAVAPEDDY
Computer Laboratory, University of Cambridge
15 JJ Thomson Avenue, Cambridge CB3 0FD, UK
(*e-mail:* avsm2@cl.cam.ac.uk)

YARON MINSKY
Jane Street Capital
1 New York Plaza, New York NY, USA
(*e-mail:* yminsky@janestreet.com)

MARIUS ERIKSEN
Twitter, Inc.
795 Folsom St., Suite 600
San Francisco, CA 94107, USA
(*e-mail:* marius@twitter.com)

1 Introduction

Commercial Users of Functional Programming (CUFP) is a yearly workshop that is aimed at the community of software developers who use functional programming in real-world settings. This scribe report covers the talks that were delivered at the 2011 workshop, which was held in association with ICFP in Tokyo. The goal of the report is to give the reader a sense of what went on, rather than to reproduce the full details of the talks. Videos and slides from all the talks are available online at http://cufp.org.

2 Keynote: pragmatic Haskell

Lennart Augustsson from Standard Charter gave the keynote address, relating his longtime use of Haskell and Haskell-derived languages in commercial settings. Augustsson's experience with Haskell dates back to its inception; he authored the first Haskell compiler, `hbc`, which remains competitive with the Glasgow Haskell Compiler to this day. Augustsson subsequently developed several Haskell variants that were tailored for his needs.

The first of these was `pH`, the parallel Haskell compiler (Aditya *et al.*, 1995). `pH` sought to exploit the implicit parallelism in a Haskell program by combining the Haskell syntax and type system with the evaluation strategy of the `id` parallel programming language (Arvind *et al.*, 1978). `pH` introduced several concepts that are still used in modern systems, such as the `MVar` abstraction.

Augustsson then applied his Haskell experience to the field of hardware design. The Bluespec hardware description language is a full compile-time implementation of Haskell, outputting Verilog for hardware synthesis. Bluespec significantly raised the level of abstraction for hardware design and is still available commercially

today (Rishiyur & Arvind, 2008). Interestingly, Bluespec's type system directly incorporates numbers and arithmetic.

Augustsson is currently working in the banking industry, providing in-house technology for traders and quantitative analysts. `Mu` is Lennart's latest Haskell dialect designed specifically for this use. Most traders and quantitative analysts are chiefly interested in data transformation and do not necessarily care about effectful operations. `Mu` provides a simplified dialect of Haskell catering to these needs. Interestingly, `Mu` eschews Haskell's laziness and is a strict language instead. A number of pragmatic design decisions were made: most performance sensitive code reuses C++ from in-house libraries, and strings in `Mu` are not lists of characters. Furthermore, recursion is provided only optionally (and only 6% of their modules make use of it). `Mu` provides a fine-grained I/O monad, allowing for both "I/O" and "I" (input only, that does not modify the world). In order to attain easy interprocess communication, all values in `Mu` are serializable.

`Mu` is a true Haskell dialect in that code written in `Mu` may be compiled with a Haskell compiler. As with its language and libraries, the `Mu` compiler is a work of pragmatic design. All compiler transformations are done assuming it is operating on terminating code, and the compiler uses `LLVM` (Lattner & Adve, 2004) for its backend.

Standard Chartered's `Mu` codebase is of significant size and lives within a library for quantitative analysts written in a combination of C++, Haskell and `Mu` itself. Language interoperability is key; all parts of the system can easily be invoked from Excel, C#, Java or any other component.

Their experience with strict semantics has been positive. Particularly useful is the ease of obtaining meaningful stack traces, tracking resource usage, debugging and exception propagation. The chief downside of strict semantics, in their experience, is the increased difficulty of modular composition.

Augustsson noted that Excel has a more expressive effect system than Haskell and provides an effective front-end to composing `Mu` modules. He demonstrated coding `Mu` within Excel live on stage, via an interactive and interpreted version.

3 Cryptol: theorem-based derivation of an AES implementation

John Launchbury from Galois then described Cryptol, their declarative specification language for cryptographic protocols (Erkök & Matthews, 2008). Galois is now using Cryptol to derive a highly efficient implementation of AES targeted at FPGAs.

Cryptol is a first-order functional language with size-type declarations. Recursion is available via stream equations. The sequentiality constraints are due to data dependencies, making it possible to efficiently evaluate this language on FPGAs. Cryptography is also a natural match for FPGAs, and Cryptol makes it easier to experiment with several implementations. Cryptol has a `theorem` keyword to express properties such as the inverse relationship between encryption and decryption operations. A tool that is similar to QuickCheck (Claessen & Hughes, 2000) is used to generate test cases to ensure that these theorems hold.

https://doi.org/10.1017/S0956796812000020 Published online by Cambridge University Press

The derivation of the design was preceded by a series of stepwise refinements to the original specification, all of which were written in Cryptol. The theorem-based testing system was used as a way of gaining confidence that each stage in the refinement was solid. The end result was a chip that was competitive with top-notch manually constructed implementations, but that the designer could have far more confidence in.

4 Erlang: large-scale discrete event simulation

Olivier Boudeville from EDF described the use of Erlang for building large-scale simulations. Sim-Diasca (Simulation of Discrete Systems of All Scales) is a system implemented in Erlang for building large scale discrete simulations of the kind that are used for simulating Smart Energy Grids and other large-scale industry projects.

Sim-Diasca was—as its name implies—designed to meet relatively uncommon scalability requirements, supporting upwards of millions of parallel instances of complex models. It prescribes no fixed topology and uses Erlang's actor mechanism for communication between nodes. The main role of the generic engine is scheduling. Its scheduler enforces causality, reproducibility of the model simulation and some forms of ergodicity.

As typical language choices for this domain are C++ and CORBA, the authors wrote a macro package—dubbed WOOPER—to aid development and provide a more familiar environment by providing object-oriented primitives on top of Erlang.

The team did encounter resistance with its esoteric language choice. In particular, hiring developers was harder. However, Boudeville characterized their choice of Erlang and functional programming as "massively positive". In particular, Erlang's support for distributing computation coupled with more suitable language constructs made complex algorithms significantly easier to express.

5 Erlang: testing safety critical automotive components

The next talk came from Thomas Arts, who is the CTO of Quviq, a company he co-founded with John Hughes. Quviq is devoted to automated testing of software using QuickCheck and Erlang. The talk covered the application of Erlang and QuickCheck to the testing of automotive components. The average modern car has over 64 computers that are networked (for example, "the brakes might need to know how fast the car is going!"). Manufacturers use components from a variety of vendors and so interoperability testing takes on critical importance.

The industry standard for automotive components is AUTOSAR.[1] Initially, the AUTOSAR consortium outsourced its testing, and over thirty person-years were spent on writing manual tests for component interoperability. The result was disastrous and unsuitable for use. The chief reason for this is that AUTOSAR is highly configurable, with thousands of standardised parameters that are difficult to express

[1] http://www.autosar.org/

https://doi.org/10.1017/S0956796812000020 Published online by Cambridge University Press

manually. Furthermore, this means that meaningful tests must be parametrised on the configurations.

The solution came in the form of Erlang and QuickCheck. Quiviq's AUTOSAR testing produced far better test coverage, tailored to the manufacturer's particular configuration. Furthermore, the terseness of Erlang and QuickCheck reduced the code size of the tests by at least an order of magnitude.

6 OCaml: mobile HTML5 development

We moved on from testing and simulation to the world of functional web development in Japan. Keigo Imai delivered an entertaining talk about the consultancy (IT Planning, Inc.) he works for, where 45% of their annual sales are from functional programming projects.

Their typical sales story goes like this. Customers specify their choice of programming language and an impossibly tight deadline. ITPL then propose using their existing OCaml codebase to deliver the solution, but within the required deadline and with much less risk. ITPL continually educates its customers about the benefits of rapid development using functional languages.

Imai then demonstrated an example project: a foreign exchange chart application that works on iPhone and Android, dynamically drawn using an HTML5 canvas. It was written in OCaml and translated to Javascript using the OCamlJS compiler.[2] In their experience, static typing was very helpful for web development. In particular, typing of DOM elements avoided many runtime errors, especially in the complex `<canvas>` tag. Higher level language features also facilitate the asynchronous programming style prevalent in web apps. Their Web SQL database API is wrapped in a monad and continuation passing style, allowing for ease of programming without having to create explicit callbacks chains. OcamlJS allows for inlining Javascript, which was useful when more low-level control was required.

Performance has been fine with OCamlJS, and no bugs have been reported yet on its output. To quote Imai, "it is written in our miraculous super OCaml technique!". An audience member asked if codesize (from the generated Javascript) is a problem, especially in view of this being a mobile project. Keigo replied, they haven't had any problems with this.

7 Scala: large-scale internet services at Twitter

Twitter is a social network for sharing short text messages. It is growing in popularity extremely fast, and some backend components have to serve in excess of 10^6 queries per second. Steve Jenson and Wilhelm Bierbaum described how many of Twitter's infrastructure components are implemented in Scala. The JVM is their preferred virtual machine due to its maturity and performance, and Scala provides a much better type system and functional programming features than Java.

[2] https://github.com/jaked/ocamljs

https://doi.org/10.1017/S0956796812000020 Published online by Cambridge University Press

All Twitter client HTTP requests are answered by a reverse proxy called TFE, which routes requests using the Finagle distributed RPC system.[3] Request streams are passed through filters that transform and apply processing functions on them. The process is afforded the full facilities of the JVM, including the use of pre-existing libraries written in Java.

The use of the JVM means that many existing debugging and profiling tools (e.g. Yourkit, JStat, VisualVM) can be used, but the name mangling and anonymous functions in Scala occasionally introduce obscure results. The use of immutable values result in a lot of pressure on the garbage collector. When developing high-volume interactive services, it is not uncommon to spend a significant amount of time understanding and tuning garbage collection.

Twitter has also been hiring many programmers who have never used Scala before. While many of them are familiar with the concepts underlying functional programming, the syntax of Scala can be subtle, and is often difficult for beginners to learn.

8 F#: mobile applications using WebSharper

Adam Granicz from IntelliFactory began by pointing out that the market for mobile applications is massive. It is projected that in 2011, the phone market will have grown to 3.7 billion users, with 18% of them owning smartphones capable of running applications. However, smartphone platforms are a heterogenous bunch. The current major platforms (Android, iPhone, Windows Mobile) differ in both programming style and language choice.

Ideally, mobile applications could be developed using higher level abstractions, support compilations to multiple targets, and make use of desktop and cloud resources when available. The switch to proprietary and divergent APIs on different devices is a step backwards.

Javascript is becoming the intermediate language that connects these devices together, and Windows 8 is even promoting it for building native desktop applications. IntelliFactory built the WebSharper programming framework in F# that outputs Javascript that works with all of these diverse devices.

With WebSharper, all of the server and client code is written in F#, and compiles to a complete standalone web application. The F# interfaces make good use of the language's facilities: Type-safe URLs helps prevent common errors, and "sitelets" and "formlets" that are composable abstractions for fragments of websites.

9 Haskell: a real time programming project in real time

Gregory Wright from Alcatel-Lucent Bell Labs described a project that used Haskell to build the core of a real time communication system. It was built by the GreenTouch consortium, an organization of equipment vendors, service providers, and research institutes to show a new antenna technology that reduces the energy

[3] http://twitter.github.com/finagle/

https://doi.org/10.1017/S0956796812000020 Published online by Cambridge University Press

used by wireless networks. The goal of the project was to demonstrate an algorithm that reduces radio transmission power as the number of base station antenna elements is increased and is capable of scaling to antennas with thousands of elements.

An antenna can focus on a user and calibrate the amount of data sent over the link. As a result, the power levels of an individual handset can be adjusted with respect to a target rate. This was initially simulated and then later run on real hardware as a soft real time system.

In total, the project lasted 14 real days. As a result of the timescale, it used very vanilla Haskell (e.g. arrays weren't unboxed), with no tricks to improve memory behaviour, or strictness annotations. The STM library was used, as well as the DSP library from Hackage.

Despite this simplicity, the project was a big success. An important factor was that they always had a working system, with upgrades staged well. Haskell provided a high degree of safety from "crashing and burning".

Furthermore, the nature of the project was itself very compatible with the principles of Haskell. The core algorithms implemented were all "dataflow-like", using laziness quite effectively.

10 Erlang/OCaml: disco, a MapReduce platform

Prashanth Mundkur from Nokia Research in Palo Alto talked about the Disco Project. When Nokia started evaluating systems for distributed data processing, the most popular solution was Hadoop. However, this was a "massive pile of Java software" that was difficult to configure and operate. An Erlang hacker at Nokia—Ville Tuulos—wondered how hard it would be to implement a simpler alternative.

The first Disco prototype took a few weeks to implement. The core coordination components are implemented in Erlang making use of its runtime facilities for distributed computation. Python is the primary language used to write the data processing scripts, i.e. the "mappers" and "reducers". OCaml has recently been added as a more strongly typed data processing alternative language.

The Erlang environment provides quite a few useful tools that made Disco easier to implement. In particular, a shell to invoke remote procedure calls on hosts in the cluster and `fprof` to collect profiling information in real time. The dynamic typing did result in some hard to find bugs, and the `dialyzer` static analysis tool is now used extensively on the codebase (Sagonas, 2007).

Disco is open-source software, and available from http://discoproject.org.

11 mzScheme: functional DSLs for game development

Dan Liebgold from Naughty Dog Software in Santa Monica then came on stage with the first gaming related talk at CUFP. They produce the popular Uncharted game series for the Playstation, which is famous for its complex and interactive scripted scenes. Dan described modern game development as a major production effort where, roughly, artists produce data and programmers produce code.

https://doi.org/10.1017/S0956796812000020 Published online by Cambridge University Press

Naughty Dog has a history of using various Lisp dialects to handle the code and data in a unified way. But when making the jump from the Playstation 2 to the Playstation 3, they decided that maintaining a custom Lisp-based game development system was too costly, and instead dedicated their efforts to rebuilding the tools, engine, and game in C++ and assembly language.

This decision left no scripting system for gameplay and, more importantly, no system for creating DSLs and the extensive glue data that is typically required to develop a major video game. There was no off-the-shelf scripting system that fit the stringent memory requirements in a Playstation 3, and no language that would allow rapid DSL creation that fit into the existing tool chain.

With a bit of naivety, a penchant for the Scheme language, and a passion for functional programming techniques, the team dove in and put together a system to fill in the gaps! They used mzScheme[4], which can compile to fast native code. Dan reported that the results have been very good, but not without issues. In particular, garbage collector performance sometime led to manual tuning being required, and build environment integration was tricky. Syntax transformations and error reporting led to confusion with new programmers too.

On the other hand, the functional nature of the system was a big win, as it allowed them to flexibly distill game data down to just the right form to embed into the resource-constrained run-time environment. The final result is a system where programmers, artists, animators, and designers are productively programming directly in an S-expression Scheme-like language. Dan closed his talk by wowing the audience with the trailer for the game, which has now been released and is garnering extremely positive reviews.

12 OCaml: the Acunu storage platform

Tom Wilkie from Acunu in London presented their Castle storage system to optimise big data storage and retrieval. Castle consists of two components: a management service and storage stack. The management service is written in OCaml and handles control requests (e.g. snapshots, backups, rollbacks, coordination). The storage stack implements an efficient disk-backed indexed key-value store. While developing their storage stack, Acunu needed rapid prototyping in order to explore new techniques and iterate on ideas. Given that issues in storage systems tend to surface only after a certain amount of data or request volume, the prototypes also needed to be reasonably performant.

However, although OCaml was effective for implementing the core algorithm, they spent a lot of time on the support code to serialise data to and from disk. When the filesystem codebase was moved to Java, performance increased by six times. The audience suggested a few techniques to speed up the OCaml code, threading library, the Bitstring syntax extension, and increasing the number of I/O threads used. Wilkie acknowledged these could help, but that their availability should be promoted and documented better.

[4] Now known as Racket, available: at http://racket-lang.org

https://doi.org/10.1017/S0956796812000020 Published online by Cambridge University Press

13 Conclusion

This year's CUFP workshop covered a broad set of functional languages—Scala, Haskell, Scheme, OCaml, Erlang and F# were all well represented, along with a nascent but growing interest in more formal tools such as Coq and Isabelle. We also ran a well-attended 2-day tutorial series on using Scala, OCaml, Haskell and F# for problem solving in scientific computing, web programming, parallel programming and cloud computing.

The diversity of industries, where functional programming is represented, is also encouraging. Our speakers worked in the financial industry, big data processing, safety critical systems in automobiles and energy systems, the real time and mobile Internet, and for the first time, the gaming industry.

We would like to thank Michael Sperber for organising the CUFP tutorial series, and Manuel Chakravarty, Zhenjiang Hu, and the whole ICFP/CUFP team for their assistance in Tokyo. We look forward to continuing the conference series in Copenhagen next year!

References

Aditya, S., Arvind, Maessen, J.-W., Augustsson, L. & Rishiyur, S. N. (1995) Semantics of ph: A parallel dialect of Haskell. In *Proceedings of the Haskell Workshop (at FPCA 1995)*, pp. 35–49.

Arvind, Gostelow, K. P. & Plouffe, W. E. (1978 December) *An Asynchronous Programming Language for a Large Multiprocessor Machine*. Tech. Rep. TR114a. UC Irvine.

Claessen, K. & Hughes, J. (2000) QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: ACM, pp. 268–279.

Erkök, L. & Matthews, J. (2008) Pragmatic equivalence and safety checking in Cryptol. In *Proceedings of the Third Workshop on Programming Languages Meets Program Verification*. New York, NY, USA: ACM, pp. 73–82.

Lattner, C. & Adve, V. (2004) LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04. Washington, DC, USA: IEEE Computer Society.

Rishiyur, S. N. & Arvind. (2008) What is Bluespec? *Sigda Newsl.* **38**(December), 1–1.

Sagonas, K. (2007) Detecting defects in Erlang programs using static analysis. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. New York, NY, USA: ACM, pp. 37–37.