JFP **23** (6): 701–712, 2013. © Cambridge University Press 2014
doi:10.1017/S0956796813000257

Commercial users of functional programming workshop report

MICHAEL SPERBER
Active Group GmbH, Hornbergstra ße 49, 70794 Filderstadt, Germany
(*e-mail:* sperber@deinprogramm.de)

ANIL MADHAVAPEDDY
Computer Laboratory, University of Cambridge, 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK
(*e-mail:* anil@recoil.org)

1 Overview

Commercial Users of Functional Programming (CUFP) is an annual 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 2012 workshop, which was held in association with International Conference on Functional Programming (ICFP) in Copenhagen, Denmark. 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: adopting functional programming

Kresten Krab Thorup, CTO of Trifork, Aarhus, Denmark delivered the keynote address. He took us on the voyage he had taken from being an "object head" to "Erlang land." Thorup's foundational training in software development was all in terms of object-oriented methodologies. He went on to work on Objective C for NeXT, then earned his PhD and subsequently founded Trifork, an IT services company that currently employs 250 people and develops software solutions, provides training, and organizes several well-respected conferences.

While Trifork originally capitalized almost exclusively on its Java expertise, it now successfully applies Erlang in large-scale industrial projects. Taking cues from anthropology, Thorup described how many organizations have not been able to make such transitions easily. Groups tend to gather around an idea that keeps them together, and try to keep new ideas at bay. This makes it difficult for long-time OO developers to adopt functional programming.

Trifork managed to stay flexible by making learning about new ideas and communicating them as part of their regular operation. Everyone at Trifork is encouraged to spend 10% of his/her time in the structured exchange of knowledge by giving presentations, organizing meetings, give training classes, or organizing conferences.

Thorup reviewed object-oriented programming and the ecosystem around it to show how it had become successful through an intuitive idea – "an object is an independent encapsulated entity that interprets inputs on its own account" – but also because of the

availability of thinking tools: graphical notation for design, tools for mapping those designs to programs, books on common problems akin to "Design Patterns" (Gamma *et al.*, 1995), analysis methods for producing systems, and standardized qualification processes.

However, Thorup also saw a serious problem with the object-oriented model, as objects have no coherent model of time and no good way to compose behaviors over time. With the rise of multicore and distributed computing, these become increasingly important. Erlang, supporting functional programming and an actor model for concurrency, parallelism, and distribution, addresses this issue. Thorup stressed that Erlang is not primarily a functional programming language, but that functional programming helps Erlang meet its goal of being a language for writing robust distributed applications.

Thorup described large-scale projects done using Erlang, one for managing healthcare records in Denmark, and another for sharing data among "sometimes connected devices" such as a cell phones at a music festival.

Thorup concluded by noting that two fundamental classes of problems in software development require different classses of solutions: *Interactive systems* with multiple parties are fundamentally stateful, and where developers should understand the handling of state – for those problems, actors are a good model. *Transformational systems* map input to output, where developers want to abstract over the details of hardware utilization, the handling of mutable state and coordination.

3 OCaml: Jane Street status report

Jane Street is a proprietary quantitative trading firm that has been a well-known user and supporter of OCaml, and was last presented at CUFP in 2006. Yaron Minsky reviewed the full past decade of OCaml use at Jane Street.

Jane Street has the following three kinds of requirements on their own software:

Correctness. Jane Street trades billions of dollars every day – much more than the company is worth. Hence, a software error can have disastrous consequences.

Agility. Jane Street needs to be able to adapt the software quickly to exploit new market opportunities as they are found.

Performance. The software needs to run fast to exploit market opportunities.

Jane Street has written OCaml code for a number of application areas: research tools for investigating trading strategies, trading systems replacing legacy systems written in VBA/Excel that make trades automatically, order gateways that implement protocols to interact with markets, post-trade software to analyze and clean up completed trades, systems infrastructure to manage clusters of physical machines, development tools, trading tools, and tools for managing market-data and desk infrastructure.

Minsky could not remember a single crucial error in the software that was discovered after deployment. Code at Jane Street generally fails by turning off the system, which is acceptable in this environment. Jane Street's user interfaces are all text-based and written using Curses and OCaml – this "prevents bad UX designers from doing UX."

Another important task that favors OCaml is how the type system eases large refactoring projects. Jane Street's code base is designed to use the OCaml type system to statically

https://doi.org/10.1017/S0956796813000257 Published online by Cambridge University Press

catch most mistakes at compile-time, thus helping trap most common mistakes that occur in the middle of large-scale refactoring efforts. The type system helps programmers write readable, explicit code, which is more important than great productivity increases or extremely concise programs.

Jane Street has developed a number of generally useful libraries for OCaml, many of which are open source.[1] In particular, *Core* is an enhanced "standard library," meant to supplement the minimalistic libraries that ship with OCaml. *Async* is a library with monadic concurrency abstractions, similar to those available in other functional languages such as F# (Syme *et al.*, 2011). *Incremental* describes large-scale computations with small updates, *Catalog* is a publish/subscribe system, and *Nile* is a distributed message-passing library.

In summary, Jane Street is fully committed to OCaml, and will continue to do its development in OCaml. Jane Street will continue to contribute to open-source projects, and to collaborate with others working on or in OCaml.

4 Erlang: transmitting customized ads to set-top boxes

Macías Lopez, David Cabrero, and Laura M. Castro from the University of La Coruña reported on the *ADVERTISE* project (Lopez *et al*. 2012a, 2012b). ADVERTISE is a distributed system for transmission of customized ads to TV set-top boxes via the TV network of a cable provider. The system – entirely written in Erlang – compiles events, emits "advertising signals" to set-top boxes, and collects statistics about how many times specific ads were displayed on consumer boxes. ADVERTISE sends ads to more than 100,000 clients.

Lopez reported on the difficulties of developing and deploying such a large-scale system. In particular, after the original system implementation, the customer provided hardware that did not satisfy the original minimum specifications. Moreover, the network exhibited frequent node failures and netsplits, which disrupted the operation of the original system. Thus, while Erlang was a good choice for implementing the system, merely using Erlang did not make a distributed system robust in highly unreliable environments. ADVERTISE yielded insights into best practices for implementing distributed systems in such environments.

Netsplits in particular are problematic, as nodes may incorrectly conclude other nodes are down, and then compete for control of the network. This may lead to data inconsistencies and duplicate implementations of responsibilities assumed to be unique. An ADVERTISE node, when it loses network connectivity, immediately suspends execution and waits until it is restored, choosing consistency over availability in such scenarios to avoid corrupting advertising campaigns.

5 Haskell: used in Citrix XenClient

Matthias Görgens reported on using Haskell in the XenClient project at Citrix. XenClient is a virtual machine manager for clients, primarily laptops in corporate and government environments, where XenClient offers functionality different from the long-established

[1] http://janestreet.github.io/

https://doi.org/10.1017/S0956796813000257 Published online by Cambridge University Press

XenServer such as trusted-computing support with hard drive encryption and native graphics performance.

The XenClient management stack consists of many daemons that communicate via D-Bus and V4V.[2] Originally many of the daemons were written using Ruby, but a rising bug count motivated the development team to look for alternatives, particularly statically typed languages to catch more bugs at development time. Some daemons were then rewritten in Haskell. By now there are about 25,000 lines of code in Haskell in the system. The XenClient daemons are typically long-running, perform short bursts of communication and computation upon a request discovered via polling, do not hold much state, and are restartable. Haskell is well suited for this kind of application.

As XenServer already uses OCaml for its management toolstack, it is a bit surprising that XenClient chose Haskell. Görgens cited personal preferences, the availability of more libraries, and the fact that Haskell "relieves pressure to share code with XenServer."

Görgens also cited a few problems with using Haskell. XenClient found it difficult to handle IO-heavy workloads, make Haskell compile with the OpenEmbedded build system used in XenClient, and successfully train developers not yet familiar with functional programming.

6 OCaml: functional programming @ Ghent IT Valley

Romain Slootmaekers and Nicolas Trangez of Incubaid Research Lab reported on using OCaml for implementing a distributed storage service. Incubaid Research Lab is an incubator laboratory for startup companies.

In 2009, Amplidata – one of those spinoffs – was working on a *dispersed storage system* (DSS). The system consists of a multi-stage pipeline involving metadata storage, encoding and decoding, storage management, and the backend disks. At the time, development on the storage-management component, written in C++, had stalled: There were problems with resource management and threads, the software had many bugs, and its object model had poor locality, which led to poor performance. Slootmaekers was able to correctly reimplement the storage component in OCaml within two days, which led to Amplidata considering using languages other than C++ for system development.

The original re-implementation of the storage component had to be done quickly, and was able to leverage OCaml's object-oriented substrate to duplicate the architecture of the C++ original. Also, the existence of precise specifications and a test suite helped speed the rewrite.

The successor version of the system was then re-implemented mostly in OCaml, and involved more leisurely and more complete refactoring. The rewrite used the *Lwt* (Vouillon, 2008) library, and was delightfully painless. Performance improved more than twofold, and Amplidata was able to preserve code size while adding significant features. In particular, the newly written Arakoon[3] distributed key-value store was developed to hold the system's metadata.

[2] a VM-to-VM communications protocol.
[3] `http://arakoon.org`

https://doi.org/10.1017/S0956796813000257 Published online by Cambridge University Press

OCaml helped developers through type inference, a fast compiler that produces sufficiently fast code, the convenient C FFI, and the help the type system gives when refactoring. Downsides are poor tool support, a scarce and fragmented library landscape, problems with multicore support, and that the object-oriented model does not always fit well with the rest of the language. Slootmaekers also cited as notable problem the lack of visual tools to talk about system architecture.

In conclusion, Slootmaekers noted that any sufficiently large project should be ready to use more than one language, and that functional programming and distributed systems are a good match.

7 Star: from streams to functions (and back again)

Frank McCabe of Starview Inc. reported on Star, a new programming language, that is part of Starview's event-processing platform.[4] The purpose of Starview's platform is to provide *operational intelligence*: to notice when a significant business event happened, and to decide what to do about it, in real time. The Star language was developed to express this intelligence. It started out as "StarRules," a simple language based on an "on pattern do something" construct. It could perform straightforward statistical processing and infer significant events from event data.

StarRules's first application was scheduling in the semiconductor industry. In this domain, it is not enough for software to make simple inferences from the data – it needs to make complex decisions in the face of an ever-changing environment where machines frequently break, thereby invalidating any long-term plans. The software needed here did not fit StarRules' "on pattern do" construct. Consequently, Starview decided to turn StarRules into a general-purpose language and renamed it to Star.

A number of requirements shaped Star's design, and it needed to support different programming styles in use at Starview. The scheduling application for semiconductor fabrication plant demanded safety, as errors in the deployed software can be extremely costly. Moreover, it needed to satisfy modest real-time requirements.

McCabe then formulated basic design requirements for Star: to strongly support certain safety properties, but also to build tooling and serve as a communication medium in teams. The language also has automatic type inference to avoid the bureaucracy of dealing with types in Java. The type system is based on algebraic types rather than objects and has no `null`. This decision was controversial within Starview, which is still in many ways a "Java house."

Star amalgamated influences from various languages: in particular functions as an organizing principle from other functional languages, macros from April (McCabe & Clark, 1995) and Lisp to support syntactic extensibility, as well as the Concurrent ML substrate for parallelism and concurrency (Reppy, 1999). The type system borrows much from Haskell, as Star's type contracts are a variant of Haskell's type classes (Wadler & Blott, 1989). Star also includes influences from Prolog and SQL. Moreover, its actors are based on *speech actions* (Searle, 1969) and provide a mechanism for implementing agent-like entities. Star

[4] Star is open-source and available at `www.star-lang.com`

https://doi.org/10.1017/S0956796813000257 Published online by Cambridge University Press

is designed to be readable rather than concise: this makes its texture distinct from other functional languages such as Haskell or ML.

The combination of functions, macros, and overloading provides a coherent methodology for implementing domain-specific languages (DSLs) in Star. A developer can start with an ontological commitment in his/her problem domain, provide appropriate syntax, which is translated into a macro invocation, which is turned into function invocations, which are often backed by contracts. In particular, actors are implemented this way, as is higher level functionality for analytics and event processing.

8 OCaml: functional big-data genomics

Ashish Agarwal reported on the Genomics Sequencing Core used to enable entry, storage, and analysis of genomic sequencing data. This was a joint work with Sebastien Mondet, Paul Scheid, Avid Madar, Richard Bonneau, Jane Carlton, and Kristin C. Gunsalus at New York University. The software that Agarwal described is implemented in OCaml.

Since the sequencing of the human genome in 2000, modern sequencing equipment has been getting faster quickly. The data that accumulates in genome sequencing double in volume every five months, while storage costs per amount of data are *halving* every 14 months. This places high demands on the computational infrastructure used to process sequencing data. In particular, where the sequencing itself – performed by custom machines – used to dominate the time spent in a typical genome-related project, the emphasis is migrating to experimental design and datastream analysis.

The project described by Agarwal provides computational infrastructure for accepting and storing sequencing data and sequencing, and making it accessible to distributed computations on a compute cluster. In particular, the Genomics Sequencing Core provides an application server for managing the overall functioning of the system, a job queue for computations that interfaces to the compute cluster, and a web front end for the entire system. The system maintains metadata for racking samples, libraries, and protocols.

The data that accumulates in the system is characterized by high volume and also a high variety of different formats. The velocity at which it arrives is not yet a problem, but may become a challenge in the future. Agarwal and his colleages have developed a DSL embedded into OCaml that is used to generate multiple system functionalities, among them serialization, SQL schemas, query scripts, and OCaml code for performing reads in inserts, web widgets, and diagrams. This enables rapid development of functionality and easy migration between formats.

The Genomics Sequencing Core also maintains a virtual filesystem that distinguishes between “original” and “derived” data, recomputing “derived” data on the fly.

The entire system is implemented in OCaml, using a wide variety of libraries – in particular the Ocsigen (Balat *et al.*, 2009) web framework, the Core and Batteries libraries, and the Biocaml, PG’OCaml, Xmlm, and OCamlNet libraries. Developers having at their disposal about 1.3 full-time-equivalents of time built the system and delivered the first version in production within two months. The experience with OCaml has been mostly positive. Agarwal cited OCaml’s industrial-strength implementation, the availability of

https://doi.org/10.1017/S0956796813000257 Published online by Cambridge University Press

needed libraries, and the excellent performance. Agarwal mentioned the complications maintaining a build system and the lack of "blessed libraries" as factors that could still be improved.

Agarwal closed by summarizing the standing of functional programming in biology. Functional programming is becoming a recognized term, and thus the field is developing demand for software engineers who can acquire domain knowledge and build software fast. He noted that this profile is different from that of data analysts, who need more in-depth knowledge of the field and statistics.

9 Using F# to prove stabilization of biological networks

Samin Ishtiaq reported on the *Bio Model Analyzer* or BMA (Benque *et al.*, 2012) (available online at `http://biomodelanalyzer.research.microsoft.com/`) and developed by an interdisciplinary team at Microsoft Research. The BMA analyzes models from System Biology, which are program-like descriptions of networks that describe systems like skin or blood.

Particularly important to BMA are *stability* properties. For example, healthy skin should grow as many cells as it sheds. If it sheds more than it grows, sores develop and wounds do not heal well. Growing more than it sheds is the definition of cancer.

The programs describing such biological networks are asynchronous dataflow diagrams with typically tens of thousands of variables. Each variable has an associated update function that computes a new value from the values of other variables connected with it in the network. The objective of BMA is not just to simulate a network but also to prove general stability-related properties independent of a particular starting state, such as the existence of a unique fixpoint, several fixpoints, or cycles. This may be useful for developing new drugs.

Traditional program analysis tools do not scale or do not work for programs with this many variables. Consequently, BMA uses newly developed techniques. In particular, it attempts to prove lemmas for small subnetworks and propagate them through the entire network in the hope that enough lemmas propagate to prove stability. The prover core does not just report on the success of the proof search but also allows interactive stepping through the lemma propagation process. The designers in the BMA team have targeted the user interface (UI) at systems biologists, which use visualizations different from those familiar to computer scientists.

BMA consists of three parts: the prover core written in F#, Microsoft's Z3 SMT solver written in C++, and the user interface written in C#. The "debugging functionality" in the prover provides propagation steps in a lazy sequence. This is very natural in F#. The UI is not written in F#. In particular, it benefits from the better tool support for C#.

Ishtiaq had previously worked in OCaml using Emacs, and offered some thoughts on the transition to F# in Visual Studio: In particular, Visual Studio offers interactive type checking and thus supports exploratory programming he had not seen in Emacs. On the other hand, OCaml still offers some higher level abstraction mechanisms such as GADTs, modules, and functors, which F# still lacks.

https://doi.org/10.1017/S0956796813000257 Published online by Cambridge University Press

10 Developing an F# bioinformatics application with HTML5 visualization

Adam Granicz reported on joint work between University of Nebraska Medical University and IntelliFactory on the *functional genomics explorer* (or *fgx*), which visualizes the genetic structure of a Methicillin-Resistant Staphylococcus Aureus (MRSA). MRSA cause more deaths in the United States annually than HIV/AIDS.

The `fgx` software is a web application written in F# using IntelliFactory's *WebSharper* web framework. WebSharper contains a transpiler from F# to JavaScript, which allows the complete application – with all client and server code – to be written in F#. The `fgx` software uses an HTML5 canvas for drawing, and interfaces to the Krona JavaScript visualization library (Ondov *et al.*, 2011) through an F# wrapper. The data that `fgx` visualizes is too large to transport to the web browser in its entirety. Hence, `fgx` uses a combination of RPC calls and streaming via WebSockets to transfer it incrementally.

Plans are underway to extend the software to a full Laboratory Information Management System, which would track and manage sequencing experiments, samples, tools, and resources used.

11 Developing medical software in Scala and Haskell

Stefan Wehr reported on software developed by factis research in Freiburg for managing electronic patient records. Doctors can access and enter patient information on tablets, which synchronize with a central server.

The software needs to deal with significant amounts of data – a hospital department with 170 patients generates about five laboratory reports per patient per day and 20 reports in intensive care. The department also produces about 150 images per day, growing to more than 1,000 if CTs or MRI scans are made. The hospital's IT systems also produce 34,000 HL7 messages.[5] All in all, the software needs to synchronize about 13 Megabytes per patient per day and be very reliable and fault-tolerant.

The tablets serve as simple data viewers and entry points, with little domain-specific knowledge of the healthcare application domain itself. They are able to function offline, with periodic synchronization with a *synchronization server*, written in Haskell. The synchronization server generates documents from standardized import data, and regenerates them if the input data changes. It then transfers the documents and images to the tablets, and receives and processes data from them, keeping track of the synchronization state. These server components also contain only little domain-specific code.

The domain-specific code resides in a separate *data server*, written in Scala, which connects to the hospital's IT systems using various protocols and application programming interfaces (APIs) such as HL7, SQL, and DICOM SAP. It communicates with synchronization server via the *roundtrip* library, based on invertible syntax descriptions (Rendel & Ostermann, 2010). factis developed roundtrip for Haskell and then ported it to Scala.

Generally, the experience with using functional programming for this project has been positive. factis – with four employees and six freelancers – has been working on the

[5] HL7 is a set of interoperability standards for healthcare IT systems.

https://doi.org/10.1017/S0956796813000257 Published online by Cambridge University Press

application since 2010. As of late 2012, the synchronization server had about 55,000 lines of code, and the data server had about 37,0000 lines of code.

Wehr cited as Haskell's advantages the expressive and rich static type system, fine-grained control over side effects, "immutability by default," very good support for testing, the excellent support for concurrency as well as an active and helpful community. Haskell's laziness occasionally proved problematic, with space profiling providing only partial solutions. Also, `cabal` proved problematic. Wehr also reported that Haskell had a steep learning curve for developers new to Haskell.

Factis research chose Scala for the data server because many Java APIs were available that were helpful in integrating existing hospital IT infrastructure. Scala also has a very expressive, rich static type system, favors immutability, good support for testing, a powerful yet simple build system (sbt), and direct access to the Java API, and monitoring facilities through the Java ecosystem. Java programmers find it quite easy to adopt Scala. On the other hand, Scala provides no static control over side effects. Subtyping makes Scala's type system quite complex, which occasionally makes the type system difficult to control. Also, the easy migration path for Java programmers is a double-edged sword, as Scala programmers coming from Java do not always adopt functional programming as completely as they should.

All in all, Wehr concluded that both Haskell and Scala are excellent languages for commercial software development.

12 Erlang/F#: functional programs connected to the power grid

Sebastian Egner of Entelios AG, Berlin, reported on Entelios's project on coordinated reduction of electrical loads at industrial production facilities. Entelios is a venture captial-funded startup founded in 2010 and had 20 employees as of late 2012.

In order to maintain stability in the German electrical system, companies called *Transmission System Operators* (TSOs) maintain the network itself and balance supply and demand of electricity. This is increasingly becoming a challenge with the advent of electricity from renewable sources, particularly solar and wind energy. While these sources were able to provide 20% of Germany's electricity demands on a single day in 2012, their supply is subject to fluctuations and complex regulation.

Entelios provides *operating reserve power* – the electricity reserve needed to maintain stability – to the network by *demand–response management*, particularly by cooperating with big consumers of electricity in the 100–100 MW range such as arc furnaces or paper mills. These consumers can be switched off for limited time periods, thus providing "negative consumption" equivalent to positive production of electricity. Entelios is currently pre-qualified to provide operating-reserve power to all four TSOs.

As consumers, Entelios installs "EBoxes," small embedded systems connected to the Internet that interface with the control systems of the consumers. The software at Entelios's office controls the EBoxes. In particular, Entelios maintains two redundant Network Operation Centers at its two offices, with back-office software for maintaining communication with the EBoxes and front-office software for user interaction. Both components are subject to frequently changing requirements as the regulatory framework and market environment change.

The back-office software was originally written in Python, which was available on the embedded systems at the heart of the EBoxes. While Python worked well enough at first, software had difficulty keeping up with changing requirements, however, especially with unanticipated changes in the sampling rate. Python's threading facilities had trouble keeping up with the soft real-time requirements. As a result, the back-end software has been rewritten in Erlang.

The front office software is a rich client running on Windows, written in F# using the WPF framework. Development of the front office was outsourced to a company whose programmers are experts in *functional-reactive programming* (Elliott, 2009), which became the paradigm of the software for managing time-series data.

Experience with using functional programming has generally been positive and enabled Entelios to roll out new features and react to changing requirements quickly – in particular, moving from Python to Erlang showed a striking contrast. Whereas Python required substantial work going from prototype to production-ready software, prototypes in Erlang were usually quite close to production already. Moving from Python to Erlang required moderate deployment effort. Entelios filed two bug reports for the Erlang system that have since been pushed upstream.

The experience of F# was more mixed – while F# is an effective language for developing Windows Presentation Foundation (WPF) applications, new developers found it difficult to start working on the code base. F#'s notational density and rich interface to .NET often make it difficult to see the language feature that is exercised by a particular piece of code. Also, differences between production and debugging environments made it difficult to weed out some time and space leaks, which required withdrawing and redoing several releases.

13 Clojure: iPad analytics dashboard in the energy sector

Kevin Lynagh of Keming Labs described the approach his consultancy has been taking toward visualizing data. Keming Lab's mission is to make data formats that are not immediately fit for human understanding accessible for non-technical people.

Lynagh demonstrated two visualization applications that he had implemented: The *o8* framework,[6] done for the Harvard School for Public Health, provides a web-based platform to interactively explore and analyze human variation data. The other application was an analytics dashboard for a farm of wind turbines, which visualizes the status of wind turbines in a form easily accessible for maintenance personnel.

Both of these applications were implemented using Lynagh's *C2* data visualization library,[7] which is written in Clojure and ClojureScript. C2 enables developers to deploy their visualization applications either on the web server or on the client. For the client, ClojureScript compiles the code to JavaScript that runs in the browser.

Using a browser for visualization has a number of advantages: scalable vector graphics, CSS, a scenegraph encoded in the DOM, and many tools, all platform-independent. ClojureScript allows working around JavaScript's quirks and provides rich data structures,

[6] `https://github.com/chapmanb/o8`
[7] `http://keminglabs.com/c2/`

https://doi.org/10.1017/S0956796813000257 Published online by Cambridge University Press

namespaces, and consistency. Clojure's design was an important factor in the designing of C2 itself, as all data structures are immutable by default, and thus encourage the developer to think explicitly about state.

In the case of the wind-turbine farm, the essential state is just that of wind turbines. C2 allows transforming this state to a visualization by a pure function generating DOM fragments. The C2 framework then synchronizes the generated HTML code with the actual DOM in the browser, keeping the visualization consistent with the state. This decouples the specification of the underlying application from visualization. It also couples markup and software development, which may not be compatible with all design/development workflows. C2 thus provides a clear dataflow in one direction, which makes development easier than with direct manipulation of the DOM. C2's method is inherently slower than direct DOM manipulation, but the difference is irrelevant for many applications.

14 The awesome Haskell FPGA compiler

Peter Braams of Parallel Scientific described the benefits of using the *Haskell Hardware Description Language (HHDL)* for programming field programmable gate arrays (FPGAs). HHDL[8] is a DSL embedded in Haskell. HHDL programs can either run directly in the Haskell run-time environment or be compiled to Verilog code and deployed on FPGAs. Using HHDL instead of Verilog directly allows FPGA programmers to design at the conceptual level, and shorten the test/design cycle significantly. HHDL provides a rich set of types, and the ensuing type safety helps assure the correctness of HHDL programs. The initial version of HHDL took merely five months to develop.

Parallel Scientific uses HHDL to program an Arista Ethernet switch, which contains an FPGA directly connected to eight 10-GB Ethernet ports. As the FPGA sits directly in the data path, it can receive and thus process the Ethernet traffic faster than a separate CPU over a traditional bus. The data rate on the ports exceeds the bandwidth on a PCIe, for example. HHDL allows building combinators to parse and build packets, which can be layered to build complex algorithms. In particular, Parallel Scientific is using HHDL on the switch to build a ticker plant for financial exchanges, which aggregates and then normalizes and maintains market data from multiple sources. For ticker plants, performance is crucial. Parallel Scientific's solution uses both the regular CPU in the switch for command and control and the FPGA to receive and normalize financial data. The resulting application is able to keep up with the port traffic in real time.

15 Conclusions

This year's CUFP workshop covered a broad spectrum of general-purpose languages – Scala, Haskell, OCaml, Erlang, Clojure, and F# – but also an emerging breed of domain-specific languages such as Star and HHDL that are designed for a specific industry, but heavily inspired by the functional programming literature. There was also a strong surge of

[8] Designed by Serguey Zefirov, available at `http://hackage.haskell.org/package/HHDL`

https://doi.org/10.1017/S0956796813000257 Published online by Cambridge University Press

submissions from biological sciences this year, with five talks covering scientific computing from very different angles: data visualization, data processing, and complete systems for cutting-edge research in sectors such as genomics and systems biology.

We would like to thank Simon Thompson and Duncan Coutts for helping to organize CUFP tutorials – which covered Erlang, F#, Haskell, Clojure, Scala, and OCaml, and were well-attended – and Ashish Agarwal for organizing evening BoF sessions. We thank Kresten Krab Thorup and Yaron Minsky for providing notes on their talks. Also, we thank Peter Thiemann and Fritz Henglein and the whole ICFP/CUFP team for their assistance in Copenhagen.

References

Balat, V., Vouillon, J. & Yakobowski, B. (2009) Experience report: Ocsigen, a web programming framework. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. New York, NY: ACM, pp. 311–316.

Benque, D., Bourton, S., Cockerton, C., Cook, B., Fisher, J., Ishtiaq, S., Piterman, N., Taylor, A. & Vardi, M. Y. (2012) BMA: Visual tool for modeling and analyzing biological networks. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV '12)*. Berlin, Germany: Springer-Verlag, pp. 686–692.

Elliott, C. M. (2009) Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*. New York, NY: ACM, pp. 25–36.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley.

López, M., Castro, L. M. & Cabrero, D. (2012a) Declarative distributed advertisement system for iDTV: An industrial experience. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming. (PPDP '12)*. New York, NY: ACM, pp. 185–194.

López, M., Castro, L. M. & Cabrero, D. (2012b) Failover and takeover contingency mechanisms for network partition and node failure. In *Proceedings of the 11th ACM SIGPLAN Workshop on Erlang (Erlang '12*). New York, NY: ACM, pp. 51–60.

McCabe, F. & Clark, K. (1995) April – Agent Process Interaction Language. In *Intelligent Agents, Lecture Notes on Artificial Intelligence*, Jennings, N. & Wolldridge, M. (eds), vol. 890. Berlin, Germany: Springer-Verlag, pp. 324–340.

Ondov, B. D., Bergman, N. H. & Phillippy, A. M. (2011) Interactive metagenomic visualization in a web browser. *BMC Bioinformatics* **12**(385).

Rendel, T. & Ostermann, K. (2010) Invertible syntax descriptions: Unifying parsing and pretty printing. In *Proceedings of the 3rd ACM Haskell Symposium on Haskell (Haskell '10)*. New York, NY: ACM, pp. 1–12.

Reppy, J. H. (1999) *Concurrent Programming in ML*. Cambridge, UK: Cambridge University Press.

Searle, J. (1969) *Speech Acts: An Essay in the Philosophy of Language*. Cambridge, UK: Cambridge University Press.

Syme, D., Petricek, T. & Lomov, D. (2011) The F# asynchronous programming model. In *Proceedings of the 13th International Conference on Practical Aspects of Declarative Languages (PADL '11)*. Berlin, Germany: Springer-Verlag, pp. 175–189.

Vouillon, J. (2008) Lwt: A cooperative thread library. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML (ML '08)*. New York, NY: ACM, pp. 3–12.

Wadler, P. & Blott, S. (1989) How to make Ad-Hoc polymorphism less Ad-Hoc. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*. Austin, TX: ACM Press, pp. 60–76.