Under consideration for publication in J. Functional Programming

CUFP'13 Scribe's Report

MARIUS ERIKSEN
Twitter, Inc.
1355 Market Street, Suite 900
San Francisco, CA 94103, USA.
MICHAEL SPERBER
Active Group GmbH, Hornbergstraße 49
70794 Filderstadt, Germany
ANIL MADHAVAPEDDY
Computer Laboratory, University of Cambridge
15 JJ Thomson Avenue, Cambridge CB3 0FD, UK

1 Overview

The Commercial Users of Functional Programming workshop (CUFP) is an annual workshop held in association with the International Conference on Functional Programming (ICFP). The aim of the CUFP workshops is to publicize the use of functional programming in commercial ventures. Its motto is "functional programming as a means, not an end."

This paper summarizes the presentation of the 2013 event, which took place in Boston, Massachusetts, continuing the tradition of the *Journal* to report on the presentations on a yearly basis (Sperber & Madhavapeddy, 2013; Madhavapeddy *et al.*, 2012). It sketches the essence of each of the presentations. Curious readers may wish to peruse the recorded videos from the workshop.[1]

The 2013 version of CUFP present a record number of talks, a reflection of the growing popularity of functional programming. The talks also covered a wide variety of topics, ranging from implementing systems for serving advertisement in Erlang to medical device automation in Scheme.

2 Keynote

Dave Thomas delivered the 2013 keynote, entitled "21s Century Crusades of Knights of the Lambda Calculus—Lessons from Past Language Crusades." As former CEO of OTI, the company responsible for the creation of the Eclipse IDE, he talked about his experiences with the "language wars" of decades past, dealing mostly with the introduction of object-oriented programming into the mainstream as well as the advocacy of logic programming and expert systems. Thomas sprinkled his talk with anecdotes and history lessons, mixing personal opinions with history-based advice to the the functional programming community.

Thomas joked that his qualifications included his proclivity to "infect" organizations with new technology which they had no intention of actually using. The only reason he

[1] Available at `http://cufp.org/2013/`.

was able to do so was that "it worked." He was careful to introduce new technologies only when they helped solve actual problems and when he could find a team that would make full, productive use of the newly introduced technology.

Next, Thomas suggested that functional programming was experiencing a surge in popularity because of multicore computing and "big data" applications. He warned, though, that the goal of a community should be to experience *modest* success as massive industrial successes usually creates a "really ugly winner," attendant with lots of "really ugly code." Thomas proceeded to deliver a laundry list of important considerations for a language to succeed commercially.

Language interoperability A new language must be able to make use of prior work. Most companies own a large legacy code basis. New technologies cannot ignore this code base and operate in isolation.

For example, Tektronix[2] was the first company to commercially deploy Smalltalk for its oscilloscopes. The project integrated technologies from several "cultures," including hardware, firmware, systems software, and application software. It succeeded only because the Smalltalk implementation was *designed to* interoperate in this manner.

Have an end-to-end story A software engineer must be able to diagnose a system from one end to the other. This aspect is especially important when solving "space and time problems." Sophisticated JIT run-times, such as Oracle's HotSpot, are antipatterns in this context. They make in nearly impossible to understand the run-time behavior of a system.

Serialization; data and code portability Serialization has taken a prominent role in the modern computing environment. The increasingly distributed nature of most systems necessitates good serialization. Most of these problems have been solved, and thus new systems should use prior art. Modern systems need tools that can extract data from the computing environment; new languages must have fine-grained control of where this data lives. Distribution also demands easy *deployment*; a simple way to create deployable executables is a requirement for a modern language.

Performance "Computers like rectangles." Language designers must put care and thought into the handling of arrays and their implementation. Doing so solves many problems and avoids the need for fancy optimization. Modern architectures rely on caching for performance, thus control over data locality is paramount as critical parts of your application must be kept in cache.

"Stacks are easy to put together but hard to make fast." Instead, implementors should use a register model for a virtual machine. Virtual machines also need intrinsics to take advantage of modern hardware architectures. Intrinsics help future-proof virtual machines because hardware changes rapidly.

Adoption Thomas's last point concerns perception. He claims that functional languages can *make users feel stupid*. While dealing with the already-daunting task of understanding

[2] See `http://www.tek.com/`

foreign ways of solving problems, newcomers must understand comprehensions, folds, and monads, which alienates them. More generally, functional languages tend to erect high barriers to entry.

Conversely, if the functional programming community wishes to see its programming languages adopted, it needs to pay special attention to this problem by focusing on affordances and education. It needs to empathize with users and understand that they change at different rates. A recent example is Haskell's mode for running *wrong* programs. Thomas considers this step a healthy development.

Finally, much of a typical functional language gets in the way of the development of "CRUD (create-read-update-delete) apps," which is the kind of application most of the world's developers are busy writing. Thomas's point is that functional programming languages ought to be smaller and more focused than they are now. "What we need is a collection of little right languages instead of a smaller collection of partly right and partly wrong kitchen-sink languages."

3 Analyzing PHP Statically

Julien Verlaguet (Facebook) presented Hack (Verlaguet & Menghrajani, 2014), a statically typed dialect of PHP.

Hack runs on Facebook's PHP virtual machine, HHVM; its compiler is implemented in OCaml. Much of the HHVM team's efforts are focused on this "developer experience," with a special emphasis on a rapid feedback cycle. In particular, the goal is to waste no time between saving a source file and having the results show up on the screen.

Due to the large size of Facebook's deployed software, performance is critical. Even a small performance improvement, say improving CPU utilization by 1%, can have a significant cost impact. PHP has been difficult to optimize due to its dynamic nature, requiring an ever-more sophisticated virtual machine.

To address this gap between PHP and feasible optimizations, Hack is statically typed yet compatible with PHP. Indeed, Hack code can interoperate with legacy PHP code without imposing any run-time penalty. Thus, developers can adopt Hack on an incremental basis, similar to gradually typed language though without their safety (Siek & Taha, 2006; Tobin-Hochstadt & Felleisen, 2006). Hack also comes with type inference, minimizing notational overhead, and further aiding adoption.

Hack's type system is interesting in its own right. It handles a large number of idioms from the dynamically-typed world. At the same time, it provides sufficiently strong static guarantees to catch a significant number of errors during type checking.

Facebook's Hack comes with a web-based integrated development environment (IDE). In order to provide auto-completion and code-navigation features in the IDE, Hack's type checker is compiled to JavaScript via `js_of_ocaml` (Vouillon & Balat, 2014) and runs alongside the IDE in the web browser.

In keeping with the stated goals, the response time from the compiler is nearly instantaneous. The type checker processes even thousands of files within a second, which makes for an impressive live demo. The Hack compiler uses a number of resident background processes. A master process delegates work to a number of child processes, which communicate via shared memory in a lock-free fashion. This architecture allows Hack to eagerly

type check files before a user could even type the requisite commands. This is particularly important for large changes, for example, when a developer switches git branches.

A vast amount of Hack's code is in OCaml. The latter is ideal for symbolic computation, has excellent performance and can be compiled to JavaScript. The main challenge with the choice of OCaml is the lack of native multicore support. To address this gap, the Facebook team engineered its own multiprocess architecture.

4 OpenX and Erlang Ads

Anthony Molinaro (OpenX) shared his perspective on his employer's ad exchange technology. The advertising technology company developed its original software in PHP, but it transitioned to Erlang. As the company's platform grew, the weaknesses of its software became quite apparent. In addition to architectural issues—a poor choice of databases, the lack of HTTP load balancing—the application run-time grew to be an expensive aspect of OpenX's day-to-day operations. Additionally, OpenX wanted improved support for concurrent and low latency operations.

Molinaro decided that Erlang was a good fit for the problem space of highly concurrent systems with soft real-time requirements. He prototyped a first implementation within a few months. This experience left him quite satisfied and he started to evangelize it to his coworkers. In response, the engineering team began to find additional projects that were suitable for Erlang:

- OpenX moved from the Cassandra data-management stack to the Erlang-based Riak.
- The developers added Erlang-based services for various elements of their stack.
- They also created a DSL for selecting ads so that the application logic could be interpreted by both Erlang and Java-based systems.
- OpenX implemented a data-service layer, abstracting the database.
- Erlang Solutions wrote an API router for OpenX.

OpenX currently has around 15 services written in Erlang, around 8 in Java, and a mix of Python and PHP for front-end tasks.

Molinaro emphasized the architectural choices that enabled the introduction of Erlang:

- OpenX's cloud-based systems use generic hardware, automate bootstrapping, are package-oriented and fault-tolerant.
- OpenX's software infrastructure use many cross-language tools. Thrift (Slee *et al.*, 2007), Protocol Buffers (Google Inc, 2014), and the Light Weight Event System[3] all contribute to a language-agnostic environment. An in-house system called "Framework" provides scaffolding or code layouts and provides support for building deployable packages from code. Framework also enforces versioning and reproducibility across languages.
- OpenX's architecture is service-oriented. Every component has a single purpose; overall, the components are loosely coupled.

While architectural choices enabled the use of Erlang, Malinaro reiterated that it was important to find a project with which to showcase the technology.

[3] `http://www.lwes.org/`

5 Redesigning the Computer for Security

Tom Hawkins (BAE Systems) spoke to us about the Darpa-funded SAFE project,[4]. The SAFE project aimed to co-design

- a new application language, Breeze;
- a new systems programming language, Tempest;
- a new operating system; and
- a new processor.

The goal of SAFE was "security at every level," for defense in depth. SAFE focused on hardware-enforced security, considering dynamic checking in software as too expensive.

The SAFE model requires fine-grained information flow control, implemented in hardware. Words of data in SAFE are called *atoms* comprising 64 bits of meta-data and a 64-bit payload. Atom meta-data contains type and label information, used by the hardware to perform access and integrity checks. Every bit of data in SAFE belongs to an atom; meta-data can always be recovered.

The assembly language is quite traditional, with a few high-level constructs to make programming the SAFE architecture convenient. It is implemented as an EDSL in Haskell, using the host language as a macro language. A monad captures the program description.

Tempest is an imperative language with automatic register allocation, and an optimizing compiler. It, too, uses the SAFE EDSL assembler as a backend. Tempest is itself an EDSL in Haskell. This arrangement makes it straightforward to in-line SAFE assembler.

Hawkins concluded with a handful of lessons:

- Designing a higher-order language with information flow control is hard.
- Starting with the systems programming language is better than with an assembler. While a valuable building block, programming directly in assembly language is unproductive. Furthermore, having a higher-level language isolates the software and hardware teams. The hardware team can change the instruction-set architecture without rewriting all the software, and the programming language team does not have to anticipate all hardware features.
- EDSLs are excellent for bootstrapping language-intensive systems. Most importantly, they are also highly reusable components.
- EDSLs demand that engineers are comfortable with the host language. They are also more difficult to debug than programs in an external DSL.
- Concrete syntax remains important. It is probably best to switch from abstract syntax to concrete syntax when a language gains modularity, because then the switch can be made without disruption.

(Finally, Hawkins jokingly declared that the optimal number of PL researchers on any given project is somewhere between two and seven.)

[4] `http://crash-safe.org/`

6 End to End Reactive Programming

Jafar Hussain of Netflix discussed the company's use of functional programming techniques, in particular, reactive programming. Netflix enabled its move toward reactive programming when its development team modularized the company's software stack. The middle tier and UI used to be highly coupled, causing numerous problems, especially inefficient calls between components.

Netflix developers roughly fell into two groups: "Cloud people" and "UI people." Due of its massive scale, the company decided that all developers had to "think at scale." Hence any switch posed the challenge of how to get UI engineers to think at scale. Netflix answered the question by providing "UI comforts," a reactive API.

The next challenge was exploiting parallelism. Hussain stated that this parallelism had to come easily, because no developer could be trusted with locks. He went on to describe the `Observable` monad, the central data structure used to compose Netflix's sub-systems.

`Observable` is part of Netflix's Java port of Rx.NET (Christensen & Husain, 2013). Briefly, it is is a vector version of the `Continuation` monad with the null-propagation semantics of the `Maybe` monad and the error propagation semantics of the `Either` monad. It is composed in a functional fashion, and has clean cancellation semantics. `Observable` can be used in either synchronous or asynchronous settings.

At Netflix, `Observable` is used as the singular data structure for both cloud and UI developers. It is applied with ease to problems in either setting, and it provides a uniform API around which applications are structured. The video of Hussain's talk includes two demonstrations of how the API smooths over the differences between cloud and UI programming, with a uniform structure for both. The first example shows an implementation of social notifications, a classic "long polling" example. The second example is an autocomplete system for search. The interested reader is referred to the video for details.

Like other speakers, Hussain emphasized the need for evangelism. He recommended practicing public speaking and a particular focus on "soft skills." In his own experience, he honed his speaking skills in a short time, and he was then able to clearly articulate the benefits and importance of a particular technology to his target audience. Hussain's group also developed interactive training exercises that illustrated the benefits of reactive programming. These exercises helped colleagues develop an intuition of how to use reactive constructs. During the process, the group's members were almost always available to help.

Netflix has turned its Java and JavaScript libraries for reactive programming into open-source projects.[5]

7 Medical Device Automation using Message-Passing Concurrency in Scheme

Vishesh Panchal (Beckman Coulter, Inc.) gave a talk on his team's use of Scheme to automate a molecular diagnostic device. His talk included a video of such a device playing a piece of music by actuating its motors.

In general, the purpose of a molecular diagnostic device is to detect the presence of specific strands of DNA/RNA in a sample. It is a complex machine, complete with temperature

[5] `https://github.com/Netflix`

and motor control, sensors, barcode readers and a spectrometer, plus a total of 19 mother boards. Operators use a thin, stateless client to interface with the device.

The server software is written in Scheme. It implements Erlang's message-passing concurrency model. This library can create, inspect, and update Erlang-style records. It also provides Erlang's supervisor structure to isolate hardware failures. Based on the library, the instrument server is decomposed into several processes, all arranged in a supervisor tree. These processes communicate by passing messages. The server also has an event manager that logs events and handles subscriptions to event streams. The use of message passing renders several common run-time errors impossible; others are mitigated by the use of supervisor trees, enabling principled handling of such failures at every layer in the software stack.

Scientists program the device with a EDSL hosted in Scheme. The language's implementation greatly benefits from Scheme's hygenic macros, first-class functions, and continuations. In addition, Scheme's arbitrary precision arithmetic was important for the numerous numeric computations needed.

The DSL greatly increases the flexible use of the diagnostic devices. Since scientists are the ultimate users of this DSL, not programmers, the language is highly specialized; it provides constructs for specifying high-level goals that are familiar to scientists. The video presentation includes a number of examples that demonstrate the direct translation of a scientific process into programs in this DSL.

Panchal conclusion consists of the following set of lessons:

- Message passing is a useful model for reasoning about the semantics of systems. Erlang/OTP's fault isolation leads to concise programs, employing only a small amount of defensive code. Concision often implies ease of testing. It is not a silver bullet, however, especially when dealing with concurrency and concurrency bugs. For these concerns, it remains crucial to use timeouts and supervisor trees to detect problems.
- Supervisor trees come with little "prior art." Developers are on their own.
- Automated unit testing is crucial throughout the process.
- Hiring into this unique environment remains difficult.
- Existing quality metrics (e.g. bug density) do not carry over to languages such as Scheme due to the terseness of the programs.
- Gluing together components written in different DSLs allows programmers to mix the good bits from both Scheme and Erlang at will.
- DSLs also enable rapid prototyping by non-expert programmers.

Finally it is noteworthy that the software also passes the FDA's scrutiny.[6]

8 Enabling Micro-service Architectures with Scala

Kevin Scaldeferri (Gilt Groupe) reported on the experience of building a large system from a large number of small services.

[6] The US Food and Drug Administration (FDA) is the regulating body for medical devices in the United States.

The Gilt Groupe is an Internet clothing retailer employing highly user-specific targeting. The company employs several schemes to drive sales, mostly centered around time- and quantity-limited offerings, which renders their web traffic rather uneven, with massive spikes around sales periods. By implication, their revenue is distributed along the same spikes and valleys, meaning stability during traffic spikes is an imperative.

The software system at Gilt used to be a largely monolithic Ruby-on-Rails application. Scaldeferri explained that, with a growing application, and a growing number of engineers, there were seemingly intractable software engineering challenges with this model. The setup also caused a number of production issues.

The team decided to switch to a "micro-service" architecture, splitting their application into a large number of well-defined, self-contained services. The transition began by factoring the Rails application into a few core infrastructure systems, using HTTP to communicate with each other. Within four years, these services numbered 300. Each micro-service was converted to Scala.

Scaldeferri outlined uses of "reactive" programming in this context, with many examples focusing on real-time updating. Such updates were constructed using Play (Typesafe Inc, 2014b) and Akka (Typesafe Inc, 2014a) actors.

During this transition, the engineering group developed several architectural components to support this large number of services in production:

Builds The group constructed plug-ins for Scala's Simple Build Tool (SBT) to abstract over build, configuration, and dependency management.

Configuration A ZooKeeper cluster stores configurations, which can be overridden locally. Configurations are deserialized into to Scala data structures with strict validation.

Testing Due to the complex set of dependencies testing remains challenging in micro-service architectures. Gilt's code base now uses the "cake pattern" extensively in testing to fully or partially satisfy dependencies that would otherwise be handled by another service in their production environment. The group uses Scala's traits, also known as mixins, to implement the cake pattern.

Delivery A key part of Gilt's deployment strategy is to deliver systems on a continuous basis. Twenty to thirty services are deployed automatically on a typical day.

9 Functional Infrastructures

Antoni Batchelli (PalletOps) described the Pallet platform for the automated generation of infrastructure software.[7]

Pallet allows users to write programs that build and operate computing environments, both locally and in the cloud. It provides abstractions to write *plans* that describe configuration actions independently of the target platform. It then translates those plans to shell scripts. While a shell script generated from a plan is specific to a particular target platform, the plan itself is target-independent. Pallet currently knows about several flavors of Linux and Unix systems.

[7] `http://palletops.com/`

The central entity in a Pallet configuration is a *plan function*, which is a pure function that generates a plan object. The plan object can be inspected, that is, the user may query the plan in various ways. For example, the user can find out what actions a plan would trigger on a particular platform.

Pallet also optimizes plans. For example, it can coalesce many small actions into a single large one if the target operating systems supports this optimization. The user can assemble plans into phases that run on servers. Plans can be abstracted over servers to instantiate entire families of similar installations.

The implementation of Pallet uses Clojure and exploits Clojure's multi-methods to specialize actions in plans. While the dynamic typing of Clojure is an enabling factor for many parts of Pallet, developers sometimes wish for static typing.

Batchelli concluded by noting that, over time, Pallet put a growing emphasis on data instead of functions to allow the inspection and manipulation of plan objects.

10 Realtime Map/Reduce at Twitter

Sam Ritchie (Twitter, Inc.) described Summingbird, a new open-source system for computing aggregates in real time.

Summingbird is a declarative, Scala-hosted EDSL for expressing map/reduce-style aggregates over streaming data. It bridges the gap between streaming and batch computation, enabling developers to write logic once and deploy it in a combination of batch- and streaming-computation systems. An important goal of Summingbird is to improve developer productivity by solving the systems problems in one place, so that the run-time handles efficient execution as well as scaling resources usage up and down based on need.

Summingbird's core operation is an "associative plus" operation, the Monoid. Its underlying data structure is practical for aggregation. The associativity of Monoids makes computations parallelizable in a straightforward way. According to Ritchie, many common data structures and aggregations that are Monoids, including sets, lists, maps, hyper-log logs, Bloom filters, moments, count-min sketch, and more.

It is common to deploy Summingbird in a dual batch/real-time configuration. The batch portion, working off of log files or ground truth data, computes the aggregate up to a given time stamp; a real-time streaming system maintains a sliding window of the same aggregate. Clients query both of these stores, merging the results. This style of deployment is desirable because it cleanly separates two concerns: the batch system aggregates over the entire data set, optimizing for throughput and the streaming system has a much smaller fixed window of computation, computing updates with lower latency.

Ritchie's central example concerns tweets. When Twitter displays a tweet, it shows a list of web sites that embed this tweet, an act that is based on impression data. The list is ordered by popularity. The Summingbird implementation consumes events of the form `(TweetId, (URL, Count))`. The event denotes three facts: the tweet `TweetId` is reachable `Count` times. This fits neatly into the model of Summingbird, as these tuples are trivially summable. To deal with the large number of web sites, Twitter uses a Count-Min sketch to reduce the memory requirements for keeping the counts.

11 Functional Probabilistic Programming

Avi Pfeffer (Charles River Analytics) introduced the *Figaro* language for probabilistic programming (Pfeffer, 2009).

The aim of functional probabilistic programming is to "democratize" building probabilistic models. A motivating example imagines that someone has some information and wants to derive answers from this information, keeping track of the uncertainty of the answers. The solution is to create a joint probability distribution over the variables, assert the evidence, and probabilistically infer the answer.

A common approach to probabilistic functional programming uses generative models. Variables are generated in order such that later values may bind (depend on) previous values. Developing such a model is not a simple task and is an active area of research.

Expressions in functional probabilistic programming languages are computations that produce values with uncertainty. Consider the following example from Pfeffer's talk:

```
let student = true in
let programmer = student in
let pizza = student && programmer in
(student, programmer, pizza)

let student = flip 0.7 in
let programmer = if student flip(0.2) else flip(0.1) in
let pizza =
  if student && programmer
    flip(0.9)
  else
    flip(0.3) in
(student, programmer, pizza)
```

Such programs are best understood via a *sampling semantics*. That is, the program is run many times. Each outcome has some probability of being generated, the program thus defines a probability distribution over outcomes. Since the language itself is Turing complete, it is capable of expressing a wide range of models.

Figaro's central data type is called `Element[T]`, i.e., the class of probabilistic models over type `T`. `Element[T]`s may be stochastic or non-stochastic. A number of atomic elements are defined, e.g. `Constant`, `Flip`, `Uniform`, and `Geometric`. These data types are combined to form compound elements. For example, the compound element, `If(Flip(0.9, Uniform(0, 10), Normal(1.0, 0.3))` is the uniform distribution from 0–10 90% of the time, and the normal distribution with mean 1.0 and a standard deviation of 0.3 the remainder of the time.

Figaro uses a probability monad to track state, with `Constant` representing the monadic unit, and `Chain(T, U)` the monadic bind. Most of Figaro's elements are implemented in terms of this monad.

Figaro is an Scala-hosted EDSL that allows for distributions over any data type. It has an expressive constraint system, and it comes an extensible library of inference algorithms containing many popular algorithms. Using its host language, Figaro can be used as a library with any JVM-based programming language.

Figaro is an open-source project (Charles River Analytics, 2014).

12 Building a commercial development platform Haskell

Gregg Lebovitz (FP Complete) reported on the FP Haskell Center, a web-based IDE for Haskell.

FP Complete aims to improve Haskell adoption and support the Haskell community in the process. Its primary goal is to make Haskell accessible to ordinary developers via educational materials and tools. In addition to free tools, they offer "commercial grade" development tools.

The FP Haskell Center greatly simplifies using Haskell. It is a ready-to-use integrated development environment for Haskell. The IDE consists of

- a web front end;
- a Haskell back end, implementing project management;
- integration with the compiler to achieve instant developer feedback, mostly in the form of error reporting;
- a help and documentation system;
- git-based version control;
- a build system; and
- an execution and deployment platform.

The IDE is itself built almost entirely in Haskell, using libraries and frameworks, available on Hackage. The front end uses Yesod (Snoyman, 2012) and Fay,[8] a proper subset of Haskell that compiles to JavaScript. The back end continuously precompiles the user's code so that the bytecode can be run instantaneously within the IDE.

13 Common Pitfalls of Functional Programming and How to Avoid Them: A Mobile Gaming Platform Case Study

Yasuaki Takebe (GREE, Inc.) spoke about the use of functional programming in a large, mobile gaming platform (37 million users, mostly in Japan, 2000 games, 2600 employees). Historically, his company built mobile games in web programming languages such as PHP or Ruby. Recently, GREE began using Haskell for some of its back-end systems. Takebe described one of these projects, a management system for their in-house key-value storage system.

The systems task is to manage and scale capacity in the company's storage clusters. It might, for example, increase the cluster size due to hardware faults or to access spikes.

Takebe presented a few Haskell-specific implementation issues:

memory leaks due to lazy evaluation The front-end server kept a list of active threads in a `TVar` for monitoring purposes. Operations to remove threads from this list were evaluated lazily, and could thus create a large memory leak.

[8] `https://github.com/faylang/fay/wiki`

race conditions A race condition between dequeueing items and an asynchronous exception thrown by a timeout handler caused the loss of data.

performance degradation The GREE team used the `http-conduit` library to perform health checks of various servers. In a minor version update, this library started to fork new threads for each http request, leaving the caller with the responsibility to perform explicit resource management. As a result, the program ran with as many threads as there had been health checks, causing a resource leak.

The GREE team decided to improve their testing practices to battle these problems. Using standard Haskell tools, especially QuickCheck (Claessen & Hughes, 2011) and HUnit (Herington, 2014), they developed a harness within which they could start their servers and test them in-situ. They developed over 150 systems tests with more than 5,000 assertions.

Next they started to document the issues they ran into in order to share their experiences and prevent future mistakes of the same kind. While they put significant effort into this process, few developers bothered reading them. In response, the team started enforcing a useful subset of the rules via HLint (Mitchell, 2014).

Finally, the group focused on proactive education. They set up a brown-bag lunch where they covered Haskell and Scala topics. They also ran a class for new graduates in which students solved problems from project Euler in Haskell.

Takebe reported that some of GREE's major software component had been converted to Haskell and considered the project a success. He urged attendees to pay particular attention to the "superstructure" of a language: its community, the documentation, and the tool chain. Takebe expressed his belief that these aspects were the sine qua non of introducing FP in a setting such as GREE.

14 Building scalable, high-availability distributed systems in Haskell

Jeff Epstein (Parallel Scientific) spoke about the use of Haskell in a high-availability (HA) distributed system for managing resources in a large (10k+ nodes) cluster manager. For undisclosed reasons, the team determined that existing solutions, ZooKeeper among them, were not up to the task. They therefore set out to build their own implementation of Paxos (Lamport, 1998) in Haskell.

The job of a cluster manager is to present a consistent view of the cluster's state and to recover from failures quickly. While the Haskell implementation employs purely functional data structures employed—the state of a cluster, for example, is represented by a purely functional graph—the code itself has an imperative appearance because of the inherently imperative nature of the domain.

The code uses Cloud Haskell (Epstein *et al.*, 2011) for distribution management. Cloud Haskell is a actor-style message-passing system, similar to Erlang. It is a particularly good fit for this particular project as its model of independent, communicating processes meshes well with Paxos.

The Haskell implementation of Paxos is a general purpose library on top of Cloud Haskell. Each component of the algorithm—the client, acceptor, proposer, learner, and leader—are Cloud Haskell processes. Their implementation consist of about 1.5kLOC

of Haskell, closely matching the pseudo-code in Lamport's original paper. This kind of near-transliteration increases the confidence in the implementation's correctness. The team is now working on adopting modifications from the literature (Chandra *et al.*, 2007) to improve the library's performance.

As has often been the case, Haskell's lazy evaluation poses problems. In this case, the lazy evaluation caused space leaks in low-level networking code. Distribution is a natural barrier to laziness since messages must be serialized across process boundaries. In contrast, Haskell's strong typing is a great aid for re-factoring tasks. Also, since Cloud Haskell provides a platform for distribution, the language makes it easy to develop and debug distributed systems on a single machine. Since such systems generally, and Paxos especially, is sensitive to non-determinism, it is imperative to use a deterministic scheduler during development, which allows to test the system in a reliable and meaningful manner, with the possibility of reproducing errors in a deterministic fashion.

15 IQ: Functional Reporting

Edward Kmett (S&P Capital) discussed the company's use of functional programming. He started with the introduction of Scala and followed up with a presentation of *Ermine* (Compall, 2014), a new Haskell-like language for their domain.

Kmett's team used Scala and FP techniques in a "portfolio analytics" engine, a product used for performance and risk attribution across financial portfolios. The old version of the portfolio analytics engine was written in Java. It was hard to extend, and required all data to be in memory. They rewrote this code in Scala and introduced monoids for simple parallelization, solving both the performance and extensibility problem. Reducers were used to derive structure from containers, in a parallel manner across monoidal structures. Kmett noted how this rewrite removed all the obvious "wiring" from their code. This project really helped sell the use of Scala and, more generally, FP to the rest of the company.

Ermine is a Haskell-like language, developed to build a generic reporting and visualization framework. It is a JVM-based language and will be used in multiple products. Like the portfolio analysis engine, Ermine is a response to problems with a prior implementation—this one in Scala. Specifically, writing monadic code in Scala can be quite painful as it is easy to overflow the stack without trampolining.

Ermine has a Haskell-like type system, but with row types, constraint kinds, and rank-N types. It also provides a built-in database support sub-system. Row types are useful in this context as they provide a powerful mechanism to describe the structure of data. The type system models constraints with "has", "lacks" and "subsumes".

To support development, Ermine comes with a structured code editor; the editor prevents programmers from creating code with type errors. In addition, the language now has a declarative reporting layer that can push reports into various back ends.

16 Enterprise scheduling with Haskell at skedge.me

Ryan Trinkle (skedge.me) presented skedge.me's cloud-based scheduling platform.

The company's software handles complex "enterprise" scheduling for, among others, retailers. For example, Sephora, a makeup company, uses skedge.me to schedule appoint-

ments with customers. The skedge.me software is integrated into Sephora's site via an iframe; it is styled seamlessly to look like a part of the host site.

Originally, skedge.me consisted of 43,000 lines of code in Groovy on Rails. The codebase had several major intractable issues: timezone problems, double bookings, recurring events not firing notifications, and delayed notifications. The application also had severe performance issues. Worst of all, the application was inflexible. It was nearly impossible to respond to customers' requests; on occasion, they had to ask their customers to change the way they conducted business to combine a skedge.me model with the host site.

After careful deliberation, the team decided to rewrite skedge.me in Haskell. Trinkle had worked with Haskell previously, though not to build a web site. The team began by constructing a monad, `RawDB`, to maintain ACID (Atomicity, Consistency, Isolation, Durability) guarantees during transactions.

The revised system consists of three layers. The `RawDB` layer tracks effects and can automatically retry operations on temporary failure. The `DB` monad is built on top of `RawDB` monad and provides a high-level "CRUD" interface. This layer makes heavy use of algebraic data types, and performs caching and validation. The final layer before the application code is the security layer. It implements security policies for various customers. Implementing security turns out to be tricky due to the myriad ways in which the product can be configured by customers. For example, they may define roles (e.g. "owner," "staff," "customer") that make sense only within their environment; the *verbs* of the product, too, may be configured (e.g. appointments may be joined or rescheduled) depending on the environment. Thus both sides of the security equation—nouns and verbs—can change from instance to instance. The skedge.me software uses Haskell's type class facility to model these security policies. These help map customer-specific customizations into a standard schema that can be manipulated on a component-by-component basis. This technique affords the team a great deal of static guarantees from type checking, a critical property when implementing security sensitive systems.

The team's code base makes heavy use of Hackage, linking in 71 unique libraries from the repository. An additional 87 are brought in from the transitive dependency graph. The team considers Hackage to be a well organized repository. Because most libraries are purely functional in nature, they are easy to vet for quality.

Trinkle finally noted that, while Haskell provided a great platform for "building code for the long run," the team also made good use of the language for quick-and-dirty hacks, e.g., for importing data from the old system. Even for such hacks, however, Trickle emphasized the strong support from the type system. Concerning libraries, Trinkle's team replaced just one library due to bugs. While Haskell might have fewer libraries when compared to other, more popular languages, those that are available, are of higher quality.

17 Wolfram: Programming Map/Reduce in Mathematica

Paul-Jean Letourneau (Wolfram) closed this year's CUFP with his talk on implementing MapReduce in Mathematica. Specifically, Letourneau described HadoopLink (Letourneau, 2013), an integration of Mathematica and Hadoop.

In Mathematica, everything is an expression. Expressions are rewritten until the process reaches a fixed point. Expressions are also data structures, similar to Lisp's S-expressions.

As such Mathematica follows a familiar LISP mantra of "programs are data." Homoiconicity abounds, which allows a Mathematica program to manipulate expressions, e.g., to perform rebinding, which is powerful for distribution. Letourneau's video presents some examples of impressively short Mathematica programs. Following Theo Gray, "everything is a one-liner in Mathematica ... for a sufficiently long line", including an image constructed recursively. In short, Mathematica as "a gateway drug to declarative programming."

HadoopLink allows for nearly seamless distribution of Mathematica programs: mappers and reducers both are ordinary Mathematica functions, stitched together by a Hadoop link object for input and output. The programs can be defined on a single page; HadoopLink takes care of the rest.

Letourneau concluded his talk with the impressive example of a simple genome search engine. This problem lends itself particularly well to map/reduce style computation. The program, including large data sets, is definable within a single Mathematica session. Turning it into a distributed implementation is indeed a seamless process.

18 Conclusion

CUFP 2013 was a watershed event. It put an incredible breadth, depth, and broad applicability of functional programming on display. This was not only true of the papers presented at the workshop, but also of the many submissions rejected due to time constraints.

The program was rich in every dimension covered: techniques, languages, and industries. We had talks from Internet companies, the biotech industry, the medical device industry, gaming, and the financial industry. Languages in use varied from EDSLs, to academic stalwarts, to home-grown languages. It seems that language considerations have taken a front-seat in modern engineering practices.

As the numerous talks on *service oriented architectures* show, the process of decomposing a monolithic application into many, smaller services has been a fertile field for adopting functional programming. Teams use the transition as an opportunity to re-implement smaller parts of the system in new languages, which are better suited than the old ones. In other words, the strategy enables the use of functional programming an incremental and controlled manner. The advantages of functional programming come across in a small setting, without having to take large risks.

Finally, we would like to thank Simon Thompson and Francesco Cesarini for organizing this year's tutorials. Ashish Agarwal organized the evening BoF sessions. We would also like the to thank the ICFP organizers for their assistance in Boston. We also thank Matthias Felleisen for his editorial help creating this report, and David Sheets for scribing during the conference.

References

Chandra, Tushar D, Griesemer, Robert, & Redstone, Joshua. (2007). Paxos made live: an engineering perspective. *Pages 398–407 of: Proc. Twenty-sixth Annual Symposium on Principles of Distributed Computing*. ACM.

Charles River Analytics. 2014 (Sept.). *Figaro*. `https://www.cra.com/commercial-solutions/probabilistic-modeling-services.asp`.

Christensen, Ben, & Husain, Jafar. 2013 (Feb.). *Reactive programming in the Netflix API with RxJava*. `http://techblog.netflix.com/2013/02/rxjava-netflix-api.html`.

Claessen, Koen, & Hughes, John. (2011). Quickcheck: a lightweight tool for random testing of Haskell programs. *Proc. International Conference on Functional Programming*, **46**(4), 53–64.

Compall, Stephen. 2014 (Apr.). *A users guide to Ermine*.

Epstein, Jeff, Black, Andrew P, & Peyton-Jones, Simon. (2011). Towards Haskell in the cloud. *Pages 118–129 of: Proc. Haskell Symposium*, vol. 46. ACM.

Google Inc. 2014 (Sept.). *Protocol buffers*. `https://developers.google.com/protocol-buffers/`.

Herington, Dean. 2014 (Sept.). *HUnit 1.0 user's guide*. `http://www.haskell.org/haskellwiki/HUnit_1.0_User's_Guide`.

Lamport, Leslie. (1998). The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, **16**(2), 133–169.

Letourneau, Paul-Jean. 2013 (July). *Mathematica gets big data with HadoopLink*. `http://blog.wolfram.com/2013/07/31/mathematica-gets-bigdata-with-hadooplink/`.

Madhavapeddy, Anil, Minsky, Yaron, & Eriksen, Marius. (2012). CUFP 2011 workshop report. *Journal of Functional Programming*, **22**(1), 1–8.

Mitchell, Neil. 2014 (Sept.). *HLint manual*. `http://community.haskell.org/~ndm/darcs/hlint/hlint.htm`.

Pfeffer, Avi. (2009). Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, 137.

Siek, Jeremy G., & Taha, Walid. 2006 (September). Gradual typing for functional languages. *Pages 81–92 of: Scheme and Functional Programming Workshop, University of Chicago, Technical Report TR-2006-06*.

Slee, Mark, Agarwal, Aditya, & Kwiatkowski, Marc. (2007). *Thrift: Scalable cross-language services implementation*. `http://thrift.apache.org/static/files/thrift-20070401.pdf`.

Snoyman, Michael. (2012). *Developing web applications with Haskell and Yesod*. O'Reilly Media, Inc.

Sperber, Michael, & Madhavapeddy, Anil. (2013). Commercial users of functional programming workshop report. *Journal of Functional Programming*, **23**(11), 701–712.

Tobin-Hochstadt, Sam, & Felleisen, Matthias. (2006). Interlanguage migration: from scripts to programs. (Companion Dynamic Languages Symposium).

Typesafe Inc. 2014a (Sept.). *Akka documentation: Release 2.0.2*. `http://doc.akka.io/docs/akka/2.0.2/Akka.pdf`.

Typesafe Inc. 2014b (Sept.). *Play 2.2 documentation*.

Verlaguet, Julien, & Menghrajani, Alok. 2014 (Mar.). *Hack: a new programming language for HHVM*. `https://code.facebook.com/posts/264544830379293/hack-a-new-programming-language-for-hhvm/`.

Vouillon, Jérôme, & Balat, Vincent. (2014). From bytecode to JavaScript: the Js_of_ocaml compiler. *Software, Practice and Experience*, **44**(8), 951–972.