Research Statement

Heather Miller

We have come to expect most applications these days to push us just-in-time information and suggestions based on our location, interests, history etc. Such applications are a feat of distribution; they dutifully synchronize with a multitude of other services, and often request results obtained by clusters of machines churning through massive datasets.

Despite their growing prevalence, building distributed applications remains markedly error prone and is of inordinate cost in time and expertise. Consequently, building interesting distributed applications is unrealistic endeavor for even a modest slice of professional software engineers.

My research strives to simplify the task of developing distributed systems, and to improve the performance and quality of such systems.

*A major recurring theme in my work is **composability**. I seek to enable the construction of complex distributed systems via the composition of well-understood components that are correct by construction.*

This goal typically requires conceptual innovations and technical advances at the boundary of distributed systems and programming languages along the following three synergistic directions:

1. New programming models which aim to facilitate the development of composable, correct-by-construction distributed systems.
2. Type systems to avoid hard-to-debug classes of errors introduced by distribution, and type-directed optimizations which improve performance in the face of distribution.
3. Techniques for extending widely-used programming languages with new programming models and domain-specific static checking.

My research has been realized in step with the development of Scala, a programming language with humble academic beginnings that now ranks just behind Swift, and just above R and Perl in popularity [9]. As a result, many of my efforts find immediate practical uses in industry, at companies like Apple, The Guardian, Tesla, and many more.

My research approach has been to jointly optimize my solutions on both elegance and practicality. Elegance, for example, in that I try to apply methodologies prevalent in the programming languages community (namely, a focus on providing sound primitives on which to compose up complex systems) to solving some of the problems of the distributed systems community. Practicality in the sense that one of my core beliefs is that research ideas should transfer to reality–they should be prototyped and integrated into real languages and systems. To this aim, all of my projects are open source, and I am fortunate that several, such as Spores [5], Pickling [4], and Scala's Futures and Promises [3], have found widespread use, and are a part of a vibrant open source community.

Distributed Functional Systems

Functional programming (FP) is often touted as the way forward for bringing parallel, concurrent, and distributed programming to the mainstream. As evidence, consider popular big data frameworks such as MapReduce and Spark. Both leverage functional patterns, sending closures across the network to large, persistent data sets.

Despite the increasing popularity of FP as an enabling technology, practical uses such as these expose several critical problems at the level of the programming language, which my dissertation research tackles head-on. Namely, (1) distributing functional language features and (2) providing data representations suitable for today's breed of in-memory distributed systems. Concretely, my work makes three contributions: spores, pickling, and function-passing for typed distributed functional programming.

Spores [5] is my project which addresses the first problem; reliably distributing functions and closures, core functional language primitives. In order to distribute closures, one must be able to serialize them– a goal that remains tricky to reliably achieve not only in object-oriented languages but also in pure functional languages like Haskell.

Spores are an abstraction and type system that can guarantee closures to be serializable, thread-safe, or even have custom user-defined properties such as deep immutability. Based on the principle of encoding type information corresponding to captured variables in the type of a spore, a number of previously impossible opportunities are now possible. Types facilitate the verification of closure-heavy code; they open up the possibility for IDEs to assist in safe closure creation, advanced refactoring, and debugging support; and they make it possible for spores to integrate with pluggable type systems and type class-based frameworks like Pickling. My work showed that unchecked patterns for serializable closures are widespread in real Spark and Akka applications, and that benefiting from static guarantees provided by spores requires only little syntactic overhead. Spores have become a high in demand language feature for inclusion in a future release of Scala by way of a popular Scala Improvement Proposal [6].

Pickling [4] is my project which addresses the second problem; extensible, performant, and *open* ways to represent and exchange data, aimed at distribution. The performance and simplicity of exchanging data between compute nodes/devices is increasing in importance as more applications migrate to the cloud, and as "big data" edges into even more production environments. Yet, a central aspect to the efficiency and ease of communication between nodes which has received surprisingly little attention is the need to efficiently and flexibly transform one data representation to another, typically to transmit data over the wire, or to interact with some other system or service. Built-in serialization on the JVM has long been known to be prohibitively expensive to use in high-performance distributed applications, leading systems builders to choose to instead use higher-performance alternative frameworks such as Google's Protocol Buffers, Apache Avro, or Kryo. However, the higher efficiency typically comes at the cost of weaker or no type safety, a fixed serialization format, more restrictions placed on the objects to-be-serialized, or only rudimentary language integration.

Pickling strikes a sweet spot in this design space via a novel technique for generating and composing pickler combinators at compile-time. The result is high-performance, better type safety, no boilerplate, and unrivaled extensibility thanks to functional composability and a typeclass-based design based on Scala's implicits [7]. Most notably, users may swap in their own "pickle" format, consequently liberating the choice of data representation. My work showed Pickling to perform close to an order of magnitude over Java, and up to 4x faster than state-of-the-art industrial frameworks, while still managing to handle fundamental features of object-oriented languages such as subtyping polymorphism. Pickling has since grown into a popular project in the Scala community, finding widespread use.

Function-Passing [2] is a new programming model for typed distributed functional programming based on Pickling and Spores. Popular frameworks like Apache Spark leverage functional techniques to provide high-level, declarative APIs for in-memory data analytics, often outperforming traditional "big data" frameworks like Hadoop/MapReduce. However, some aspects of their programming models remain rather ad-hoc; static typing, formal semantics, and implementation trade-offs are not yet well-understood. As a research intern at Databricks, it quickly became evident that systems like Spark made use of strong static types typically only at interfaces. The weaker typing of other system layers often resulted in considerable debugging efforts, or lost opportunities for performance optimizations.

Function-Passing is a new programming model based on sending safe closures between immutable data "silos" that can be thought of as a generalization of the MapReduce/Spark model. Crucially, strong static typing throughout all layers of the system enables (a) performance optimizations and (b) static guarantees such as safe closure passing and serializability. My work shows that by merely toggling the use of static type information throughout the layers of a mini-Spark implementation amounts to a performance boost of 50% in some applications.

Asynchronous & Concurrent Systems

My thesis work has taken me on additional forays into different walks of parallel, concurrent, and asynchronous programming. Some efforts now power companies that are household names such as The Guardian, Apple, and Netflix. This section sketches these efforts and the essential results.

Futures & Promises in Scala [3] was a joint project between individuals at EPFL, Twitter, and Typesafe[1] to arrive at a functional and non-blocking variant of futures available in the Java-sphere. The key advance is an efficient non-blocking core coupled with a monadic interface for functionally composing operations, which in many cases eliminates the "callback hell" that plagues other languages. As a result, services based on these new futures smoothly scale from small to huge and complex applications. Futures & Promises have become the most widely used concurrency library in Scala, powering a vast number of high-traffic commercial services.

FlowPools [8] are a novel data-flow collection abstraction that is implemented atop a non-blocking, lock-free and provably deterministic data structure. They come with a functional interface, and are therefore composable via ordinary higher-order functions such as map and aggregate–this means one can concisely form data-flow graphs, in which associated functions are executed asynchronously as elements of FlowPools in the data-flow graph become available. My benchmarks show that FlowPools can perform one to two orders of magnitudes faster than their closest non-data-flow non-blocking concurrent counterparts available in Java's standard library.

Menthor [1] is a programming model and framework for parallel graph processing aimed at machine learning applications. One of Menthor's key strengths is its ability to accommodate iterative algorithms, without requiring the inversion of control that tends to plague most graph processing frameworks. We built numerous, real, nontrivial applications atop the framework, including maximum entropy classifiers, naive Bayes classifiers, expectation-maximization algorithms, and collaborative filtering algorithms, all of which achieved substantial speedups while written in a clear and declarative style. In standard graph processing benchmarks such as Pagerank, Menthor scales near linearly beyond 16 cores.

Where next?

In my future work, I plan to build on my experience in real-world programming languages for distributed systems. I plan to revisit my work on programming for data-centric distributed systems in new areas such as machine learning, and expand to other areas concerned with distribution, such as edge computing. Some areas I am interested in include:

Coordination-Free Primitives for Edge Computing: Edge computing is the shift of computation typically centralized in the cloud to devices at the *edge;* think smartphones, web browsers, or even things like VR headsets, for example. For these devices to remain responsive to user input (to achieve high availability), synchronization (or coordination) with other nodes in the system must be reduced as much as possible. However, this is an issue for applications that operate on data typically stored remotely in a strongly-consistent data stores–the coordination required to retrieve data from the store is typically too great of a sacrifice to the application's availability. Instead, in applications involving edge computation, data is typically replicated to and stored locally on these edge devices, with local updates to the data propagated throughout the network and merged at other devices containing replicas.

Recent work on a new kind of distributed data type known as Conflict-Free Replicated Data Types (CRDTs) enable conflict- and coordination-free merges of such updates to across replicas, while guaranteeing *eventual* consistency. That is, they guarantee that all replicas in the system will agree on the same value *eventually*. These data types are a great for edge computing applications.

However, so far, CRDTs and systems which have

[1] now known as Lightbend.

been proposed for coordination-free programming on replicated data aren't composable, that is, computations on many existing realizations of CRDTs (e.g., sets, registers, maps) cannot be easily composed together to build up richer, eventually-consistent replicated data types. Nor do they aid the user in any way when it comes to avoiding combinators or operators which violate the invariants of the model or data type, which causes users to easily and inadvertently lose consistency guarantees. Thus, only a precious few implementations of CRDTs exist in a handful of forms.

I would like to enable systems builders to more easily compose up complex schemas and data types for systems with replicated data. One thread of my ongoing work is on providing coordination-free primitives upon which to compositionally *build up* complex data types which guarantee eventual consistency. Beyond programming models that enable correct-by-construction coordination-free computations, static analysis or type and effect systems could also help out here in guaranteeing that required invariants hold. I plan to realize these primitives in the context of Scala, to assess their expressiveness via several case studies implementing existing distributed applications in terms of these primitives, and to evaluate their performance at scale across large clusters.

Programming Systems for Data Science: In all walks of programming, and especially in typed languages, reading in and making sense of large collections of unstructured or semistructured data is a challenge. The work on type providers in F# has shown how to generate typed interfaces for certain kinds of structured data at compile time. How could this work be generalized to semistructured or unstructured data? Going further, what about data with known probability distributions?

In many distributed systems, management and processing of data is central. In my work on integrating Scala Pickling into Apache Spark, it became clear that many applications could benefit from unconventional data access modes like partial deserialization of incoming data–that is, not every piece of may be required for each step of an algorithm. One exciting direction I would like to explore is the inference of typed interfaces for such semistructured or unstructured data to ease the burden of ingesting and cleaning data for Data Science applications. Going further, another direction I would like to explore is inferring probabilistically-typed interfaces for data in a data set with known probability distributions. Naturally, this changes how we look at persisted or serialized data, and the question arises how the principles behind Pickling could be generalized to support these patterns.

Typed Distributed Functional Systems:
A natural question to ask is which programming languages, models, and systems are best suited to build future distributed systems. If Function-Passing is a suitable foundation for big-data-style distributed systems, what other kinds of systems does it support effectively? Could Software-as-a-Service style applications benefit from the programming model? Could Function-Passing underlie mechanisms for new computing paradigms such as cloud computing?

Or perhaps more obvious; given the focus of moving computation to the edge in edge computing applications, with the addition of a sufficient gossip algorithm, could Function-Passing prove to be a useful model in the context of edge systems? I'd like explore the limits of this model of computation.

Asynchronous and Reactive Systems: While designing and standardizing Scala's Futures and Promises, one of the clearest needs from talking to users across industry is the declaration, propagation, and handling of asynchronous events such as cancellation. This has practical importance in real systems–at Twitter, for example, not being able to propagate cancellation signals meant that large and expensive jobs would continue running, unneeded, wasting resources. Enabling cancellation amounts to propagating signals backwards up directed acyclic graphs (DAGs) of computations on futures which may potentially span several disparate compute nodes.

I would like to develop an elegant and robust model of such asynchronous events that is straightforward and amenable to formalization, and which is accompanied by an extension or re-imagination of Scala's Futures and Promises. Such asynchronous events could also prove useful in the context of debugging tools for Scala's Futures and Promises or asynchronous programming more generally; e.g., debuggers which can *replay* DAGs of asynchronous computation.

Overall, my philosophy is to find fundamental problems in distributed systems and to seek elegant solutions that are practical for developers *now*, in many cases drawing on years of insight and experience from communities like programming languages. I would like to do for distributed systems community what abstraction and composition did in the context of programming languages; bring more reusability and reliability to aid in solutions to the fundamental problems in distributed systems.

References

[1] P. Haller and H. Miller. *Parallelizing Machine Learning–Functionally: A Framework and Abstractions for Parallel Graph Processing*. in Scala 2011.

[2] P. Haller, H. Miller and N. Müller. *A Programming Model and Foundation for Lineage-Based Distributed Computation*. Journal of Functional Programming, (to appear) 2018.

[3] P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn and V. Jovanovic. *Futures and Promises in Scala* Scala Documentation, 2012.

[4] H. Miller, P. Haller, E. Burmako and M. Odersky. *Instant Pickles: Generating Object-Oriented Pickler Combinators for Fast and Extensible Serialization*. in OOPSLA 2013.

[5] H. Miller, P. Haller and M. Odersky. *Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution*. in ECOOP 2014.

[6] H. Miller, M. Odersky and P. Haller *Spores*. Scala Improvement Process2013.

[7] M. Odersky, O. Blanvillain, F. Liu, A. Biboudis, H. Miller and S. Stucki. *Simplicitly: Foundations and Applications of Implicit Function Types*. in POPL 2018.

[8] A. Prokopec, H. Miller, T. Schlatter, P. Haller and M. Odersky. *FlowPools: A lock-free deterministic concurrent dataflow abstraction*, Languages and Compilers for Parallel Computing. Springer Berlin Heidelberg 2013.

[9] RedMonk. *The RedMonk Programming Language Rankings: June 2017*, http://redmonk.com/sogrady/2017/06/08/language-rankings-6-17/, 2017.