Research Statement

Philipp Haller

The software industry is undergoing a fundamental shift toward Software-as-a-Service making applications accessible from anywhere via any device. Web browsers, traditionally "thin" clients, have become powerful enough to host even standard-bearers like word processors. This shift is radically changing the technologies and skills needed by software engineers to build this new generation of cloud-hosted applications:

(1) The requirement of applications to scale to high volumes of users or data makes it indispensable to distribute work loads across large clusters. Practical programming models for this new cluster computing setting are still at an early stage, and often leave programmers no choice other than solving difficult concurrency problems using low-level abstractions, resulting in bugs that are hard to diagnose.
(2) The runtime platform for these new applications is fundamentally heterogeneous. While cluster-side code runs on "traditional" runtimes like the Java Virtual Machine, client-side code typically runs on markedly different execution engines for dynamic languages like JavaScript. However, enforcing safety and security properties is very difficult in distributed, multi-language applications.

This new cloud computing setting has been an ideal application area for much of my previous research. I have contributed to advances in both simpler, scalable concurrency and distribution, and programming languages for targeting heterogeneous platforms. My work has ranged from theoretical foundations to programming models and artifacts proven in wide production use. I believe the challenges of engineering reliable, scalable, and responsive systems and applications have the potential to change future programming technology in significant ways that will benefit not only cloud-hosted software.

With Scala Actors [7, 8] I integrated the high-level actor concurrency model into the Scala language, through an embedded domain-specific language (DSL). Besides serving as a basis for subsequent research of my own (*e.g.,* [3, 4]), the project has supported a substantial body of highly-visible research of others, *e.g.,* on formal analysis of Scala actor programs. As part of Scala's standard library Scala Actors have been proven in numerous production environments such as Twitter's core messaging system. The rich topic of concurrent programming gave me an opportunity to revisit foundational questions in type-and-effect systems (*e.g.,* how to statically reason about object references), resulting in a new uniqueness type system [6] based on static capabilities. The type system has been a major influence (discussed in [1]) on a variant of C# for systems programming in development at Microsoft, in particular with respect to side-effect tracking and isolation. At Typesafe I led the specification of Scala's futures, a new concurrency package at the core of frameworks like Play, powering the websites of companies like LinkedIn and the Guardian. I have presented this and further work on asynchronous programming [9] at leading industrial conferences such as Strange Loop.

Throughout these projects, my research approach has been (a) to identify a problem for which no satisfactory solution exists in mainstream software development (for example, the problem of statically enforcing the isolation of concurrent processes); (b) to search for solutions that are simple, elegant, reusable, and widely applicable; and (c) to try to generalize the problem as much as possible to increase the applicability of its solution. Moreover, I believe that practical, mature research results should be introduced in real programming languages and systems. Some of my projects have reached a level of maturity where this has been possible in the context of the mainline distribution of the Scala programming language. Other projects are more exploratory and require more research until truly practical solutions are discovered. I am also very interested in empirical studies to guide some of my future research. I believe carefully-designed empirical studies have the potential to significantly increase our understanding of the role of programming languages in practical software engineering; they might also help discover new real-world challenges of current programming technology.

Concurrent and Parallel Programming

Actors. As part of my thesis work I created Scala Actors [7, 8]. The system demonstrated for the first time how the actor model of concurrency can be integrated in mainstream technologies, in particular the Java Virtual Machine, in a way that scales to a very large number of lightweight concurrent processes. Scala Actors also demonstrated as one of the first systems how to provide an expressive, high-level programming interface for concurrency as a DSL embedded in a general-purpose programming language. Scala's original actor library has been part of the Scala mainline distribution since September 2006; during that time Scala actors have influenced a considerable amount of further research on actors, and the framework has also been proven in production environments such as Twitter's Kestrel message queue system which comfortably sustained record-high traffic on its website during Obama's first inauguration in 2008[1] amongst many others. From 2009, Jonas Bonér *et al.* have been working on a new design for a Scala actor framework, called Akka, as the foundation of a distributed, event-driven middleware. Akka's design and implementation have been influenced to a large extent by Scala Actors. Thus, my research on Scala Actors laid the groundwork for what is today Typesafe's main middleware for concurrent, event-driven applications that scale to multicores and clusters.

Joins. While actors provide a versatile concurrency model that scales to multicore processors and distributed systems, coordinating groups of actors remains a challenge. A certain class of coordination problems can be simplified using join patterns. Building on Scala's flexible pattern matching construct, I have devised a new implementation scheme for join patterns [3] that also integrates with actors.

Data-flow programming. I have also contributed to collection-like data-flow abstractions [10] with efficient, non-blocking implementations that are provably lock-free. This research shows that it is possible to leverage the high degree of parallelism of an execution model based on data flow while providing a familiar collection-style interface to the programmer.

Asynchronous programming. At Typesafe I have been co-leading the Scala Async project [9]. It introduces a way to suspend within a block of Scala code until a future, a placeholder for the result of an asynchronous computation, has been completed. This form of suspension avoids the drawbacks of programming in an event-driven style while enabling the use of efficient non-blocking concurrency abstractions under the hood. The novelty of our design is the fact that it does not require extending the Scala language, thus avoiding an increased language complexity, while enabling the same expressiveness as similar constructs of C# and F#. Scala Async has been proposed for inclusion in the Scala standard distribution. Building on Scala Async I have designed a new programming model [5] that unifies direct-style futures and asynchronous event streams, avoiding the well-known "callback hell" also in stateful, stream-based applications.

[1]See `http://blog.twitter.com/2009/01/inauguration-day-on-twitter.html`

Type and Effect Systems

Concurrent processes, such as actors, as provided by imperative, object-oriented languages typically suffer from the problem that process isolation is conventional rather than enforced by the programming language. As a result, even using message-passing concurrency in these languages is not guaranteed to prevent data races because of unsynchronized access to shared data. This problem has motivated my work on a new type system for unique references in object-oriented languages [6]. The type system is based on static capabilities which enable patterns such as ownership transfer of objects between concurrent actors. A prototype of the type system has been implemented as a pluggable type system in Scala, and it has been used to type-check substantial programs such as a subset of Scala's collections library as well as the parallel testing framework used to test the Scala compiler and standard library. I have also contributed to a framework for polymorphic effect checking [12]; this framework is particularly lightweight, which significantly reduces the burden of adding effect annotations on the programmer in order to check properties such as purity.

Data-Centric Programming

I have found that large-scale, parallel machine learning is a fruitful application area for programming language research [4]. Advances in programming abstractions, optimizations, and DSLs [11] can significantly increase the flexibility, efficiency, and scalability of machine learning frameworks. I have also contributed to a new approach for efficient, extensible serialization [2] which is central to frameworks for large-scale data analytics.

FUTURE RESEARCH

I envision two major avenues for my future research: first, I would like to build on my experience in concurrency, type and effect systems, and data-centric programming. Second, I would like to expand my research to other areas of programming languages, compilers, and software engineering. More specifically, I am interested in pursuing research in the following areas:

Concurrency. When talking to users of actors and futures in Scala, an often-voiced concern is concurrency hazards due to unsafe uses of libraries without static or dynamic checks. I would like to explore new ways to address this lack of robustness using several approaches that can initially be explored independently; however, to leverage synergies, ultimately, I'd like to bring them together in one integrated actor-based programming system. First, I'd like to find practical solutions to enforce correct usage of actors, futures, and combinations thereof in existing programming languages. To enable comprehensive safety checks, I plan to investigate the symbiosis of libraries and safe language subsets. In this approach, programs in the safe language subset are valid programs in the full language; thus, compilers and tooling can be reused. Second, I'd like to explore new ways to detect concurrency hazards, in particular data races. Encouraged by recent results on dynamic data race detection for event-driven programs, I plan to devise a new dynamic race detection algorithm for reactive programs based on actors and futures. Third, to increase programmer productivity, I'm interested in semi-automatic synthesis of actor programs based on partial implementations. I plan to leverage language subsetting to identify restricted actor models amenable to model checking. An important open question I'd like to answer is: what kinds of partial inputs are well-suited for reactive applications?

Type and Effect Systems. Type systems based on static capabilities are effective at tracking disjoint object graphs in the heap [1]. Among others, such heap partitions support safe concurrency and parallelism, thread locality optimization, and security. However, such type systems still require a non-negligible amount of additional type annotations in programs which typically limits their practical use to experts. I would like to investigate how far our capability-based type system [6] can be specialized for programming models that already provide determinacy properties without such capabilities. Such a specialization could result in significant simplifications and much fewer required type annotations.

Compilers. Scala-Virtualized [11] enables compiler frameworks for embedded DSLs targeting heterogeneous platforms such as multicores, GPUs, and clusters. While such a language-based framework is well-suited for high-performance computing DSLs, it does not support general-purpose applications. I would like to develop a principled approach based on types and effects to identify portable code together with compiler technology for generating code for heterogeneous platforms including the "glue code" required to communicate between different runtime environments. One exciting outlook of this work is enabling "full stack" development where the entire code of a typical cloud application is written in a single programming language.

Programming with Objects and Functions. New computing paradigms, such as cloud computing or "big data" processing, present new challenges that current programming languages do not always support satisfactorily. I am very interested in revisiting the foundations of object-oriented and functional languages to better support these new computing paradigms. Besides a new approach to working with off-heap data [2], I am currently exploring a variant of closures that provides more safety in concurrent and distributed settings.

References

[1] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. "Uniqueness and Reference Immutability for Safe Parallelism". In: *OOPSLA*. 2012.

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

[3] **P. Haller** and T. Van Cutsem. "Implementing Joins Using Extensible Pattern Matching". In: *COORDINATION*. 2008.

[4] **P. Haller** and H. Miller. "Parallelizing Machine Learning Functionally: A Framework and Abstractions for Parallel Graph Processing". In: *Scala Workshop*. 2011.

[5] **P. Haller** and H. Miller. "RAY: Integrating Rx and Async for Direct-Style Reactive Streams". In: *ACM SPLASH REM Workshop*. 2013.

[6] **P. Haller** and M. Odersky. "Capabilities for Uniqueness and Borrowing". In: *ECOOP*. 2010.

[7] **P. Haller** and M. Odersky. "Scala Actors: Unifying Thread-based and Event-based Programming". In: *Theoretical Computer Science* 410 (2-3 Feb. 2009), pp. 202–220.

[8] **P. Haller** and F. Sommers. *Actors in Scala*. Artima Press, 2012.

[9] **P. Haller** and J. Zaugg. *SIP-22: Async*. `http://docs.scala-lang.org/sips/pending/async.html`. Accessed: 2013-12-20. 2013.

[10] A. Prokopec, H. Miller, T. Schlatter, **P. Haller**, and M. Odersky. "FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction". In: *LCPC*. 2012.

[11] T. Rompf, A. Moors, N. Amin, **P. Haller**, and M. Odersky. "Scala-Virtualized: Linguistic Reuse for Deep Embeddings". In: *Higher-Order and Symbolic Computation* (Sept. 2013).

[12] L. Rytz, M. Odersky, and **P. Haller**. "Lightweight Polymorphic Effects". In: *ECOOP*. 2012.