Language Support for Distributed Functional Programming

THIS IS A TEMPORARY TITLE PAGE
It will be replaced for the final print by a version
provided by the service academique.

Thèse n. 6784
présenté le 3 Septembre 2015
à la Faculté Informatique et Communications
laboratoire de méthodes de programmation
programme doctoral en en informatique et communications
École Polytechnique Fédérale de Lausanne

pour l'obtention du grade de Docteur dès Sciences
par

Heather Miller

acceptée sur proposition du jury:

Prof James Larus, président du jury
Prof Martin Odersky, directeur de thèse
Prof Viktor Kuncak, rapporteur
Prof Jan Vitek, rapporteur
Prof Matei Zaharia, rapporteur

Lausanne, EPFL, 2015

Acknowledgements

A PhD is never easy for anyone. For most, it's a road fraught with challenges, technical and ideological. I had a rougher start than most, bouncing around completely disparate fields for two full years, an entire ocean away from home, in a country where I knew no one and couldn't speak the local language before I joined the LAMP group. Therefore, I must first and foremost thank my advisor, Martin Odersky, for looking at this oddball PhD student with a background in signal processing and electrical engineering, in another research group, doing something totally different, and giving me a chance to try and build meaningful frameworks and abstractions as part of the Scala team at EPFL these past four years. Without his support and insight, this dissertation would not have been possible.

I'd also like to thank my friends in Lausanne and colleagues at EPFL, those in LAMP and those not. If it wasn't for you, I'd not have gotten here. Switzerland can be a lonely place for those from far away. You were the people I could speak to, joke with, and generally relax around during these long six years. The list is long, and I hope I manage to mention everybody.

To my friends who started this journey with me; roommates and EDIC office colleagues, thank you. Yuliy Schwarzburg, a longtime friend from Cooper Union in New York City, and roommate here in Switzerland, and his fiancée Lyvia Fishman. Arash Farhang, my best mountain buddy and now caretaker of my old best friend, Umlaut. Evan Williams and Davide de Masi – 'MURICAH! – thanks for all of the good times and solidarity in being ignorant Americans lost on this continent without HVAC and diners together. Petr Susil, Iulian Dragos, Tanja Petricevic, Cristina Ghiurcuta, Jennifer Sartor, Eva Darulova, Tihomir Gvero, Horesh Ben Shitrit, Adar Hoffman, and Alla Merzakreeva, thank you for being some of my first friends in Switzerland.

One person that stood out during these years in Switzerland is Liz Daley. Liz didn't live in any one place. She rode big mountains and climbed epic splitter all over the world, based often in Seattle or Chamonix/Lausanne. She lived her dreams and became one of the first and few pro woman snowboarders and mountaineers. I never had the chance to tell her how inspiring she was. Liz, you constantly remind me what stoke is. Even I (one of a thousand distant non-mountaineering buddies) think of you often. You were an example to myself and many. Your time with us was far too short.

Acknowledgements

Importantly, I want to thank my colleagues in the LAMP laboratory. You all were the source of so many deep discussions, explorations of ideas, and of course many beers, ski trips, or other unforgettable shenanigans. Sandro Stucki, Manohar Jonnalagedda, Alex Prokopec, Ingo Maier, Vojin Jovanovic, Hubert Plociniczak, Tobias Schlatter, Donna Malayeri, Vlad Ureche, Lukas Rytz, Adriaan Moors, Gilles Dubochet, Tiark Rompf, Miguel Garcia, Denys Shabalin, Eugene Burmako, Sébastien Doeraene, Christopher Jan Vogt, Dmitry Petrashko, Samuel Grütter, Nada Amin, and Antonio Cunei – thank you for the camaraderie all these years. And of course, thank you to Danielle Chamberlain and Fabien Salvi for fielding all of my administrative and computer-related questions, respectively.

Being in a position to start a PhD is one thing, and a PhD thesis acknowledgement section wouldn't be complete without thanking those unknowing mentors who are largely responsible for me taking this path to higher education at all. Firstly, I must thank a high school teacher of mine, all the way back to the days where I majored in fine arts at Alexander W. Dreyfoos School of the Arts back home in West Palm Beach, Florida. Jenny Gifford helped me to realize that I had any potential at all. Without her encouragement, I'd have never ended up at univeristy at all, let alone a top university like Cooper Union. So Jenny, thank you for seeing something in me. Without you, I'd not be where I am. Secondly, I'd like to thank Professor Bethanie Stadler, a short-term advisor I had while I participated in a US National Science Research Experience for Undergrads (REU) program at the University of Minnesota. Professor Stadler is a professor of Materials Science – a field I knew nothing about upon starting an REU with her. Professor Stadler helped me to realize that I was capable of doing independent research, even if I was entering a new field where I had little to no experience at the onset. The encouragement she showed me during my time in Minnesota led to a trip to present our work at a scientific conference in the French Alps – my first trip to Europe, and the turning point that helped me to realize (1) I can do a PhD, and (2) in Switzerland. Beth introduced me to EPFL. Without her, I'd likely not have gone to grad school, and I'd never have heard of EPFL.

I would like to thank my close friends here in Switzerland and back home in the US. Darja Jovanavic and (not-so) little David Jovanovic, you were my buddies in Lausanne through all of the moments when life and the PhD got tough – a mere thank you is not enough. I'd also like to thank my lifelong friends back home in the US, Lindsay Hebrank, Beth Bachelor, thanks for always being a friend, no matter how far apart we are.

I'd also like to thank my siblings for teaching me so much about life and even kids. Thank you to my little sisters Ashley Marcantonio and Kayla Marcantonio for always teaching me something new and for always such being a riot. And of course, thank you to my little brother Sean Miller for teaching me the virtue of patience.

I'd like to thank my parents, my mother Christina Ellis Marcantonio, my step-father Timothy Marcantonio, and my father Steve Miller for being there for me all these years. I know we didn't come from a lot, but you did the best for me that you could, and I will always be thankful. Mom

– if you hadn't patiently spent your days teaching me how to read and write as a toddler, I might not have ever realized the power of knowledge and thought. You gave me the educational foundation that I have built upon throughout my entire life, and for that I am eternally grateful.

I want to thank my husband Daniel Klug for his everlasting patience and unconditional love and support. Daniel has had to put up with many late nights spanning from from paper deadlines, to lectures, to organizing conferences, as well as months of me traveling, from India to San Francisco, and all the while, he has been there for me, helping me through life with an uplifiting smile, and always a hilarious pun. I'm truly lucky to have you by my side in this life, and I still don't know what I did to deserve you.

Last but not least, I'd like to thank Philipp Haller, my closest friend these past five years, and my co-author. Philipp, you were the one that stood next to me through all of the toughest moments and greatest triumphs during the PhD these past five years. I will never forget what we went through together – the paper pushes, the epic travels, the times when life in general got immeasurably difficult. Not only did you stand with me through unspeakably hard times and the good, always kind, forgiving, and patient, but you also spent countless hours teaching me much of what I know about Computer Science and Programming Languages. I know I can never repay the time that you invested in me, and for that, know that I am forever grateful. Thank you from the bottom of my heart.

Basel, Switzerland, July 26th, 2015 H. M.

Abstract

Software development has taken a fundamental turn. Software today has gone from simple, closed programs running on a single machine, to massively open programs, patching together user experiences byway of responses received via hundreds of network requests spanning multiple machines. At the same time, as data continues to stockpile, systems for big data analytics are on the rise. Yet despite this trend towards distributing computation, issues at the level of the language and runtime abound. Serialization is still a costly runtime affair, crashing running systems and confounding developers. Function closures are being added to APIs for big data processing for use by end-users without reliably being able to transmit them over the network. And much of the frameworks developed for handling multiple concurrent requests byway of asynchronous programming facilities rely on blocking threads, causing serious scalability issues.

This thesis describes a number of extensions and libraries for the Scala programming language that aim to address these issues and to provide a more reliable foundation on which to build distributed systems.

This thesis presents a new approach to serialization called *pickling* based on the idea of generating and composing functional pickler combinators statically. The approach shifts the burden of serialization to compile time as much as possible, enabling users to catch serialization errors at compile time rather than at runtime. Further, by virtue of serialization code being generated at compile time, our framework is shown to be significantly more performant than other state-of-the-art serialization frameworks. We also generalize our technique for generating serialization code to generic functions other than pickling.

Second, in light of the trend of distributed data-parallel frameworks being designed around functional patterns where closures are transmitted across cluster nodes to large-scale persistent datasets, this thesis introduces a new closure-like abstraction and type system, called *spores*, that can guarantee closures to be serializable, thread-safe, or even have custom user-defined properties. Crucially, our system is based on the principle of encoding type information corresponding to captured variables in the type of a spore. We prove our type system sound, implement our approach for Scala, evaluate its practicality through a small empirical study, and show the power of these guarantees through a case analysis of real-world distributed and concurrent frameworks that this safe foundation for closures facilitates.

Finally, we bring together the above building blocks, pickling and spores, to form the basis of a new programming model called *function-passing*. Function-passing is based on the idea of a

distributed persistent data structure which stores in its nodes transformations to data rather than the distributed data itself, simplifying fault recovery by design. Lazy evaluation is also central to our model; by incorporating laziness into our design only at the point of initiating network communication, our model remains easy to reason about while remaining efficient in time and memory. We formalize our programming model in the form of a small-step operational semantics which includes a precise specification of the semantics of functional fault recovery, and we provide an open-source implementation of our model in and for Scala.

Key words: distributed programming, functional programming, closure, serialization, programming model, concurrency, asynchronous programming, dataflow.

Zusammenfassung

Die Software-Entwicklung hat eine grundlegende Wendung durchlaufen. Software hat sich heutzutage von einfachen geschlossenen Programmen, die auf einem einzigen Rechner laufen, hin zu „massive open programs“ gewandelt, die Nutzeranfragen zusammenführen, die als Antworten von hunderten von Netzwerkanfragen an eine Vielzahl an Diensten eingegangen sind. Zeitgleich dazu werden Daten weiterhin angesammelt und Systeme für Big Data Analytics sind auf dem Vormarsch. Trotz des Trends zum verteilten Computing, sind Fragen zu Programmiersprachen und Laufzeitsystemen im Überfluss vorhanden. Serialisierung ist hinsichtlich der Laufzeit nach wie vor eine kostspielige Angelegenheit, die laufende Systeme zum Abstürzen bringt und Entwickler verwirrt. Funktions-Closures werden zu APIs hinzugefügt, um durch Anwendungsentwickler zum Bearbeiten von Datensätzen massiver Grösse genutzt werden zu können, jedoch ohne sicherzustellen, dass diese über das Netzwerk gesendet werden können. Und ein Grossteil der Frameworks, die zur Verarbeitung multipler, gleichzeitiger Anfragen durch asynchrone Programmierabstraktionen entwickelt wurden, basiert auf dem Blockieren von Threads, was schwerwiegende Skalierungsprobleme verursacht.

Die vorliegende Dissertation beschreibt eine Reihe von Erweiterungen und Bibliotheken für die Programmiersprache Scala, um die genannten Probleme anzugehen und eine zuverlässigere Grundlage für die Konstruktion verteilter Systeme zu entwickeln.

Die Arbeit stellt einen neuen Ansatz zur Serialisierung, *Pickling*, vor, welcher auf der Idee der Generierung und Komposition statischer Pickling-Funktionen beruht. Dieser Ansatz verlagert den Aspekt der Serialisierung so stark wie möglich auf die Übersetzungszeit, um Anwendern zu ermöglichen, Serialisierungsfehler zur Übersetzungszeit zu erkennen statt zur Laufzeit. Des Weiteren ist unser Framework durch den Serialisierungscode, der beim Kompilieren erzeugt wird, deutlich performanter als andere existierende Serialisierungs-Frameworks. Zudem verallgemeinern wir unseren Ansatz zur Generierung weiterer Datentyp-generischer Funktionen neben der Serialisierung.

In Anbetracht der Tendenz verteilter daten-paralleler Frameworks, die für funktionelle Muster entworfen wurden, bei denen Closures über Cluster-Knoten zu großen persistenten Datensätzen übertragen werden, stellt diese Arbeit eine neue Closure-artige Abstraktion und Typsystem, *Spores*, vor, dass garantieren kann, dass Closures serialisierbar sind, Thread-sicher sind, und sogar benutzerdefinierte Eigenschaften haben. Entscheidend ist, dass unser System auf dem Prinzip basiert, im Typ eines Spores Typinformation zu kodieren, welche den gefangenen Variablen entspricht. Wir beweisen die Korrektheit unseres Typsystems, implementieren unseren

Ansatz in Scala, evaluieren dessen Praktikabilität mithilfe einer kleinen empirischen Studie, und zeigen die Mächtigkeit dieser Garantien mithilfe einer Fallstudie realistischer verteilter und nebenläufiger Frameworks, die durch diese sichere Grundlage für Closures unterstützt werden.

Schließlich bringen wir die obengenannten Bausteine, Pickling und Spores, zusammen, um die Basis eines neuen Programmiermodells, genannt *Function-Passing*, zu bilden. Function-Passing basiert auf der Idee einer verteilten persistenten Datenstruktur, die in ihren Knoten Daten-Transformationen anstelle der verteilten Daten selbst enthält, was die Fehlerbeseitigung per Konstruktion vereinfacht. Lazy Evaluation ist auch von zentraler Bedeutung für unser Modell; da Lazy Evaluation in unserem Design nur an der Stelle der Initiierung von Netzwerk-Kommunikation Bedeutung hat, bleibt die logische Grundlage unseres Modells leicht verständlich, während es hinsichtlich Zeit und Speicherverbrauch effizient bleibt. Wir formalisieren unser Programmiermodell in Form einer strukturierten operationellen Semantik, die eine präzise Spezifikation der Semantik funktionaler Fehlerbeseitigung umfasst, und wir stellen eine Open-Source-Implementierung unseres Modells in und für Scala bereit.

Stichwörter: Verteilte Programmierung, Funktionale Programmierung, Closure, Serialisierung, Programmiermodell, Nebenläufigkeit, Asynchrone Programmierung, Datenfluss

Contents

List of Figures

List of Tables

1 Introduction

Developing professional software these days has become quite an involved affair. Not long ago, a team of engineers would sit down to develop an application that would simply and modestly run on a single computer. Such software would operate completely in its own world, blissfully unaware of the internet, only making a network call on seldom occasions, *e.g.,* to phone home to its vendor to ask for software updates. This was the state of software development a few short years ago.

Today, large swaths of most applications have been woven into "the cloud" or other network services. Web applications are becoming patchwork quilts made up of calls to multitudes of different microservices. Modest mobile "apps" now make network calls to dozens or even hundreds of services. Meanwhile as software becomes evermore pervasive, weaving itself more into more of our daily habits in more places, content providers are focusing their energies on collecting any and all seemingly innocuous pieces of our data that they can, in an attempt to unlock some sort of market value in peoples' trails of digital breadcrumbs. With all of this data piling up, industry and academia are scrambling to build distributed systems that can help more users make sense of it–clusters of machines working together to churn through datasets too large to fit in the memory of a single machine.

This is the new computing landscape; the network has become ubiquitous and is now baked into much of the programming that professional developers do.

Meanwhile, at the same time, we are witnessing a renaissance of functional programming so prevalent that it has permeated the daily routines of software developers on all ends of the software development spectrum, from the client side[1] to the server side.[2] Further, the distributed system cores of services like Twitter are based on functional APIs [Eriksen, 2013], and frameworks for big data analytics like Spark [Zaharia et al., 2012] credit functional patterns for enabling more powerful computation patterns; *i.e.,* general graphs of computations built

[1] Popular functional languages for the client side include: numerous JavaScript libraries such as Underscore.js, Elm [Czaplicki, 2012], PureScript, Scala.js, amongst many others.

[2] Popular functional languages for the server side include: Scala [Odersky et al., 2010], Clojure [Hickey, 2008], Erlang [Armstrong, 2010], Haskell [Peyton Jones, 2014], amongst many others.

up of compositions of higher-order combinators rather than just maps and reduces like in MapReduce [Zaharia, 2014]. Just about everywhere you look nowadays, you will find functionally-inspired software springing up in the wild.

But how have our most important tools in professional software development – programming languages – kept up as the network and functional programming have begun to proliferate software development environments?

As it turns out, there are still numerous issues using language constructs such as objects and functions in a distributed setting. Moreover, due to their nature of being built-in to the language, it is impossible to rely on libraries and frameworks to provide support for the reliable distribution of these constructs. As a result, even mature libraries and frameworks can exhibit bugs that are hard to diagnose and fix.

For example, in mainstream languages like Java, even the serialization of simple objects, a prerequisite for sending them across the network, can lead to runtime errors that can be difficult to diagnose and fix. Consequently, many frameworks and systems use alternative serialization frameworks, such as Google's Protocol Buffers, Apache Avro, or Kryo. However, these typically have their own set of limitations: weaker or no type safety, a fixed serialization format, more restrictions placed on the objects to-be-serialized, or only rudimentary language integration.

This issue is exacerbated when using closures, which are increasingly appearing in popular distributed frameworks such as Spark [Zaharia et al., 2012] and Scalding [Twitter, 2015]. One of the main reasons is that closures, as they exist in virtually all wide-spread languages, leave essential components, such as their captured variables, implicit, preventing customizations necessary to make closures safer and more efficient to distribute.

The goal of this dissertation is to revisit the fundamental concepts of modern languages, objects and functions, and to make them safer and more efficient to use in a distributed environment. We focus on three important and orthogonal building blocks for distributed programming:

- ***Pickling (serialization)***
- ***Functions***
- ***Asynchronous programming***

This thesis is concerned with two essential aspects of distribution: communication and concurrency. First, we present a new approach to communicate both objects and functions between distributed nodes safely and efficiently. Second, we present novel lock-free concurrency abstractions suitable for building large-scale distributed systems. Finally, we integrate the two approaches in the context of a new distributed programming model. Designed from

the ground up using our new primitives for distribution, the model generalizes existing widely-used programming systems for data-intensive computing.

More specifically, this dissertation aims to address the following questions:

- How can existing programming-language features be improved in order to better support concerns like performance and latency across a general slice of distributed systems?
- Which important features and aspects of existing programming languages are left unsupported by the language in the face of distribution? Is it possible to support such features?
- How can core ideas behind the development of functional programming be applied to the distributed scenario? What other models for functional programming in a distributed environment are there?
- What are good abstractions for reasoning about concerns like network I/O and failure at the level of the compiler and programming language?

1.1 Contributions

This dissertation describes a number of extensions and libraries in and for Scala which aim to provide a more reliable foundation for building distributed systems atop of.

In detail, our contributions are the following:

- We describe an abstraction and underlying data structure for parallel dataflow programming, *FlowPools*. FlowPools are fully asynchronous, and functionally-inspired, and as a result are composable. We prove several important properties about FlowPools, including lock-freedom, linearizability, and determinism. We also show through a detailed evaluation that FlowPools can outperform similar concurrent collections in the Java standard library.
- We introduce an extension to pickler combinators, well-known in functional programming, to support the core concepts of object-oriented programming namely subtyping polymorphism, open class hierarchies, and object identity.
- We provide a framework called *scala/pickling* based on object-oriented pickler combinators which (a) enables retrofitting existing types with pickling support, (b) supports automatically generating picklers at compile time and at runtime, (c) supports pluggable pickle formats, and (d) does not require changes to the host language or the underlying virtual machine. We also provide an experimental evaluation that shows scala/pickling to outperform Java serialization and Kryo on a number of data types used in real-world, large-scale distributed applications and frameworks.

- We generalize the generation technique used in scala/pickling to generic functions other than pickling. The technique, called *Self Assembly*, is a general technique for defining generic operations or properties that operate over a large class of types which requires little boilerplate; shares the extensibility and customizability properties of type classes; and, due to compile-time code generation, provides high performance. Importantly, our approach enables the definition of datatype-generic functions that support features present in production OO languages, including subtyping, object identity, and generics.

- We describe how `self-assembly` enables the definition of custom lightweight static type checks to guarantee that certain static properties hold at runtime, *e.g.*, immutability.

- We cover the `self-assembly` library, a complete and full-featured implementation of our technique in and for Scala. The library includes several auxiliary definitions, such as generic queries and transformations, that help define new lightweight static checks of generic properties. Importantly, `self-assembly` doesn't require any extension to the language or compiler. We also evaluate the expressivity and performance of `self-assembly` by porting scala/pickling, keeping the same published performance numbers while reducing the code size for type class instance generation by 56%.

- We introduce a closure-like abstraction and type system, called *spores* which avoids typical hazards when using closures in a concurrent or distributed setting through controlled variable capture and customizable user-defined constraints for captured types. Further, we describe an approach for type-based constraints on spores that can be combined with existing type systems to express a variety of properties from the literature, including, but not limited to, serializability and thread-safety/immutability. We formalize spores with these type constraints and prove soundness of the type system.

- We present an implementation of spores in and for the full Scala language, and (a) demonstrate the practicality of spores through a small empirical study using a collection of real-world Scala programs, and (b) show the power of the guarantees spores provide through case studies using parallel and distributed frameworks.

- We introduce a new data-centric programming model called *function-passing*, based on pickling and spores, for functional processing of distributed data which makes important concerns like fault tolerance simpler by design. The main computational principle is based on the idea of sending safe, guaranteed serializable functions to stationary data. Using standard monadic operations our model enables creating immutable DAGs of computations, supporting decentralized distributed computations. Lazy evaluation enables important optimizations while keeping programs simple to reason about. We describe a distributed implementation of the programming model in and for Scala.

- A provide a formalization of our programming model based on a small-step operational semantics. Inspired by widespread systems like Spark [Zaharia et al., 2012], our formalization is a first step towards a formal, operational account of real-world fault recovery mechanisms. The presented semantics is clearly stratified into a deterministic layer

and a concurrent/distributed layer. Importantly, reasoning techniques for sequential programs are not invalidated by the distributed layer.

1.2 Structure

The rest of this dissertation is organized as follows.

- Chapter 2 describes *futures* and *FlowPools*, functionally-inspired and fully asynchronous and non-blocking single-assignment variables (futures) and pools (FlowPools) useful for reducing coordination in distributed systems. The chapter sketches a proof of linearizability, lock-freedom, and determinism of FlowPools. The full proof of lock freedom can be found in Appendix A, and the full proofs of linearizability and determinism can be found in the companion technical report [Prokopec et al., 2012b].
- Chapter 3 introduces *object-oriented picklers* and *scala/pickling*, a new distribution-focused approach to serialization that generates serialization code statically, allowing for more type safety. The chapter includes a formalization of object-oriented picklers as well as a description of the generation algorithm used for automatically generating picklers for arbitrary types. A performance evaluation is also included which examines the performance of the serialization framework across different sorts of serialization workloads, and which compares scala/pickling against other state-of-the-art serialization systems like Java and Kryo, and reports significant speedups.
- Chapter 4 covers a new technique for extensible and static datatype-generic programming. In this chapter, the generation technique used for generating pickling code is generalized to be able to generate arbitrary type class instances, at compile time.
- Chapter 5 introduces *spores*, a new abstraction and type system designed to enable function closures to be serializable by design. The type system presented here also generalizes its added static checking capabilities to arbitrary user-defined *properties*, *e.g.,* immutability.
- Chapter 6 describes a new programming model for functional distributed programming called *function-passing* which aims to simplify the implementation of and reasoning about fault-recovery mechanisms. This programming model can be thought of as a generalization of the Spark or MapReduce programming model.
- Chapter 7 concludes and discusses possible directions for future work.

1.3 Previously Published Material

This dissertation draws heavily on earlier work described in the following papers, written jointly with several collaborators (in the order of appearance in this dissertation):

- Prokopec, Miller, Schlatter, Haller, and Odersky (2012). FlowPools: A lock-free deterministic concurrent dataflow abstraction. In proceedings of Languages and Compilers for Parallel Computing (LCPC).
- Miller, Haller, Burmako, and Odersky (2013). Instant Pickles: Generating object-oriented pickler combinators for fast and extensible serialization. In proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).
- Miller, Haller, and Odersky (2014). Spores: A type-based foundation for closures in the age of concurrency and distribution. In proceedings of the European Conference on Object-Oriented Programming (ECOOP).
- Haller and Miller (2015). Distributed Programming via Safe Closure Passing. In proceedings of Programming Language Approaches to Concurrency and Communication-centric Software (PLACES).

Works that this dissertation draws upon that have been submitted but at the time of the writing remain in technical report form include:

- Miller, Haller, and C. D. S. Oliveira (2015). Self-Assembly: Lightweight language extension and datatype generic programming, all-in-one! EPFL technical report #EPFL-CONF-199389.
- Miller and Haller (2015). Function Passing: A model for typed, distributed functional programming. EPFL technical report #EPFL-CONF-205822.

2 Asynchronous Programming

Nowadays, providing a modest experience on a mobile app, or even rendering simple web pages typically requires the collaboration of dozens of network services each speaking many different languages or protocols to one another. Such systems are one of many flavors of a *distributed system*, and as such must coordinate between many network requests to, as quickly and reliably as possible, piece together an interface or some other user experience.

Responsiveness is a requirement. Yet providing a responsive experience is at odds with the need to piece together the results from many calls over the network to other services. Synchronously making a request to a remote service and blocking, or waiting, until that request is fulfilled before moving on to the next request is slow – roundtrip network communication is known to be 1,000,000 to 10,000,000 times slower than roundtrips to main memory [Norvig and Dean, 2012] – and making requests sequentially, one by one, is also often unnecessary.

Asynchronous programming solves these problems by separating the execution of individual tasks (*e.g.*, calls to network services) from the main program flow. In a language like Scala where tasks can be executed by multiple threads, this reduces blocking because rather than stopping a thread to wait on the completion of another task, a separate task is simply scheduled to proceed when the resource its waiting for becomes available. Thus freeing up the thread that would otherwise be waiting to do more meaningful work.

In this chapter, we will see two abstractions for fully non-blocking, asynchronous programming; functionally-inspired *futures and promises* in Scala [Haller et al., 2012] in Section 2.1 and a generalization of futures to a pool or multiset-type data structure called *FlowPools* [Prokopec et al., 2012a] in Section 2.2.

2.1 Futures

Futures and promises can be thought of as, together, a unified abstraction used for synchronization in programming languages with support for concurrency. Futures and promises in Scala [Haller et al., 2012] stand out from their Java counterparts in two ways; (a) they are

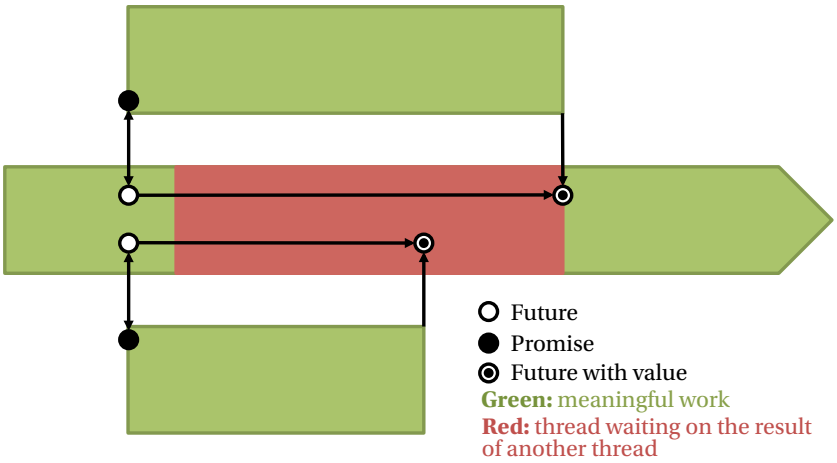


Figure 2.1 – Illustration of blocking futures, as in Java. The central green arrow can be thought of as the main program thread.

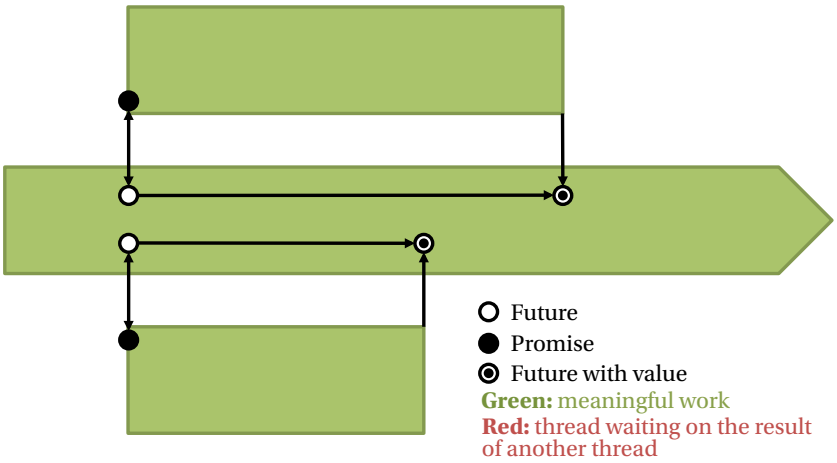


Figure 2.2 – Illustration of fully asynchronous, non-blocking futures, as in Scala. The central green arrow can be thought of as the main program thread.

functionally-inspired with monadic combinators and are thus composable, and (b) they are fully asynchronous and non-blocking by default. A visualization of this blocking difference and definition is shown in Figures 2.1 and 2.2. Here, the central green arrow in each figure can be thougth of as the main program thread.

A *future* can be thought of as a container which represents a value that will eventually be computed. They're related to *promises* in that a future is a read-only window to a single-assignment (write-once) variable called a *promise*. This relationship is illustrated in Figure 2.3.

Before a future's result is computed, we say that the future is *not completed*. If the computation representing a future is finished with a value or an exception, we say that the future is *completed*. Completion can take one of two forms: (a) when a future is completed with a

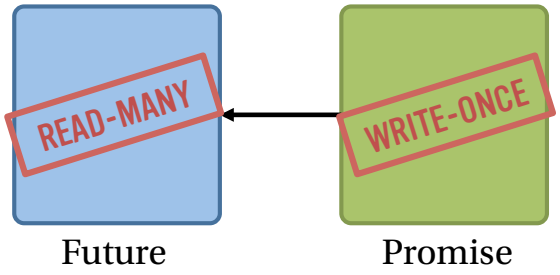


Figure 2.3 – Futures and promises can be thought of as a single concurrency abstraction.

value, we say the future was *successfully completed* with that value, or (b) when a future is completed with an exception thrown by the computation, we say the future was *failed* with that exception.

2.1.1 Basic Usage

The type of `Future` and `Promise` is as follows: (*simplified*)

```
trait Future[T] {
  def onSuccess(f: T => Unit): Unit
}
```

```
trait Promise[T] {
  def success(elem: T): Unit
  def future[T]: Future[T]
}
```

As depicted visually in Figure 2.3, every `Promise[T]` can return a reference to its corresponding `Future` with the `future` method.

An example of how a future can be created is as follows. Let's assume that we want to use a hypothetical API of some popular social network to obtain a list of friends for a given user. We will open a new session and then send a request to obtain a list of friends of a particular user:

```
val session = ... // obtain a list of friends for some user/credentials
val f: Future[List[Friend]] = Future {
  session.getFriends() // network call to get a list of that user's friends
}
```

To obtain the list of friends of a user, a request has to be sent over a network, which can take a long time. This is illustrated with the call to the method `getFriends` that returns `List[Friend]`. To better utilize the CPU until the response arrives, we should not block the rest of the program – this computation should be scheduled asynchronously. The future method does exactly that–it performs the specified computation block concurrently, in this case sending a request to the server and waiting for a response.

The list of friends becomes available in the future `f` once the server responds.

An unsuccessful attempt may result in an exception. In the following example, the session value is incorrectly initialized, so the computation in the future block will throw a `NullPointerException`. This future `f` is then failed with this exception instead of being completed successfully:

```
val session = null
val f: Future[List[Friend]] = Future {
  session.getFriends
}
```

We now know how to start an asynchronous computation to create a new future value, but we have not shown how to use the result once it becomes available, so that we can do something useful with it. Once created, a future may be used in one of two ways, either via:

- *callbacks*, or
- *composable higer-order combinators*, such as `map`, `flatMap`, and `filter`.

We will see how to use both callbacks and higher-order functions to interact with to-be-computed values in the following two subsections.

2.1.2 Callbacks

One way to interact with the result of a future computation in a non-blocking way is to attach a callback to perform some side-effecting operation such as completing another future. Callbacks are a typical way to do asynchronous computation–a callback is a function that is called once its arguments become available. There are three methods provided to work with callbacks on Scala's futures:

- `def foreach[U](f: (T) => U): Unit`
- `def onComplete[U](f: (Try[T]) => U): Unit`
- `def onSuccess[U](pf: PartialFunction[T, U]): Unit`
- `def onFailure[U](pf: PartialFunction[Throwable, U]): Unit`

The most general form of registering a callback is by using the `onComplete` method, which takes a callback function of type `Try[T]=>U` [1]. The callback is applied to the value of type `Success[T]` if the future completes successfully, or to a value of type `Failure[T]` otherwise.

[1] `Try[T]` can be thought of as being similar to `Option[T]` or an `Either[T, S]` in that it is a container type. However, it has been specifically designed to either hold a value or some throwable object. `Try[T]` is a `Success[T]` when it holds a value and otherwise `Failure[T]`, which holds an exception. Another way to think of `Try[T]` is to consider it as a special version of `Either[Throwable, T]`, specialized for the case when the left value is a `Throwable`.

To get a feeling for how `onComplete` is used, let's use a running example. Let's assume for a given social network, we want to fetch a list of our own recent posts and render them to the screen. We can do this with `onComplete`:

```
val f: Future[List[String]] = Future {
  session.getRecentPosts
}
f onComplete {
  case Success(posts) => for (post <- posts) println(post)
  case Failure(t) => println("An error has occured: " + t.getMessage)
}
```

The `onComplete` method is general in the sense that it allows the client to handle the result of both failed and successful future computations. To handle only successful results, the `onSuccess` callback is used (which takes a partial function). Similarly, to handle failed results, the `onFailure` callback is used:

```
val f: Future[List[String]] = Future {
  session.getRecentPosts
}
f onFailure {
  case t => println("An error has occured: " + t.getMessage)
}
f onSuccess {
  case posts => for (post <- posts) println(post)
}
```

The `onComplete`, `onSuccess`, and `onFailure` methods have result type `Unit`, which means invocations of these methods cannot be chained. This design is intentional, to avoid suggesting that chained invocations may imply an ordering on the execution of the registered callbacks (callbacks registered on the same future are unordered).

2.1.3 Higher-Order Combinators

While callbacks work reasonably well in simple situations, they can quickly get out of hand and when numerous, they can become difficult to reason about. Programmers affectoinately refer to this situation as *callback hell*.

Scala's futures provide combinators which allow a more straightforward composition. What's more, due to the type signature of these methods (they each return another `Future`), it's possible to *compose* operations on futures and to build up rich computation graphs. The three basic functional combinators on futures include:

- `def map[S](f: (T) => S): Future[S]`
- `def flatMap[S](f: (T) => Future[S]): Future[S]`
- `def filter(p: (T) => Boolean): Future[T]` *(Also,* `withFilter`*)*

To get a feeling for how to use these combinators, and later, how to *pipeline* or *chain* them together to build up computation graphs, let's start with a simple example.

Assume we have an API for interfacing with a currency trading service. Suppose we want to buy US dollars, but only when it's profitable. One of the basic combinators is `map`, which, given a future and a mapping function for the value of the future, produces a new future that is completed with the mapped value once the original future is successfully completed. We can use the `map` combinator to handle the successful case:

```
val rateQuote = Future {
  connection.getCurrentValue(USD)
}
val purchase = rateQuote map { quote =>
  if (isProfitable(quote)) connection.buy(amount, quote)
  else throw new Exception("not profitable")
}
purchase onSuccess {
  case _ => println("Purchased " + amount + " USD")
}
```

Here, we start by creating a future rateQuote which gets the current exchange rate. After this value is obtained from the server and the future successfully completed, we call `map` on `rateQuote`, which applies the function which checks whether or not it's profitable to buy US dollars, and if so, it buys some amount of the currency. If we now decide to sell some other currency, it suffices to use map on purchase again.

But what happens if `isProfitable` returns false, hence causing an exception to be thrown? In this case, `purchase` is failed with that exception. Furthermore, imagine that the connection was broken and that `getCurrentValue` threw an exception, failing `rateQuote`. In this case there would be no value to map, so the purchase would automatically be failed with the same exception as `rateQuote`.

In conclusion, if the original future is completed successfully then the returned future is completed with a mapped value from the original future. If the mapping function throws an exception the future is completed with that exception. If the original future fails with an exception then the returned future also contains the same exception. This exception propagating semantics is present in the rest of the combinators, as well.

Importantly, since the methods `map`, `flatMap`, and `withFilter` methods are provided on futures (there is an automatic desugaring from for-comprehensions to calls of these methods), Scala can provide built-in support using for-comprehensions on futures. We will now see an example where a for-comprehension is desirable over using chained higher-order combinators.

In this example, let's assume that we want to exchange US dollars for Swiss francs (CHF). We have to fetch quotes for both currencies, and then decide on buying based on both quotes. Here is what this example would look like using for-comprehension syntax:

```
val usdQuote = Future { connection.getCurrentValue(USD) }
val chfQuote = Future { connection.getCurrentValue(CHF) }
val purchase = for {
  usd <- usdQuote
  chf <- chfQuote
  if isProfitable(usd, chf)
} yield connection.buy(amount, chf)
purchase onSuccess {
  case _ => println("Purchased " + amount + " CHF")
}
```

The purchase `future` is completed only once both `usdQuote` and `chfQuote` are completed–it depends on the values of both these futures so its own computation cannot begin earlier.

The for-comprehension above is translated into:

```
val purchase = usdQuote flatMap {
  usd =>
  chfQuote
    .withFilter(chf => isProfitable(usd, chf))
    .map(chf => connection.buy(amount, chf))
}
```

Here, the `flatMap` operation maps its own value into some other future. Once this different future is completed, the resulting future is completed with its value. In our example, `flatMap` uses the value of the `usdQuote` future to map the value of the `chfQuote` into a third future which sends a request to buy a certain amount of Swiss francs. The resulting future `purchase` is completed only once this third future returned from `map` completes.

2.1.4 Exceptions and Recovery

Futures in Scala also come with a number of combinator methods specialized on handling failures by providing alternate operations in the event of a failure. The three main combinators

for managing failure are:

- `def recover[U >: T](pf: PartialFunction[Throwable, U]): Future[U]`
- `def recoverWith[U >: T](pf: PartialFunction[Throwable, Future[U]]): Future[U]`
- `def transform[S](s: T => S, f: Throwable => Throwable): Future[S]`

To get a feeling for how these work, let's return to the previous example of purchasing currencies. Let's assume that based on the `rateQuote` introduced above, we decide to buy a certain amount of some currency. The `connection.buy` method takes an `amount` to buy and the expected `quote`. It returns the amount bought. If the quote has changed in the meantime, it will throw a `QuoteChangedException` and it will not buy anything. If we want our future to contain 0 instead of the exception, we use the `recover` combinator:

```
val purchase: Future[Int] = rateQuote map {
  quote => connection.buy(amount, quote)
} recover {
  case QuoteChangedException() => 0
}
```

Here, `recover` combinator creates a new future which holds the same result as the original future if it completed successfully. If it did not complete successfully, then the partial function argument is applied to the `Throwable` which failed the original future. If it maps the `Throwable` to some value, then the new future is successfully completed with that value. If the partial function is not defined on that `Throwable`, then the resulting future is failed with the same `Throwable`.

The `recoverWith` combinator creates a new future which holds the same result as the original future if it completed successfully. Otherwise, the partial function is applied to the `Throwable` which failed the original future. If it maps the `Throwable` to some future, then this future is completed with the result of that future. Its relation to recover is similar to that of `flatMap` to `map`.

Combinator `fallbackTo` creates a new future which holds the result of this future if it was completed successfully, or otherwise the successful result of the argument future. In the event that both this future and the argument future fail, the new future is completed with the exception from this future, as in the following example which tries to print US dollar value, but prints the Swiss franc value in the case it fails to obtain the dollar value:

```
val usdQuote = Future {
  connection.getCurrentValue(USD)
} map {
  usd => "Value: " + usd + "$"
}
val chfQuote = Future {
  connection.getCurrentValue(CHF)
} map {
  chf => "Value: " + chf + "CHF"
}
val anyQuote = usdQuote fallbackTo chfQuote
anyQuote onSuccess { println(_) }
```

2.1.5 Execution Contexts

Throughout this chapter, we have covered asynchronous completion of tasks without actually detailing how tasks are eventually executed. All tasks are eventually completed and made available through a future are executed via a so-called `ExecutionContext`, typically, but not necessarily, backed by a thread pool.

In fact many methods, such as all higher-order combinators (`map`, `flatMap`, `filter`), callback-based methods (`onComplete`, `onSuccess`, `onFailure`) and more take an implicit `ExecutionContext` as an argument. For example:

```
def flatMap[S](f: (T) => Future[S])(implicit executor: ExecutionContext): Future[S]
```

For all of these methods, this implicit `executor`, passed via implicit scope, acts as the thread pool or event loop which the given task is executed upon. To globally import a default `ExecutionContext`, one must simply use the following import:

```
import ExecutionContext.Implicits.global
```

This imports an `ExecutionContext` backed by a pre-configured implementation of Java's ForkJoin pool [Lea, 2000].

It's also possible to provide a custom `ExecutionContext` to execute code which blocks on IO or performs long-running computations. For example one may implement a custom `ExecutionContext` by simply extending the `ExecutionContext` trait and importing the preferred execution scheme, such as an implementation of an event loop or a Java `ExecutorService`, and then by implementing a few basic methods such as `execute` and `reportFailure`.

The intent of `ExecutionContext` is to lexically scope code execution. That is, each method, class, file, package, or application should be the one to determine how to run its own code.

This avoids issues such as running application callbacks on a thread pool belonging to a networking library. The size of a networking library's thread pool can be safely configured, knowing that only that library's network operations will be affected while application callback execution can be configured separately.

2.1.6 Blocking

Scala's futures and promises have been designed to be asynchronous in order to gain performance by avoiding blocking. Nonetheless, on occasion, the need to block in an application does unavoidably arise. Thus we cover the methods our framework provides both to manage blocking code, and to accommodate the need to block.

As a running example, let's assume that we want to use a hypothetical API to fetch a potentially large list of images over the network given a list of URLs. Assume that the `download` method for fetching each image is itself a blocking operation:

```
// Retrieve URLs from somewhere
val urls: List[String] = ...

// Download image (blocking operation)
val imagesFuts: List[Future[...]] = urls.map {
  url => future { blocking { download url } }
}

// Do something (display) when complete
val futImages: Future[List[...]] = Future.sequence(imagesFuts)
Await.result(futImages, 10 seconds).foreach(display)
```

Here, `imagesFuts` uses managed blocking via the method `blocking`. `blocking` notifies the thread pool that the block of code passed to it contains long-running or blocking operations. This allows the pool to temporarily spawn new workers to make sure that it never happens that all of the workers are blocked. This is done to prevent starvation in blocking applications. Note that the thread pool also knows when the code in a managed blocking block is complete–so it will remove the spare worker thread at that point, which means that the pool will shrink back down to its expected size.

Later on, the `Await` object used to ensure that `display` is executed on the calling thread–`Await.result` simply forces the current thread to wait until the future that it is passed is completed. (This uses managed blocking internally.) Here, `Await.result` must be used to block on `futImages` so as to prevent the program's main thread from completing before `imagesFuts` or `futImages` completes.

2.2 FlowPools

In our treatment of futures, we focused mainly on applying individual asynchronous operations to future values. We alluded to the possibility of chaining together composable operations on futures in order to build up rich computation graphs. Such chaining amounts to building up a directed acyclic graph (DAG) of computations and can be viewed as a sort of asynchronous dataflow.

Thus, from a bird's eye view, one can think of Scala's futures as *single-element* asynchronous dataflow, capable of building up rich and interesting DAGs of computation.

FlowPools are a fundamental dataflow *collections* abstraction which can be used as a building block for larger and more complex *deterministic* and parallel dataflow programs. That is, one can think of FlowPools as a fully asynchronous pool-like collection of individually asynchronous elements like futures.

Our FlowPool abstraction is backed by an efficient non-blocking data structure. As a result, our data structure benefits from the increased robustness provided by lock-freedom [Herlihy, 1990], since its operations are not blocked by delayed threads. We provide a lock-freedom proof, which guarantees progress regardless of the behavior, including the failure, of concurrent threads.

In combining lock-freedom with a functional interface, we go on to show that FlowPools, like futures, are *composable*. That is, using prototypical higher-order functions such as `foreach` and `aggregate`, one can concisely form dataflow graphs, in which associated functions are executed asynchronously in a completely non-blocking way, as elements of FlowPools in the dataflow graph become available.

Finally, we present how FlowPools are able to overcome practical issues, such as out-of-memory errors, thus enabling programs based upon FlowPools to run indefinitely. By using a *builder* abstraction, instead of something like iterators or streams (which can lead to non-determinism) we are able to garbage collect parts of the data structure we no longer need, thus reducing memory consumption.

This chapter outlines the following contributions:

1. The design and Scala implementation[2] of a parallel dataflow abstraction and underlying data structure that is deterministic, lock-free, & composable.

2. Proofs of lock-freedom, linearizability, and determinism.

3. Detailed benchmarks comparing the performance of our FlowPools against other popular concurrent data structures.

[2] See https://github.com/heathermiller/scala-dataflow

2.2.1 Model of Computation

FlowPools are similar to a typical collections abstraction. Operations invoked on a FlowPool are executed on its individual elements. However, FlowPools do not only act as a data container of elements. Unlike a typical collection, FlowPools also act as nodes and edges of a directed acyclic computation graph (DAG), in which the executed operations are registered with the FlowPool.

Nodes in this DAG are data containers which are first class values. This makes it possible to use FlowPools as function arguments or to receive them as return values. Edges, on the other hand, can be thought of as combinators or higher-order functions whose user-defined functions are the previously-mentioned operations that are registered with the FlowPool. In addition to providing composability, this means that the DAG does not have to be specified at compile time, but can be generated dynamically at run time instead.

This structure allows for complete asynchrony, allowing the runtime to extract parallelism as a result. That is, elements can be asynchronously inserted, all registered operations can be asynchronously executed, and new operations can be asynchronously registered. Put another way, invoking several higher-order functions in succession on a given FlowPool does not add barriers between nodes in the DAG, it only extends the DAG. This means that individual elements within a FlowPool can *flow* through different edges of the DAG independently.

To illustrate this, let's examine a simple example, visualizing how elements are processed within a FlowPool as compared to how elements within a parallel collection like `ParVector` are processed. Let's assume we have a collection (either a FlowPool or a `ParVector`) full of users of some social network. Ultimately, we would like to obtain a list of each user's friends who are the same age as the user:

```
val users: FlowPool[User] = ... // consider also ParVector[Users]
users.map( user => (user.age, user.getFriends() ) ) // network call, long-running
     .map {
       case (userAge, friends) =>
         friends.filter(friend => userAge == friend.age)
     }
```

While the programming interface remains identical across FlowPools and parallel collections, elements are processed differently between the two. The differences are illustrated in Figures 2.4 and 2.5. As is depicted for parallel collections in Figure 2.4, elements are processed with barriers between stages, that is, the processing of all elements in the first stage (the `users` parallel collection) must be complete before processing in the second stage (the first `map` operation) can even begin. FlowPools on the other hand, as depicted in Figure 2.5, remove such barriers–when an element in `users` is finished being processed, the first `map` operation can be applied to that element individually, we need not wait until all other elements are

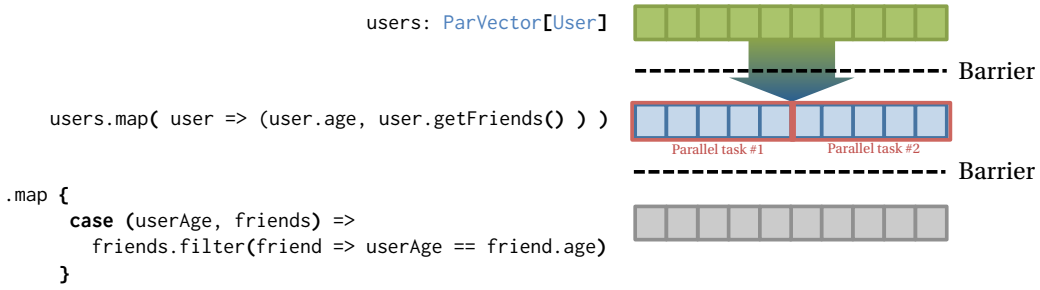


Figure 2.4 – Other collections, such as parallel collections, have barriers between nodes in the DAG. This means that all parallel computation happens only on the individual nodes (collections) meaning there is no parallelism *between* nodes in the DAG.

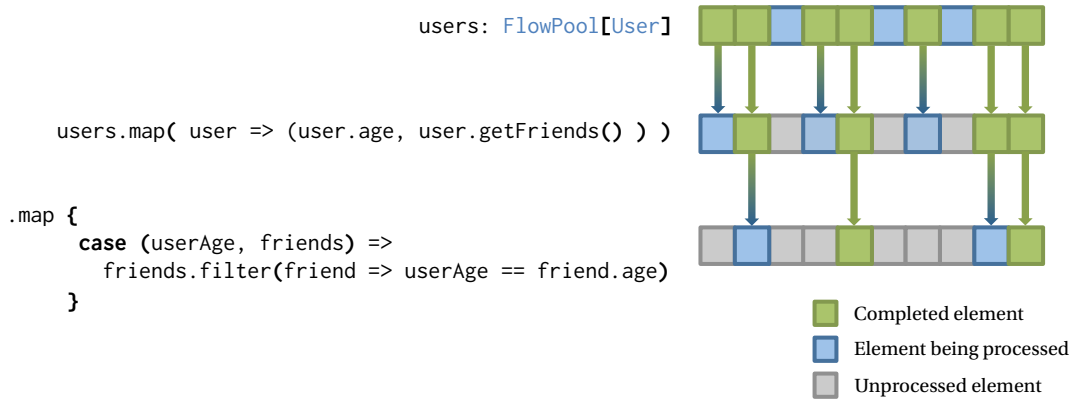


Figure 2.5 – FlowPools are fully asynchronous and barrier-free between nodes in the DAG. This means that parallel computation can happen both on the individual node (within the same collection) as well as *between* nodes (collections) along edges in the DAG.

completed in the first stage (in the `users` FlowPool) before beginning the second stage of processing (the first `map` operation). We thus refer to FlowPools as *barrier-free* between nodes in the computation DAG.

Properties of FlowPools. FlowPools have certain properties which ensure that resulting programs are deterministic.

1. Single-assignment - an element added to the FlowPool cannot be removed.
2. No order - data elements in FlowPools are unordered.
3. Purity - traversals are side-effect free (pure), except when invoking FlowPool operations.
4. Liveness - callbacks are eventually asynchronously executed on all elements.

We claim that FlowPools are deterministic in the sense that all execution schedules either lead to some form of non-termination (*e.g.*, some exception), or the program terminates and no

difference can be observed in the final state of the resulting data structures. This definition is practically useful, because in the case of non-termination it is guaranteed that on some thread an exception is thrown which aids debugging, *e.g.*, , by including a stack trace. For a more formal definition and proof of determinism, see section 2.2.4.

2.2.2 Programming Interface

A FlowPool can be thought of as a concurrent pool data structure, *i.e.*, it can be used similarly to a collections abstraction, complete with higher-order functions, or combinators, for composing computations on FlowPools. In this section, we describe the semantics of several of those functional combinators and other basic operations defined on FlowPools.

Append (<<). The most fundamental of all operations on FlowPools is the concurrent thread-safe append operation. As its name suggests, it simply takes an argument of type `Elem` and appends it to a given FlowPool.

Foreach and Aggregate. A pool containing a set of elements is of little use if its elements cannot be manipulated in some manner. One of the most basic data structure operations is element traversal, often provided by iterators or streams– stateful objects which store the current position in the data structure. However, since their state can be manipulated by several threads at once, using streams or iterators can result in nondeterministic executions.

Another way to traverse the elements is to provide a higher-order `foreach` operator which takes a user-specified function as an argument and applies it to every element. For it to be deterministic, it must be called for every element that is eventually inserted into the FlowPool, rather than only on those present when `foreach` is called. Furthermore, determinism still holds even if the user-specified function contains side-effecting FlowPool operations such as <<. For `foreach` to be non-blocking, it cannot wait until additional elements are added to the FlowPool. Thus, the `foreach` operation must execute asynchronously, and be eventually applied to every element. Its signature is `def foreach[U](f: T => U): Future[Int]`, and its return type `Future[Int]` is an integer value which becomes available once `foreach` traverses all the elements added to the pool. This integer denotes the number of times the `foreach` has been called.

The `aggregate` operation aggregates the elements of the pool and has the following signature: `def aggregate[S](zero: =>S)(cb: (S, S) => S)(op: (S, T) => S): Future[S]`, where `zero` is the initial aggregation, `cb` is an associative operator which combines several aggregations, `op` is an operator that adds an element to the aggregation, and `Future[S]` is the final aggregation of all the elements which becomes available once all the elements have been added. The `aggregate` operator divides elements into subsets and applies the aggregation operator `op` to aggregate elements in each subset starting from the `zero` aggregation, and then combines different subset aggregations with the `cb` operator. In essence, the first part of `aggregate` defines the commutative monoid and the functions involved must be non-side-effecting. In contrast, the operator `op` is guaranteed to be called only once per element and it

can have side-effects.

While in an imperative programming model, `foreach` and `aggregate` are equivalent in the sense that one can be implemented in terms of the other, in a single-assignment programming model `aggregate` is more expressive. The `foreach` operation can be implemented using `aggregate`, but not vice versa.

Builders. The FlowPool described so far must maintain a reference to all the elements at all times to implement the `foreach` operation correctly. Since elements are never removed, the pool may grow indefinitely and run out of memory. However, it is important to note that appending new elements does not necessarily require a reference to any of the existing elements. This observation allows us to move the << operation out of the FlowPool and into a different abstraction called a `builder`. Thus, a typical application starts by registering all the `foreach` operations, and then it releases the references to FlowPools, leaving only references to builders. In a managed environment, the GC then can automatically discard the no longer needed objects.

Seal. After deciding that no more elements will be added, further appends can be disallowed by calling `seal`. This has the advantage of discarding the registered `foreach` operations. More importantly, the `aggregate` can complete its future– this is only possible once it is known there will be no more appends.

Simply preventing append calls after the point when `seal` is called, however, yields a nondeterministic programming model. Imagine a thread that attempts to seal the pool executing concurrently with a thread that appends an element. In one execution, the append can precede the seal, and in the other the append can follow the seal, causing an error. To avoid nondeterminism, there has to be an agreement on the current state of the pool. A convenient and sufficient way to make `seal` deterministic is to provide the expected pool size as an argument. The semantics of `seal` is such that it fails if the pool is already sealed with a different size or the number of elements is greater than the desired size. Note that we do not guarantee that the same exception always occurs on the same thread– rather, if *any* thread throws *some* exception in *some* execution schedule, then in *all* execution schedules *some* thread will throw *some* exception.

Higher-order operators. We now show how these basic abstractions can be used to build higher-order abstractions. To start, it is convenient to have generators that create certain pool types. In a dataflow graph, FlowPools created by generators can be thought of as source nodes. As an example, `tabulate` (below) creates a sequence of elements by applying a user-specified function `f` to natural numbers. One can imagine more complex generators, which add elements from a network socket or a file, for example.

```
def tabulate[T]
  (n: Int, f: Int => T) {
  val p = new FlowPool[T]
  val b = p.builder
  def recurse(i: Int) {
    b << f(i)
    if i < n recurse(i + 1)
  }
  future { recurse(0) }
  p
}
```

```
def map[S](f: T => S) {
  val p = new FlowPool[S]
  val b = p.builder
  for (x <- this) {
    b << f(x)
  } map {
    sz => b.seal(sz)
  }
  p
}
```

```
def foreach[U](f: T => U) {
  aggregate(0)(_ + _) {
    (acc, x) =>
    f(x)
    acc + 1
  }
}
```

The `tabulate` generator starts by creating a FlowPool of an arbitrary type `T` and creating its builder instance. It then starts an asynchronous computation using the `future` construct (Appendix A for explanation and examples), which recursively applies `f` to each number and adds it to the builder. The reference to the pool `p` is returned *immediately*, before the asynchronous computation completes.

A typical higher-order collection operator `map` is used to map each element of a dataset to produce a new dataset. This corresponds to chaining or pipelining the dataflow graph nodes. Operator `map` traverses the elements of `this` FlowPool and appends each mapped element to the builder. The `for` loop is syntactic sugar for calling the `foreach` method on `this`. We assume that the `foreach` return type `Future[Int]` has `map` and `flatMap` operations, executed once the future value becomes available. The `Future.map` above ensures that once the current pool (`this`) is sealed, the mapped pool is sealed to the appropriate size.

As argued before, `foreach` can be expressed in terms of `aggregate` by accumulating the number of elements and invoking the callback `f` each time. However, some patterns cannot be expressed in terms of `foreach`. The `filter` combinator filters out the elements for which a specified predicate does not hold. Appending the elements to a new pool can proceed as before, but the seal needs to know the exact number of elements added– thus, the `aggregate` accumulator is used to track the number of added elements.

```
def filter
  (pred: T => Boolean) {
  val p = new FlowPool[T]
  val b = p.builder
  aggregate(0)(_ + _) {
    (acc, x) => if pred(x){
      b << x
      1
    } else 0
  } map {sz => b.seal(sz)}
  p
}
```

```
def flatMap[S]
  (f: T => FlowPool[S]) {
  val p = new FlowPool[S]
  val b = p.builder
  aggregate(future(0))(add){
    (af, x) =>
    val sf = for (y <- f(x))
      b << y
    add(af, sf)
  } map {sz => b.seal(sz)}
  p
}

def add(f: Future[Int], g: Future[Int]) =
  for (a <- f; b <- g) yield a + b
```

```
def union[T]
  (that: FlowPool[T]) {
  val p = new FlowPool[T]
  val b = p.builder
  val f =
    for (x <- this) b << x
  val g =
    for (y <- that) b << y
  for (s1 <- f; s2 <- g)
    b.seal(s1 + s2)
  p
}
```

The `flatMap` operation retrieves a pool for each element of `this` pool and adds its elements to the resulting pool. Given two FlowPools, it can be used to generate the Cartesian product of their elements. The implementation is similar to that of `filter`, but we reduce the size on the future values of the sizes– each intermediate pool may not yet be sealed. The operation `q union r`, as one might expect, produces a new pool which has elements of both pool `q` and pool `r`.

The last two operations correspond to joining nodes in the dataflow graph. Note that if we could somehow merge the two different `foreach` loops to implement the third join type `zip`, `zip` would be nondeterministic. The programming model does not allow us to do this, however. The `zip` function is better suited for data structures with deterministic ordering, such as Oz streams, which would in turn have a nondeterministic `union`.

```
type Terminal {
  sealed: Int
  callbacks:
    List[Elem => Unit]
}

type Elem
```

```
type Block {
  array: Array[Elem]
  next: Block
  index: Int
  blockindex: Int
}
```

```
type FlowPool {
  start: Block
  current: Block
}
LASTELEMPOS = BLOCKSIZE - 2
NOSEAL = -1
```

2.2.3 Implementation

We now describe the FlowPool and its basic operations. In doing so, we omit the details not relevant to the algorithm[3] and focus on a high-level description of a non-blocking data structure. One straightforward way to implement a growing pool is to use a linked list of nodes

[3] Specifically the builder abstraction and the `aggregate` operation. The `aggregate` can be implemented using `foreach` with a side-effecting accumulator.

```
1  def create()
2    new FlowPool {
3      start = createBlock(0)
4      current = start
5    }
6
7  def createBlock(bidx: Int)
8    new Block {
9      array = new Array(BLOCKSIZE)
10     index = 0
11     blockindex = bidx
12     next = null
13   }
14
15 def append(elem: Elem)
16   b = READ(current)
17   idx = READ(b.index)
18   nexto = READ(b.array(idx + 1))
19   curo = READ(b.array(idx))
20   if check(b, idx, curo) {
21     if CAS(b.array(idx + 1), nexto, curo) {
22       if CAS(b.array(idx), curo, elem) {
23         WRITE(b.index, idx + 1)
24         invokeCallbacks(elem, curo)
25       } else append(elem)
26     } else append(elem)
27   } else {
28     advance()
29     append(elem)
30   }
31
32 def check(b: Block, idx: Int, curo: Object)
33   if idx > LASTELEMPOS return false
34   else curo match {
35     elem: Elem =>
36       return false
37     term: Terminal =>
38       if term.sealed = NOSEAL return true
39       else {
40         if totalElems(b, idx) < term.sealed
41           return true
42         else error("sealed")
43       }
44     null =>
45       error("unreachable")
46   }
47
48 def advance()
49   b = READ(current)
50   idx = READ(b.index)
51   if idx > LASTELEMPOS
52     expand(b, b.array(idx))
53   else {
54     obj = READ(b.array(idx))
55     if obj is Elem WRITE(b.index, idx + 1)
56   }
57
58 def expand(b: Block, t: Terminal)
59   nb = READ(b.next)
60   if nb is null {
61     nb = createBlock(b.blockindex + 1)
62     nb.array(0) = t
63     if CAS(b.next, null, nb)
64       expand(b, t)
65   } else {
66     CAS(current, b, nb)
67   }
68 def totalElems(b: Block, idx: Int)
69   return b.blockindex * (BLOCKSIZE - 1) + idx
70
71 def invokeCallbacks(e: Elem, term: Terminal)
72   for (f <- term.callbacks) future {
73     f(e)
74   }
75
76 def seal(size: Int)
77   b = READ(current)
78   idx = READ(b.index)
79   if idx <= LASTELEMPOS {
80     curo = READ(b.array(idx))
81     curo match {
82       term: Terminal =>
83         if ¬tryWriteSeal(term, b, idx, size)
84           seal(size)
85       elem: Elem =>
86         WRITE(b.index, idx + 1)
87         seal(size)
88       null =>
89         error("unreachable")
90     }
91   } else {
92     expand(b, b.array(idx))
93     seal(size)
94   }
95
96 def tryWriteSeal(term: Terminal, b: Block,
97   idx: Int, size: Int)
98   val total = totalElems(b, idx)
99   if total > size error("too many elements")
100  if term.sealed = NOSEAL {
101    nterm = new Terminal {
102      sealed = size
103      callbacks = term.callbacks
104    }
105    return CAS(b.array(idx), term, nterm)
106  } else if term.sealed ≠ size {
107    error("already sealed with different size")
108  } else return true
109
110 def foreach(f: Elem => Unit)
111   future {
112     asyncFor(f, start, 0)
113   }
114
115 def asyncFor(f: Elem => Unit, b: Block, idx: Int)
116   if idx <= LASTELEMPOS {
117     obj = READ(b.array(idx))
118     obj match {
119       term: Terminal =>
120         nterm = new Terminal {
121           sealed = term.sealed
122           callbacks = f ∪ term.callbacks
123         }
124         if ¬CAS(b.array(idx), term, nterm)
125           asyncFor(f, b, idx)
126       elem: Elem =>
127         f(elem)
128         asyncFor(f, b, idx + 1)
129       null =>
130         error("unreachable")
131     }
132   } else {
133     expand(b, b.array(idx))
134     asyncFor(f, b.next, 0)
135   }
```


Figure 2.6 – FlowPool operations pseudocode

that wrap elements. Since we are concerned about the memory footprint and cache-locality, we store the elements into arrays instead, which we call blocks. Whenever a block becomes full, a new block is allocated and the previous block is made to point to the `next` block. This way, most writes amount to a simple array-write, while allocation occurs only occasionally. Each block contains a hint `index` to the first free entry in the array, i.e. one that does not contain an element. An `index` is a hint, since it may actually reference an entry that comes earlier than the first free entry. Additionally, a FlowPool also maintains a reference to the first block called `start`. It also maintains a hint to the last block in the chain of blocks, called `current`. This reference may not always be up-to-date, but it always points to some block in the chain.

Each FlowPool is associated with a list of callbacks which have to be called in the future as new elements are added. Each FlowPool can also be in a sealed state, meaning there is a bound on the number of elements it can have. This information is stored as a `Terminal` value in the first free array entry. At all times, we maintain the invariant that the array in each block starts with a sequence of elements, followed by a `Terminal` delimiter. From a higher-level perspective, appending an element starts by copying the `Terminal` value to the next entry and then overwriting the current entry with the element being appended.

The `append` operation starts by reading the `current` block and the `index` of the free position. It then reads `nexto` after the first free entry, followed by a read of the `curo` at the free entry. The `check` procedure checks the conditions of the bounds, whether the FlowPool was already sealed or if the current array entry contains an element. In either of these events, the `current` and `index` values need to be set– this is done in the `advance` procedure. We call this the **slow path** of the `append` method. Notice that there are several situations which trigger the slow path. For example, if some other thread completes the `append` method but is preempted before updating the value of the hint `index`, then the `curo` will have the type `Elem`. The same happens if a preempted thread updates the value of the hint `index` after additional elements have been added, via unconditional write in line 158. Finally, reaching an end of block triggers the slow path.

Otherwise, the operation executes the **fast path** and appends an element. It first copies the `Terminal` value to the next entry with a CAS instruction in line 156, with `nexto` being the expected value. If it fails (e.g. due to a concurrent CAS), the append operation is restarted. Otherwise, it proceeds by writing the element to the current entry with a CAS in line 157, the expected value being `curo`. On success, it updates the `b.index` value and invokes all the callbacks (present when the element was added) with the `future` construct. In the implementation, we do not schedule an asynchronous computation for each element. Instead, the callback invocations are batched to avoid the scheduling overhead– the array is scanned for new elements until the first free entry is reached.

Interestingly, note that inverting the order of the reads in lines 153 and 154 would cause a race in which a thread could overwrite a `Terminal` value with some older `Terminal` value if some

other thread appended an element in between.

The `seal` operation continuously increases the `index` in the block until it finds the first free entry. It then tries to replace the `Terminal` value there with a new `Terminal` value which has the seal size set. An error occurs if a different seal size is set already. The `foreach` operation works in a similar way, but is executed asynchronously. Unlike `seal`, it starts from the first element in the pool and calls the callback for each element until it finds the first free entry. It then replaces the `Terminal` value with a new `Terminal` value with the additional callback. From that point on the append method is responsible for scheduling that callback for subsequently added elements. Note that all three operations call `expand` to add an additional block once the current block is empty, to ensure lock-freedom.

Multi-Lane FlowPools. Using a single block sequence (i.e. lane) to implement a FlowPool does not take full advantage of the lack of ordering guarantees and may cause slowdowns due to collisions when multiple concurrent writers are present. Multi-Lane FlowPools overcome this limitation by having a lane for each CPU, where each lane has the same implementation as the normal FlowPool.

This has several implications. First of all, CAS failures during insertion are avoided to a high extent and memory contention is decreased due to writes occurring in different cache-lines. Second, `aggregate` callbacks are added to each lane individually and aggregated once all of them have completed. Finally, `seal` needs to be globally synchronized in a non-blocking fashion.

Once `seal` is called, the remaining free slots are split amongst the lanes equally. If a writer finds that its lane is full, it writes to some other lane instead. This raises the frequency of CAS failures, but in most cases happens only when the FlowPool is almost full, thus ensuring that the append operation scales.

2.2.4 Correctness

We give an outline of the correctness proof here. More formal definitions, and the full lock-freedom proof can be found in Appendix A. Linearizability and determinism proofs can be can be found in the companion technical report [Prokopec et al., 2012b].

We define the notion of an abstract pool $\mathbb{A} = (elems, callbacks, seal)$ of elements in the pool, callbacks and the seal size. Given an abstract pool, abstract pool operations produce a new abstract pool. The key to showing correctness is to show that an abstract pool operation corresponds to a FlowPool operation– that is, it produces a new abstract pool corresponding to the state of the FlowPool after the FlowPool operation has been completed.

Lemma 2.2.1. *Given a FlowPool consistent with some abstract pool, CAS instructions in lines 156, 198 and 201 do not change the corresponding abstract pool.*

Lemma 2.2.2. *Given a FlowPool consistent with an abstract pool* $(elems, cbs, seal)$*, a successful CAS in line 157 changes it to the state consistent with an abstract pool* $(\{elem\} \cup$

$t ::=$	terms	$p \in \{(vs, \sigma, cbs) \mid vs \subseteq Elem, \sigma \in \{-1\} \cup \mathbb{N},$
`create` p	pool creation	$cbs \subset Elem \Rightarrow Unit\}$
p << v	append	$v \in Elem$
p `foreach` f	foreach	$f \in Elem \Rightarrow Unit$
p `seal` n	seal	$n \in \mathbb{N}$
t_1 ; t_2	sequence	

Figure 2.7 – Syntax

$elems, cbs, seal$). *There exists a time* $t_1 \geq t_0$ *at which every callback* $f \in cbs$ *has been called on* $elem$.

Lemma 2.2.3. *Given a FlowPool consistent with an abstract pool* $(elems, cbs, seal)$*, a successful CAS in line 259 changes it to the state consistent with an abstract pool* $(elems, (f, \emptyset) \cup cbs, seal)$ *There exists a time* $t_1 \geq t_0$ *at which* f *has been called for every element in* $elems$.

Lemma 2.2.4. *Given a FlowPool consistent with an abstract pool* $(elems, cbs, seal)$*, a successful CAS in line 240 changes it to the state consistent with an abstract pool* $(elems, cbs, s)$*, where either* $seal = -1 \wedge s \in \mathbb{N}_0$ *or* $seal \in \mathbb{N}_0 \wedge s = seal$.

Theorem 2.2.1 (Safety)**.** *Operations* `append`, `foreach` *and* `seal` *are consistent with the abstract pool semantics.*

Theorem 2.2.2 (Linearizability)**.** *Operations* `append` *and* `seal` *are linearizable.*

Lemma 2.2.5. *After invoking a FlowPool operation* `append`, `seal` *or* `foreach`*, if a non-consistency changing CAS in lines 156, 198, or 201 fails, they must have already been completed by another thread since the FlowPool operation began.*

Lemma 2.2.6. *After invoking a FlowPool operation* `append`, `seal` *or* `foreach`*, if a consistency changing CAS in lines 157, 240, or 259 fails, then some thread has successfully completed a consistency changing CAS in a finite number of steps.*

Lemma 2.2.7. *After invoking a FlowPool operation* `append`, `seal` *or* `foreach`*, a consistency changing instruction will be completed after a finite number of steps.*

Theorem 2.2.3 (Lock-freedom)**.** *FlowPool operations* `append`, `foreach` *and* `seal` *are lock-free.*

Determinism. We claim that the FlowPool abstraction is *deterministic* in the sense that a program computes the same result (possibly an error) regardless of the interleaving of execution steps. Here we give an outline of the determinism proof. A complete formal proof can be found in the technical report [Prokopec et al., 2012b].

The following definitions and the determinism theorem are based on the language shown in Figure A.2. The semantics of our core language is defined using reduction rules which define transitions between *execution states*. An execution state is a pair $T \mid P$ where T is a set of concurrent threads and P is a set of FlowPools. Each thread executes a *term* of the core

language (typically a sequence of terms). State of a thread is represented as the (rest of) the term that it still has to execute; this means there is a one-to-one mapping between threads and terms. For example, the semantics of append is defined by the following reduction rule (a complete summary of all the rules can be found in the appendix):

$$\frac{t = p << v\ ;\ t' \quad p = (vs, cbs, -1) \quad p' = (\{v\} \cup vs, cbs, -1)}{t, T, p, P \longrightarrow t', T, p', P} \qquad \text{(APPEND1)}$$

Append simply adds the value v to the pool p, yielding a modified pool p'. Note that this rule can only be applied if the pool p is not sealed (the seal size is -1). The rule for $foreach$ modifies the set of callback functions in the pool:

$$\frac{\begin{array}{c} t = p \texttt{ foreach } f\ ;\ t' \quad p = (vs, cbs, n) \\ T' = \{g(v) \mid g \in \{f\} \cup cbs, v \in vs\} \quad p' = (vs, \{f\} \cup cbs, n) \end{array}}{t, T, p, P \longrightarrow t', T, T', p', P} \qquad \text{(FOREACH2)}$$

This rule only applies if p is sealed at size n, meaning that no more elements will be appended later. Therefore, an invocation of the new callback f is scheduled for each element v in the pool. Each invocation creates a new thread in T'.

Programs are built by first creating one or more FlowPools using `create`. Concurrent threads can then be started by (a) appending an element to a FlowPool, (b) sealing the FlowPool and (c) registering callback functions (`foreach`).

Definition 2.2.1 (Termination)**.** *A term t terminates with result P if its reduction ends in execution state $\{t : t = \{\epsilon\}\} \mid P$.*

Definition 2.2.2 (Interleaving)**.** *Consider the reduction of a term t: $T_1 \mid P_1 \longrightarrow T_2 \mid P_2 \longrightarrow \ldots \longrightarrow \{t : t = \{\epsilon\}\} \mid P_n$. An* interleaving *is a reduction of t starting in $T_1 \mid P_1$ in which reduction rules are applied in a different order.*

Definition 2.2.3 (Determinism)**.** *The reduction of a term t is* deterministic iff *either (a) t does not terminate for any interleaving, or (b) t always terminates with the same result for all interleavings.*

Theorem 2.2.4 (FlowPool Determinism)**.** *Reduction of terms t is deterministic.*

2.2.5 Evaluation

We evaluate our implementation (single-lane and multi-lane FlowPools) against the LinkedTransferQueue [III et al., 2009] for all benchmarks and the ConcurrentLinkedQueue [Michael and Scott, 1996] for the insert benchmark, both found in JDK 1.7, on three different architectures; a quad-core 3.4 GHz i7-2600, 4x octa-core 2.27 GHz Intel Xeon x7560 (both with hyperthreading) and an octa-core 1.2GHz UltraSPARC T2 with 64 hardware threads. In this

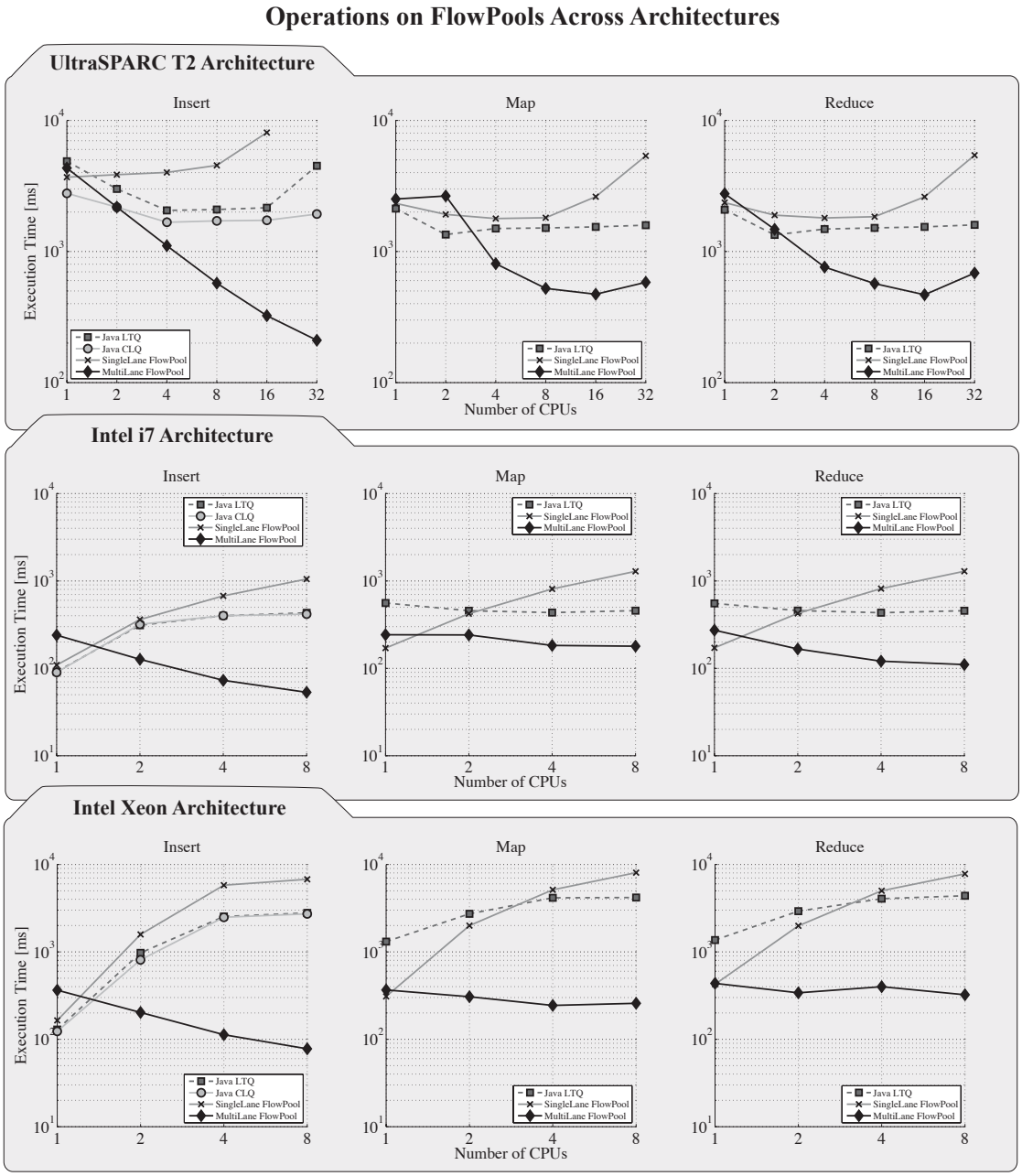


Figure 2.8 – Execution time vs parallelization across three different architectures on three important FlowPool operations; insert, map, reduce.

section, we focus on the scaling properties of the above-mentioned data structures, Figures 2.8 & 2.9.

In the *Insert* benchmark, Figure 2.8, we evaluate concurrent insert operations, by distributing the work of inserting N elements into the data structure concurrently across P threads. In Figure 2.8, it's evident that both single-lane FlowPools and concurrent queues do not scale well with the number of concurrent threads, particularly on the i7 architecture. They quickly slow down, likely due to cache line collisions and CAS failures. On the other hand, multi-

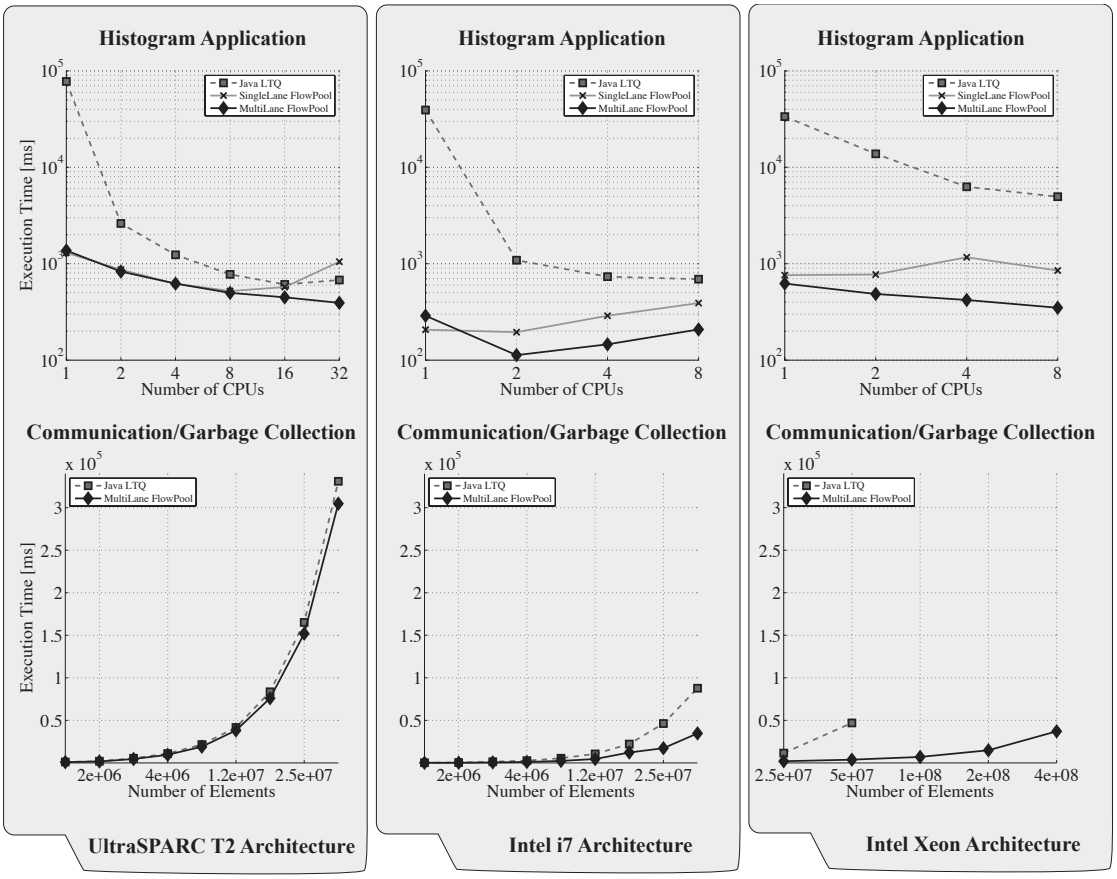


Figure 2.9 – Execution time vs parallelization on a real histogram application (top), & communication benchmark (bottom) showing memory efficiency, across all architectures.

lane FlowPools scale well, as threads write to different lanes, and hence different cache lines, meanwhile also avoiding CAS failures. This appears to reduce execution time for insertions up to 54% on the i7, 63% on the Xeon and 92% on the UltraSPARC.

The performance of higher-order functions is evaluated in the *Reduce, Map* (both in Figure 2.8) and *Histogram* benchmarks (Figure 2.9). It's important to note that the *Histogram* benchmark serves as a "real life" example, which uses both the `map` and `reduce` operations that are benchmarked in Figure 2.8. Also note that in all of these benchmarks, the time it takes to insert elements into the FlowPool is also measured, since the FlowPool programming model allows one to insert elements concurrently with the execution of higher-order functions.

In the *Histogram* benchmark, Figure 2.9, P threads produce a total of N elements, adding them to the FlowPool. The `aggregate` operation is then used to produce 10 different histograms concurrently with a different number of bins. Each separate histogram is constructed by its own thread (or up to P, for multi-lane FlowPools). A crucial difference between queues and FlowPools here, is that with FlowPools, multiple histograms are produced by invoking several `aggregate` operations, while queues require writing each element to several queues– one for each histogram. Without additional synchronization, reading a single queue is not an option,

since elements have to be removed from the queue eventually, and it is not clear to each reader when to do this. With FlowPools, elements are automatically garbage collected when no longer needed.

Finally, to validate the last claim of garbage being automatically collected, in the *Communication/Garbage Collection* benchmark, Figure 2.9, we create a pool in which a large number of elements N are added concurrently by P threads. Each element is then processed by one of P threads through the use of the `aggregate` operation. We benchmark against linked transfer queues, where P threads concurrently remove elements from the queue and process it. For each run, we vary the size of the N and examine its impact on the execution time. Especially in the cases of the Intel architectures, the multi-lane FlowPools perform considerably better than the linked transfer queues. As a matter of fact, the linked transfer queue on the Xeon benchmark ran out of memory, and was unable to complete, while the multi-lane FlowPool scaled effortlessly to 400 million elements, indicating that unneeded elements are properly garbage collected.

2.3 Related Work

An introduction to linearizability and lock-freedom is given by Herlihy and Shavit [Herlihy and Shavit, 2008]. A detailed overview of concurrent data structures is given by Moir and Shavit [Moir and Shavit, 2005]. To date, concurrent data structures remain an active area of research– we restrict this summary to those relevant to this work.

Concurrently accessible queues have been present for a while, an implementation is described by [Mellor-Crummey, 1987]. Non-blocking concurrent linked queues are described by Michael and Scott [Michael and Scott, 1996]. This CAS-based queue implementation is cited and used widely today, a variant of which is present in the Java standard library. More recently, Scherer, Lea and Scott [III et al., 2009] describe synchronous queues which internally hold both data and requests. Both approaches above entail blocking (or spinning) at least on the consumer's part when the queue is empty.

While the abstractions above fit well in the concurrent imperative model, they have the disadvantage that the programs written using them are inherently nondeterministic. Roy and Haridi [Roy and Haridi, 2004] describe the Oz programming language, a subset of which yields programs deterministic by construction. Oz dataflow streams are built on top of single-assignment variables and are deterministically ordered. They allow multiple consumers, but only one producer at a time. Oz has its own runtime which implements blocking using continuations.

The concept of single-assignment variables is used to provide logical variables in concurrent logic programming languages [Shapiro, 1989]. It is also embodied in futures proposed by Baker and Hewitt [Henry C. Baker and Hewitt, 1977], and promises first mentioned by Friedman and Wise [Friedman and Wise, 1976]. Futures were first implemented in MultiLISP [Halstead, 1985],

and have been employed in many languages and frameworks since. Futures have been generalized to data-driven futures, which provide additional information to the scheduler [Tasirlar and Sarkar, 2011]. Many frameworks have constructs that start an asynchronous computation and yield a future holding its result, for example, Habanero Java [Budimlic et al., 2011] (`async`) and Scala [Odersky et al., 2010] (`future`).

A number of other models and frameworks recognized the need to embed the concept of futures into other data-structures. Single-assignment variables have been generalized to I-Structures [Arvind et al., 1989] which are essentially single-assignment arrays. CnC [Budimlic et al., 2010, Burke et al., 2011] is a parallel programming model influenced by dynamic dataflow, stream-processing and tuple spaces [Gelernter, 1985]. In CnC the user provides high-level operations along with the ordering constraints that form a computation dependency graph. FlumeJava [Chambers et al., 2010a] is a distributed programming model which relies heavily on the concept of collections containing futures. An issue that often arises with dataflow programming models are unbalanced loads. This is often solved using bounded buffers which prevent the producer from overflowing the consumer.

Opposed to the correct-by-construction determinism described thus far, a type-systematic approach can also ensure that concurrent executions have deterministic results. Recently, work on Deterministic Parallel Java showed that a region-based type system can ensure determinism [Jr. et al., 2009]. X10's constrained-based dependent types can similarly ensure determinism and deadlock-freedom [Saraswat et al., 2007].

LVars [Kuper and Newton, 2013] are a generalization of single-assignment variables to multiple-assignment that are provably deterministic in a concurrent setting. LVars are based on a lattice and ensure determinism by allowing only monotonic writes and threshold reads. LVars were extended with freezing and handlers [Kuper et al., 2014] resembling some of the capabilities and interface of FlowPools. FlowPools differ in that they use many of the same properties (monotonicity, callbacks, and sealing) to solve a slightly different problem–FlowPools aim to provide a deterministic multiset or pool abstraction of single-assignment variables and provably non-blocking implementation. LVars aim to be a foundation for ensuring determinism based on lattices, which can be realized as a number of different types of data structures such as single variables, sets, or maps.

CRDTs [Shapiro et al., 2011a,b] are data structures with specific well-definied properties designed for replicating data across multiple machines in a distributed system. While generally useful for a different purpose (FlowPools don't aim to replicate state across a network, instead they intend to be deterministic in the face of concurrent writes), both FlowPools and CRDTs share the need for monotonic updates. One convenience FlowPools have over CRDTs is their composability. However, Lasp [Meiklejohn and Van Roy, 2015] attempts to remedy this limitation of CRDTs through a new programming model designed for building convergent computations by composing CRDTs.

2.4 Conclusion

In this chapter, we've presented two libraries and abstractions for asynchronous dataflow programming. Futures in Scala, are a fully asynchronous and non-blocking futures and promises library with a monadic interface that enables operations on futures to be composed. FlowPools are an abstraction for concurrent dataflow programming that also provides a composable programming model similar to futures. We showed that FlowPools are provably deterministic and can be implemented in a provably non-blocking manner. Finally, we showed that in addition to having a richer more functional interface than other similar data structures in Java's concurrency library, FlowPools are also efficient and out perform a number of standard Java concurrent data structures in our benchmarks.

3 Pickling

Central to a distributed application is the need to communicate with the outside world. However, in order to do this, data must be transformed from an in-memory representation to one that can be sent over the network, *e.g.,* a binary representation. This act of transforming in-memory data to some form of external representation is called *pickling* or *serialization*.

This chapter covers a new approach to this foundational aspect of distributed programming. In this chapter, we detail *object oriented picklers* and scala/pickling, a framework for generating them at compile time.

3.1 Introduction

As more and more traditional applications migrate to the cloud, the demand for interoperability between different services is at an all-time high, and is increasing. At the center of all of this communication – communication that must often take place in various ways, in many formats, even within the same application. However, a central aspect to this communication that has received surprisingly little attention in the literature is the need to serialize, or *pickle* objects, *i.e.,* to persist in-memory data by converting them to a binary, text, or some other representation. As more and more applications develop the need to communicate with different machines or services, providing abstractions and constructs for easy-to-use, typesafe, and performant serialization is becoming more important than ever.

On the JVM, serialization has long been acknowledged as having a high overhead [Carpenter et al., 1999, Welsh and Culler, 2000], with some estimates purporting object serialization to account for 25-65% of the cost of remote method invocation, and which go on to observe that the cost of serialization grows with growing object structures up to 50% [Maassen et al., 1999, Philippsen et al., 2000]. Due to the prohibitive cost of using Java Serialization in high-performance distributed applications, many frameworks for distributed computing, like Akka [Typesafe, 2009], Spark [Zaharia et al., 2012], SCADS [Armbrust et al., 2009], and others, provide support for higher-performance alternative frameworks such as Google's Protocol

Buffers [Google, 2008], Apache Avro [Apache, 2013], or Kryo [Nathan Sweet, 2013]. 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.

This chapter presents object-oriented picklers and *scala/pickling*, a framework for their generation either at runtime or at compile time. The introduced notion of object-oriented pickler combinators extends pickler combinators known from functional programming [Kennedy, 2004] with support for object-oriented concepts such as subtyping, mix-in composition, and object identity in the face of cyclic object graphs. In contrast to pure functional-style pickler combinators, we employ static, type-based meta programming to compose picklers at compile time. The resulting picklers are efficient, since the pickling code is generated statically as much as possible, avoiding the overhead of runtime reflection [Dubochet, 2011, Gil and Maman, 2008].

Furthermore, the presented pickling framework is extensible in several important ways. First, building on an object-oriented type-class-like mechanism [Oliveira et al., 2010], our approach enables retroactively adding pickling support to existing, unmodified types. Second, our framework provides pluggable pickle formats which decouple type checking and pickler composition from the lower-level aspects of data formatting. This means that the type safety guarantees provided by type-specialized picklers are "portable" in the sense that they carry over to different pickle formats.

3.1.1 Design Constraints

The design of our framework has been guided by the following principles:

- **Ease of use**. The programming interface aims to require as little pickling boilerplate as possible. Thanks to dedicated support by the underlying virtual machine, Java's serialization [Oracle, Inc., 2011] requires only little boilerplate, which mainstream Java developers have come to expect. Our framework aims to be usable in production environments, and must, therefore, be able to integrate with existing systems with minimal changes.

- **Performance**. The generated picklers should be efficient enough so as to enable their use in high-performance distributed, "big data", and cloud applications. One factor driving practitioners away from Java's default serialization mechanism is its high runtime overhead compared to alternatives such as Kryo, Google's Protocol Buffers or Apache's Avro serialization framework. However, such alternative frameworks offer only minimal language integration.

- **Extensibility**. It should be possible to add pickling support to existing types retroactively. This resolves a common issue in Java-style serialization frameworks where classes have to be marked as serializable upfront, complicating unanticipated change. Furthermore,

type-class-like extensibility enables pickling also for types provided by the underlying runtime environment (including built-in types), or types of third-party libraries.

- **Pluggable Pickle Formats**. It should be possible to easily swap target pickle formats, or for users to provide their own customized format. It is not uncommon for a distributed application to require multiple formats for exchanging data, for example an efficient binary format for exchanging system messages, or JSON format for publishing feeds. Type-class-like extensibility makes it possible for users to define their own pickle format, and to easily *swap it in* at the use-site.

- **Type safety**. Picklers should be type safe through (a) type specialization and (b) dynamic type checks when unpickling to transition unpickled objects into the statically-typed "world" at a well-defined program point.

- **Robust support for object-orientation**. Concepts such as subtyping and mix-in composition are used very commonly to define regular object types in object-oriented languages. Since our framework does without a separate data type description language (*e.g.*, a schema), it is important that regular type definitions are sufficient to describe the types to-be-pickled. The Liskov substitution principle is used as a guidance surrounding the substitutability of both objects to-be-pickled and first-class picklers. Our approach is also general, supporting object graphs with cycles.

3.1.2 Contributions

This chapter outlines the following contributions:

- An extension to pickler combinators, well-known in functional programming, to support the core concepts of object-oriented programming, namely subtyping polymorphism, open class hierarchies, and object identity.

- A framework based on object-oriented pickler combinators which (a) enables retrofitting existing types with pickling support, (b) supports automatically generating picklers at compile time and at runtime, (c) supports pluggable pickle formats, and (d) does not require changes to the host language or the underlying virtual machine.

- A complete implementation of the presented approach in and for Scala.[1]

- An experimental evaluation comparing the performance of our framework with Java serialization and Kryo on a number of data types used in real-world, large-scale distributed applications and frameworks.

[1] See http://github.com/scala/pickling/

3.2 Overview and Usage

3.2.1 Basic Usage

Scala/pickling was designed so as to require as little boilerplate from the programmer as possible. For that reason, pickling or unpickling an object `obj` of type `Obj` requires simply,

```
import scala.pickling._
val pickle = obj.pickle
val obj2 = pickle.unpickle[Obj]
```

Here, the `import` statement imports scala/pickling, the method `pickle` triggers static pickler generation, and the method `unpickle` triggers static unpickler generation, where `unpickle` is parameterized on `obj`'s precise type `Obj`. Note that not every type has a `pickle` method; it is implemented as an extension method using an *implicit conversion*. This implicit conversion is imported into scope as a member of the `scala.pickling` package.

Implicit conversions. Implicit conversions can be thought of as methods which can be implicitly invoked based upon their type, and whether or not they are present in implicit scope. Implicit conversions carry the `implicit` keyword before their declaration. The `pickle` method is provided using the following implicit conversion (slightly simplified):

```
implicit def PickleOps[T](picklee: T) =
  new PickleOps[T](picklee)

class PickleOps[T](picklee: T) {
  def pickle: Pickle = ...
  ...
}
```

In a nutshell, the above implicit conversion is implicitly invoked, passing object `obj` as an argument, whenever the `pickle` method is invoked on `obj`. The above example can be written in a form where all invocations of implicit methods are explicit, as follows:

```
val pickle = PickleOps[Obj](obj).pickle
val obj2 = pickle.unpickle[Obj]
```

Optionally, a user can import a `PickleFormat`. By default, our framework provides a Scala Binary Format, an efficient representation based on arrays of bytes, though the framework provides other formats which can easily be imported, including a JSON format. Furthermore,

users can easily extend the framework by providing their own `PickleFormats` (see Section 3.4.3).

Typically, the framework generates the required pickler itself inline in the compiled code, using the `PickleFormat` in scope. In the case of JSON, for example, this amounts to the generation of string concatenation code and field accessors for getting runtime values, all of which is inlined, generally resulting in high performance (see Section 5.5).

In rare cases, however, it is necessary to fall back to runtime picklers which use runtime reflection to access the state that is being pickled and unpickled. For example, a runtime pickler is used when pickling instances of a generic subclass of the static class type to-be-pickled.

Using scala/pickling, it's also possible to pickle and unpickle subtypes, even if the pickle and unpickle methods are called using supertypes of the type to-be-pickled. For example,

```
abstract class Person {
  def name: String
}

case class Firefighter(name: String, since: Int)
     extends Person

val ff: Person = Firefighter("Jim", 2005)
val pickle = ff.pickle
val ff2 = pickle.unpickle[Person]
```

In the above example, the runtime type of `ff2` will correctly be `Firefighter`.

This perhaps raises an important concern– what if the type that is passed as a type argument to method `unpickle` is incorrect? In this case, the framework will fail with a runtime exception at the call site of `unpickle`. This is an improvement over other frameworks, which have less type information available at runtime, resulting in wrongly unpickled objects often propagating to other areas of the program before an exception is thrown.

Scala/pickling is also able to unpickle values of static type `Any`. Scala's pattern-matching syntax can make unpickling on less-specific types quite convenient, for example:

```
pickle.unpickle[Any] match {
  case Firefighter(n, _) => println(n)
  case _ => println("not a Firefighter")
}
```

Beyond dealing with subtypes, our pickling framework supports pickling/unpickling most Scala types, including generics, case classes, and singleton objects. Passing a type argument to pickle, whether inferred or explicit, which is an unsupported type leads to a compile-time error. This avoids a common problem in Java-style serialization where non-serializable types are only discovered at runtime, in general.

Function closures, however, are not supported by scala/pickling in its standalone form. It turns out that function closures are tricky to serialize due to the complicated enviornments that they can have. Chapter 5 focuses on this problem and introduces a new abstraction and type system designed to ensure that closures are always serializable.

3.2.2 Advanced Usage

@pickleable Annotation. To handle subtyping correctly, the pickling framework generates dispatch code which delegates to a pickler specialized for the runtime type of the object to-be-pickled, or, if the runtime type is unknown, which is to be expected in the presence of separate compilation, to a generic, but slower, runtime pickler.

For better performance, scala/pickling additionally provides an annotation which, at compile-time, inserts a runtime type test to check whether the runtime class extends a certain class/trait. In this case, a method that returns the pickler specialized for that runtime class is called. If the class/trait has been annotated, the returned pickler is guaranteed to have been generated statically. Furthermore, the `@pickleable` annotation (implemented as a macro annotation) is expanded transitively in each subclass of the annotated class/trait.

This `@pickleable` annotation enables:

- library authors to guarantee to their clients that picklers for separately-compiled subclasses are fully generated at compile-time;
- faster picklers in general because one need not worry about having to fallback on a runtime pickler.

For example, assume the following class `Person` and its subclass `Firefighter` are defined in separately-compiled code.

```
// Library code
@pickleable class Person(val name: String)

// Client code
class Firefighter(override val name: String, salary: Int)
  extends Person(name)
```

Note that class `Person` is annotated with the `@pickleable` annotation. `@pickleable` is a *macro annotation* which generates additional methods for obtaining type-specialized picklers (and unpicklers). With the `@pickleable` annotation expanded, the code for class `Person` looks roughly as follows:

```
class Person(val name: String)
    extends PickleableBase {
  def pickler: SPickler[_] =
    implicitly[SPickler[Person]]
  ...
}
```

First, note that the supertypes of `Person` now additionally include the trait `PickleableBase`; it declares the abstract methods that the expansion of the macro annotation "fills in" with concrete methods. In this case, a `pickler` method is generated which returns an `SPickler[_]`.[2] Note that the `@pickleable` annotation is defined in a way where pickler generation is triggered in both `Person` and its subclasses.

Here, we obtain an instance of `SPickler[Person]` by means of implicits. The `implicitly` method, part of Scala's standard library, is defined as follows:

```
def implicitly[T](implicit e: T) = e
```

Annotating the parameter (actually, the parameter list) using the `implicit` keyword means that in an invocation of `implicitly`, the implicit argument list may be omitted if, for each parameter of that list, there is exactly one value of the right type in the *implicit scope*. The implicit scope is an adaptation of the regular variable scope; imported implicits, or implicits declared in an enclosing scope are contained in the implicit scope of a method invocation.

As a result, `implicitly[T]` returns the uniquely-defined implicit value of type `T` which is in scope at the invocation site. In the context of picklers, there might not be an implicit value of type `SPickler[Person]` in scope (in fact, this is typically only the case with custom picklers). In that case, a suitable pickler instance is generated using a macro def.

Macro defs. Macro defs are methods that are transparently loaded by the compiler and executed (or expanded) during compilation. A macro is defined as if it is a normal method, but it is linked using the `macro` keyword to an additional method that operates on abstract syntax trees.

[2] The notation `SPickler[_]` is short for the existential type `SPickler[t] forSome { type t }`. It is necessary here, because picklers must be invariant in their type parameter, see Section 3.3.1.

```
def assert(x: Boolean, msg: String): Unit = macro assert_impl
def assert_impl(c: Context)
  (x: c.Expr[Boolean], msg: c.Expr[String]):
                            c.Expr[Unit] = ...
```

In the above example, the parameters of `assert_impl` are syntax trees, which the body of `assert_impl` operates on, itself returning an AST of type `Expr[Unit]`. It is `assert_impl` that is expanded and evaluated at compile-time. Its result is then inlined at the call site of `assert` and the inlined result is typechecked. It is also important to note that implicit defs as described above can be implemented as macros.

Scala/pickling provides an implicit macro def returning picklers for arbitrary types. Slightly simplified, it is declared as follows:

```
implicit def genPickler[T]: SPickler[T]
```

This macro def is expanded when invoking
`implicitly[SPickler[T]]` if there is no implicit value of type `SPickler[T]` in scope.

Custom Picklers. It is possible to use manually written picklers in place of generated picklers. Typical motivations for doing so are (a) improved performance through specialization and optimization hints, and (b) custom pre-pickling and post-unpickling actions; such actions may be required to re-initialize an object correctly after unpickling. Creating custom picklers is greatly facilitated by modular composition using object-oriented pickler combinators. The design of these first-class object-oriented picklers and pickler combinators is discussed in detail in the following Section 3.3.

3.3 Object-Oriented Picklers

In the first part of this section (3.3.1) we introduce picklers as first-class objects, and, using examples, motivate the contracts that valid implementations must guarantee. We demonstrate that the introduced picklers enable modular, object-oriented pickler combinators, *i.e.*, methods for composing more complex picklers from simpler primitive picklers.

In the second part of this section (3.3.2) we present a formalization of object-oriented picklers based on an operational semantics.

3.3.1 Picklers in Scala

In scala/pickling, a *static pickler* for some type `T` is an instance of trait `SPickler[T]` which has a single abstract method, `pickle`:

```
trait SPickler[T] {
  def pickle(obj: T, builder: PBuilder): Unit
}
```

For a concrete type, say, class `Person` from Section 3.2, the `pickle` method of an `SPickler[Person]` converts `Person` instances to a pickled format, using a *pickle builder* (the `builder` parameter). Given this definition, picklers "are type safe in the sense that a type-specialized pickler can be applied only to values of the specialized type" [Elsman, 2005]. The pickled result is not returned directly; instead, it can be requested from the `builder` using its `result()` method. Example:

```
val p = new Person("Jack")
...
val personPickler   = implicitly[SPickler[Person]]
val builder         = pickleFormat.createBuilder()
personPickler.pickle(p, builder)
val pickled: Pickle = builder.result()
```

In the above example, invoking `implicitly[SPickler[Person]]` either returns a regular implicit value of type `SPickler[Person]` that is in scope, or, if it doesn't exist, triggers the (compile-time) generation of a type-specialized pickler (see Section 3.4). To use the pickler, it is also necessary to obtain a pickle builder of type `PBuilder`. Since pickle formats in scala/pickling are exchangeable (see Section 3.4.3), the pickle builder is provided by the specific pickle format, through builder factory methods.

The `pickled` result has type `Pickle` which wraps a concrete representation, such as a byte array (*e.g.*, for binary formats) or a string (*e.g.*, for JSON). The abstract `Pickle` trait is defined as follows:

```
trait Pickle {
  type ValueType
  type PickleFormatType <: PickleFormat
  val value: ValueType
  ...
}
```

The type members `ValueType` and `PickleFormatType` abstract from the concrete representation type and the pickle format type, respectively. For example, scala/pickling defines a `Pickle` subclass for its default binary format as follows:

```
case class BinaryPickle(value: Array[Byte]) extends Pickle {
  type ValueType = Array[Byte]
```

```
  type PickleFormatType = BinaryPickleFormat
  override def toString = ...
}
```

Analogous to a pickler, an unpickler for some type `T` is an instance of trait `Unpickler[T]` that has a single abstract method `unpickle`; its (simplified) definition is as follows:

```
trait Unpickler[T] {
  def unpickle(reader: PReader): T
}
```

Similar to a pickler, an unpickler does not access pickled objects directly, but through the `PReader` interface, which is analogous to the `PBuilder` interface. A `PReader` is set up to read from a pickled object as follows. First, we need to obtain an instance of the pickle format that was used to produce the pickled object; this format is either known beforehand, or it can be selected using the `PickleFormatType` member of `Pickle`. The pickle format, in turn, has factory methods for creating concrete `PReader` instances:

```
val reader = pickleFormat.createReader(pickled)
```

The obtained reader can then be passed to the `unpickle` method of a suitable `Unpickler[T]`. Alternatively, a macro def on trait `Pickle` can be invoked directly for unpickling:

```
trait Pickle {
  ...
  def unpickle[T] = macro ...
}
```

It is very common for an instance of `SPickler[T]` to also mix in `Unpickler[T]`, thereby providing both pickling and unpickling capabilities.

Pickling and Subtyping

So far, we have introduced the trait `SPickler[T]` to represent picklers that can pickle objects of type `T`. However, in the presence of subtyping and open class hierarchies providing correct implementations of `SPickler[T]` is quite challenging. For example, how can an `SPickler[Person]` know how to pickle an arbitrary, unknown subclass of `Person`? Regardless of implementation challenges, picklers that handle arbitrary subclasses are likely less efficient than more specialized picklers.

To provide flexibility while enabling optimization opportunities, scala/pickling introduces two different traits for picklers: the introduced trait `SPickler[T]` is called a static pickler; it does not have to support pickling of subclasses of `T`. In addition, the trait `DPickler[T]` is called a dynamic pickler; its contract requires that it is applicable also to subtypes of `T`. The following section motivates the need for dynamic picklers, and shows how the introduced concepts enable a flexible, object-oriented form of pickler combinators.

Modular Pickler Combinators

This section explores the composition of the pickler abstractions introduced in the previous section by means of an example. Consider a simple class `Position` with a field of type `String` and a field of type `Person`, respectively:

```
class Position(val title: String, val person: Person)
```

To obtain a pickler for objects of type `Position`, ideally, existing picklers for type `String` and for type `Person` could be combined in some way. However, note that the `person` field of a given instance of class `Position` could point to an instance of a subclass of `Person` (assuming class `Person` is not final). Therefore, a modularly re-usable pickler for type `Person` must be able to pickle all possible subtypes of `Person`.

In this case, the contract of static picklers is too strict, it does not allow for subtyping. The contract of dynamic picklers on the other hand does allow for subtyping. As a result, *dynamic picklers are necessary so as to enable modular composition in the presence of subtyping.*

Picklers for final class types like `String`, or for primitive types like `Int` do not require support for subtyping. Therefore, static picklers are sufficient to pickle these *effectively final types*. Compared to dynamic picklers, static picklers benefit from several optimizations.

Implementing Object-Oriented Picklers

The main challenge when implementing OO picklers comes from the fact that a dynamic pickler for type `T` must be able to pickle objects of any subtype of `T`. Thus, the implementation of a dynamic pickler for type `T` must, in general, dynamically dispatch on the runtime type of the object to-be-pickled to take into account all possible subtypes of `T`. Because of this dynamic dispatch, manually constructing dynamic picklers can be difficult. It is therefore important for a framework for object-oriented picklers to provide good support for realizing this form of dynamic dispatching.

There are various ways across many different object-oriented programming languages to handle subtypes of the pickler's static type:

- Data structures with shallow class hierarchies, such as lists or trees, often have few

final leaf classes. As a result, manual dispatch code is typically simple in such cases. For example, a manual pickler for Scala's `List` class does not even have to consider subclasses.

- Java-style runtime reflection can be used to provide a generic `DPickler[Any]` which supports pickling objects of any type [Oracle, Inc., 2011, Philippsen et al., 2000]. Such a pickler can be used as a fallback to handle subtypes that are unknown to the pickling code; such subtypes must be handled in the presence of separate compilation. In Section 3.4.4 we present Scala implementations of such a generic pickler.

- Java-style annotation processing is commonly used to trigger the generation of additional methods in annotated class types. The purpose of generated methods for pickling would be to return a pickler or unpickler specialized for an annotated class type. In C#, the Roslyn Project [Ng et al., 2012] allows augmenting class definitions based on the presence of annotations.

- Static meta programming [Burmako and Odersky, 2012, Skalski, 2005] enables generation of picklers at compile time. In Section 3.4 we present an approach for generating object-oriented picklers from regular (class) type definitions.

Supporting Unanticipated Evolution

Given the fact that the type `SPickler[T]`, as introduced, has a type parameter `T`, it is reasonable to ask what the variance of `T` is. Ruling out covariance because of `T`'s occurrence in a contravariant position as the type of a method parameter, it remains to determine whether `T` can be contravariant.

For this, it is useful to consider the following scenario. Assume `T` is declared to be contravariant, as in `SPickler[-T]`. Furthermore, assume the existence of a public, non-final class `C` with a subclass `D`:

```
class C {...}
class D extends C {...}
```

Initially, we might define a generic pickler for `C`:

```
implicit val picklerC = new SPickler[C] {
  def pickle(obj: C): Pickle = { ... }
}
```

Because `SPickler[T]` is contravariant in its type parameter, instances of `D` would be pickled using `picklerC`. There are several possible extensions that might be *unanticipated* initially:

- Because the implementation details of class `D` change, instances of `D` should be pickled using a dedicated pickler instead of `picklerC`.
- A subclass `E` of `C` is added which requires a dedicated pickler, since `picklerC` does not know how to instantiate class `E` (since class `E` did not exist when `picklerC` was written).

In both cases it is necessary to add a new, dedicated pickler for either an existing subclass (`D`) or a new subclass (`E`) of `C`:

```
implicit val picklerD = new SPickler[D] { ... }
```

However, when pickling an instance of class `D` this new pickler, `picklerD`, would not get selected, even if the type of the object to-be-pickled is statically known to be `D`. The reason is that `SPickler[C]` <: `SPickler[D]` because of contravariance which means that `picklerC` is more specific than `picklerD`. As a result, according to Scala's implicit look-up rules `picklerC` is selected when an implicit object of type `SPickler[D]` is required. (Note that this is the case even if `picklerD` is declared in a scope that has higher precedence than the scope in which `picklerC` is declared.)

While contravariant picklers do not support the two scenarios for unanticipated extension outlined above, invariant picklers do, in combination with type bounds. Assuming invariant picklers, we can define a generic method `picklerC1` that returns picklers for all subtypes of class `C`:

```
implicit def picklerC1[T <: C] = new SPickler[T] {
  def pickle(obj: T): Pickle = { ... }
}
```

With this pickler in scope, it is still possible to define a more specific `SPickler[D]` (or `SPickler[E]`) as required:

```
implicit val picklerD1 = new SPickler[D] { ... }
```

$P ::= \overline{cdef}\ t$		program
$cdef ::= \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{fld}\ \overline{meth}\}$		class
$fld ::= \texttt{var}\ f : C$		field
$meth ::= \texttt{def}\ m(\overline{x : C}) : D = e$		method
$t ::= \texttt{let}\ x = e\ \texttt{in}\ t$		let binding
	$\mid x.f := y$	assignment
	$\mid x$	variable
$e ::= \texttt{new}\ C(\overline{x})$		instance creation
	$\mid x.f$	selection
	$\mid x.m(\overline{y})$	invocation
	$\mid t$	term

Figure 3.1 – Core language syntax. C, D are class names, f, m are field and method names, and x, y are names of variables and parameters, respectively.

$H ::=$	$\emptyset \mid (H, r \mapsto v)$	heap
$V ::=$	$\emptyset \mid (V, y \mapsto r)$	environment $(y \notin dom(V))$
$v ::=$	$o \mid \rho$	value
$o ::=$	$C(\overline{r})$	object
$\rho ::=$	(C_p, m, C)	pickler
$r \in$	$RefLocs$	reference location

Figure 3.2 – Heaps, environments, objects, and picklers.

However, the crucial difference is that now `picklerD1` is selected when an object of static type `D` is pickled, since `picklerD1` is more specific than `picklerC1`.

In summary, the combination of invariant picklers and generics (with upper type bounds) is flexible enough to support some important scenarios of unanticipated evolution. This is not possible with picklers that are contravariant. Consequently, in scala/pickling the `SPickler` trait is invariant in its type parameter.

3.3.2 Formalization

To define picklers formally we use a standard approach based on an operational semantics for a core object-oriented language. Importantly, our goal is not a full formalization of a core language; instead, we (only) aim to provide a precise definition of *object-oriented picklers*. Thus, our core language simplifies our actual implementation language in several ways. Since our basic definitions are orthogonal to the type system of the host language, we limit types to non-generic classes with at most one superclass. Moreover, the core language does not have first-class functions, or features like pattern matching. The core language without picklers is a simplified version of a core language used in the formal development of a uniqueness type system for Scala [Haller and Odersky, 2010].

Figure 3.1 shows the core language syntax. A program is a sequence of class definitions

followed by a (main) term. (We use the common over-bar notation [Igarashi et al., 2001] for sequences.) Without loss of generality, we use a form where all intermediate terms are named (A-normal form [Flanagan et al., 1993]). The language does not support arbitrary mutable variables (*cf.* [Pierce, 2002], Chapter 13); instead, only fields of objects can be (re-)assigned.

We assume the existence of two pre-defined class types, `AnyRef` and `Pickle`. All class hierarchies have `AnyRef` as their root. For the purpose of our core language, `AnyRef` is simply a member-less class without a superclass. `Pickle` is the class type of objects that are the result of pickling a regular object.

We define the standard auxiliary functions $mtype$ and $mbody$ as follows. Let $\texttt{def}\ m(\overline{x : C}) : D = e$ be a method defined in the most direct superclass of C that defines m. Then $mbody(m, C) = (\overline{x}, e)$ and $mtype(m, C) = \overline{C} \to D$.

Dynamic semantics

$$\frac{\begin{array}{c} V(x) = r_p \qquad H(r_p) = (C_p, s, C) \\ V(y) = r \qquad H(r) = C(\_) \\ mbody(p, C_p) = (z, e) \end{array}}{\begin{array}{c} H, V, \texttt{let}\ x' = x.p(y)\ \texttt{in}\ t \\ \longrightarrow H, (V, z \mapsto r), \texttt{let}\ x' = e\ \texttt{in}\ t \end{array}}\ \text{(R-PICKLE-S)}$$

$$\frac{\begin{array}{c} V(x) = r_p \qquad H(r_p) = (C_p, d, C) \\ V(y) = r \qquad H(r) = D(\_) \qquad D <: C \\ mbody(p, C_p) = (z, e) \end{array}}{\begin{array}{c} H, V, \texttt{let}\ x' = x.p(y)\ \texttt{in}\ t \\ \longrightarrow H, (V, z \mapsto r), \texttt{let}\ x' = e\ \texttt{in}\ t \end{array}}\ \text{(R-PICKLE-D)}$$

$$\frac{\begin{array}{c} V(x) = r \qquad H(r) = C(\_) \\ V(\overline{y}) = r_1 \ldots r_n \\ mbody(m, C) = (\overline{x}, e) \end{array}}{\begin{array}{c} H, V, \texttt{let}\ x' = x.m(\overline{y})\ \texttt{in}\ t \\ \longrightarrow H, (V, \overline{x \mapsto r}), \texttt{let}\ x' = e\ \texttt{in}\ t \end{array}}\ \text{(R-INVOKE)}$$

Figure 3.3 – Reduction rules for pickling.

We use a small-step operational semantics to formalize the dynamic semantics of our core language. Reduction rules are written in the form $H, V, t \longrightarrow H', V', t'$. That is, terms t are reduced in the context of a heap H and a variable environment V. Figure 3.2 shows their syntax. A heap maps reference locations to values. In our core language, values can be either objects or picklers. An object $C(\overline{r})$ stores location r_i in its i-th field. An environment maps variables to reference locations r. Note that we do not model explicit stack frames. Instead, method invocations are "flattened" by renaming the method parameters before binding them to their argument values in the environment (as in LJ [Strnisa et al., 2007]).

A pickler is a tuple (C_p, m, C) where C_p is a class that defines two methods p and u for pickling and unpickling an object of type C, respectively, where $mtype(p, C_p) = C \to \texttt{Pickle}$ and $mtype(u, C_p) = \texttt{Pickle} \to C$. The second component $m \in \{s, d\}$ is the pickler's *mode*; the operational semantics below explains how the mode affects the applicability of a pickler in the presence of subtyping.

As defined, picklers are first-class, since they are values just like objects. However, while picklers are regular objects in our practical implementation, picklers are different from objects in the present formal model. The reason is that a pickler has to contain a type tag indicating the types of objects that it can pickle (this is apparent in the rules of the operational semantics below); however, the alternative of adding parameterized types (as in, *e.g.*, FGJ [Igarashi et al., 2001]) is beyond the scope of this work.

According to the grammar in Figure 3.1, expressions are always reduced in the context of a let-binding, except for field assignments. Each operand of an expression is a variable y that the environment maps to a reference location r. Since the environment is a flat list of variable bindings, let-bound variables must be alpha-renamable: $\texttt{let}\ x = e\ \texttt{in}\ t \equiv \texttt{let}\ x' = e\ \texttt{in}\ [x'/x]t$ where $x' \notin FV(t)$. (We omit the definition of the FV function to obtain the free variables of a term, as it is standard [Pierce, 2002].)

In the following we explain the subset of the reduction rules suitable to formalize the properties of picklers. We start with the reduction rule for method invocations, since the reduction rules pertinent to picklers are variants of that rule.

Figure 3.3 shows the reduction rules for pickling and unpickling an object.

Rule (R-PICKLE-S) is a refinement of rule (R-INVOKE) for method invocations. When using a pickler x to pickle an object y such that the pickler's mode is s (static), the type tag C of the pickler indicating the type of objects that it can pickle must be equal to the dynamic class type of the object to-be-pickled (the object at location r). This expresses the fact that a static pickler can only be applied to objects of a precise statically-known type C, but not a subtype thereof.

In contrast, rule (R-PICKLE-D) shows the invocation of the pickling method p for a pickler with mode d (dynamic). In this case, the type tag C of the pickler must not be exactly equal to the dynamic type of the object to-be-pickled (the object at location r); it is only necessary that $D <: C$.

Property. The pickling and unpickling methods of a pickler must satisfy the property that "pickling followed by unpickling generates an object that is structurally equal to the original object". The following definition captures this formally:

Definition 3.3.1. Given variables x, x', y, y', heaps H, H', variable environments V, V', and a term t such that

$$
\begin{array}{ll}
V(y) = r & H(r) = C(\overline{r}) \\
V(x) = r_p & H(r_p) = (C_p, m, D) \\
\begin{cases} D = C & \text{if } m = s \\ D <: C & \text{if } m = d \end{cases} & \\
V'(y') = r' &
\end{array}
$$

and

$$
\begin{array}{l}
H, V, \texttt{let } x' = x.u(x.p(y)) \texttt{ in } t \\
\longrightarrow^* \; H', V', \texttt{let } x' = y' \texttt{ in } t
\end{array}
$$

Then r and r' must be structurally equivalent in heap H', written $r \equiv_{H'} r'$.

Note that in the above definition we assume that references in heap H are not garbage collected in heap H'. The definition of structural equivalence is straight-forward.

Definition 3.3.2. (Structural Equivalence)

Two picklers r_p, r'_p are structurally equal in heap H, written $r_p \equiv_H r'_p$ *iff*

$$
\begin{array}{r}
H(r_p) = (C_p, m, C) \wedge H(r'_p) = (C'_p, m', C') \Rightarrow \\
m = m' \wedge C <: C' \wedge C' <: C
\end{array}
\tag{3.1}
$$

Two reference locations r, r' are structurally equal in heap H, written $r \equiv_H r'$ *iff*

$$
\begin{array}{r}
H(r) = C(\overline{r}) \wedge H(r') = C'(\overline{p}) \Rightarrow \\
C <: C' \wedge C' <: C \wedge \forall r_i \in \overline{r}, p_i \in \overline{p}.\; r_i \equiv_H p_i
\end{array}
\tag{3.2}
$$

Note that the above definition considers two picklers to be structurally equal even if their implementation classes C_p and C'_p are different. In some sense, this is consistent with our practical implementation in the common case where picklers are only resolved using implicits: Scala's implicit resolution enforces that an implicit pickler of a given type is uniquely determined.

3.3.3 Summary

This section has introduced an object-oriented model of first-class picklers. Object-oriented picklers enable modular pickler combinators with support for subtyping, thereby extending a well-known approach in functional programming. The distinction between static and dynamic picklers enables optimizations for final class types and primitive types. Object-oriented pick-

lers can be implemented using various techniques, such as manually written picklers, runtime reflection, or Java-style annotation processors. We argue that object- oriented picklers should be invariant in their generic type parameter to allow for several scenarios of unanticipated evolution. Finally, we provide a formalization of a simple form of OO picklers.

3.4 Generating Object-Oriented Picklers

An explicit goal of our framework is to require little to no boilerplate in client code, since practitioners are typically accustomed to serialization supported by the underlying runtime environment like in Java or .NET. Therefore, instead of requiring libraries or applications to supply manually written picklers for all pickled types, our framework provides a component for *generating picklers* based on their required static type.

Importantly, compile-time pickler generation enables *efficient picklers* by generating as much pickling code as possible statically (which corresponds to a partial evaluation of pickler combinators). Section 5.5 reports on the performance improvements that our framework achieves using compile-time pickler generation, compared to picklers based on runtime reflection, as well as manually written picklers.

3.4.1 Overview

Our framework generates type-specialized, object-oriented picklers using compile-time meta programming in the form of *macros*. Whenever a pickler for static type T is required but cannot be found in the implicit scope, a macro is expanded which generates the required pickler step-by-step by:

- Obtaining a type descriptor for the static type of the object to-be-pickled,
- Building a static *intermediate representation* of the object-to-be-pickled, based on the type descriptor, and
- Applying a pickler generation algorithm, driven by the static pickler representation.

In our Scala-based implementation, the static type descriptor is generated automatically by the compiler, and passed as an implicit argument to the pickle extension method (see Section 3.2). As a result, such an implicit `TypeTag`[1] does not require changing the invocation in most cases. (However, it is impossible to generate a `TypeTag` automatically if the type or one of its components is abstract; in this case, an implicit `TypeTag` must be in scope.)

Based on the type descriptor, a static representation, or model, of the required pickler is built; we refer to this as the *Intermediate Representation* (IR). The IR specifies precisely the set of types for which our framework can generate picklers automatically. Furthermore, these IRs are composable.

We additionally define a model for composing IRs, which is designed to capture the essence of Scala's object system as it relates to pickling. The model defines how the IR for a given type is composed from the IRs of the picklers of its supertypes. In Scala, the composition of an IR for a class type is defined based on the linearization of its supertraits.[2] This model of inheritance is central to the generation framework, and is formally defined in the following Section 3.4.2

3.4.2 Model of Inheritance

The goal of this section is to define the IR, which we'll denote Υ, of a static type T as it is used to generate a pickler for type T. We start by defining the syntax of the elements of the IR (see Def. 3.4.1).

Definition 3.4.1. (Elements of IR)

We define the syntax of values of the IR types.

$$
\begin{aligned}
F &::= \overline{(f_n, T)} \\
\Upsilon &::= (T, \Upsilon_{opt}, F) \\
\Upsilon_{opt} &::= \epsilon \mid \Upsilon
\end{aligned}
$$

F represents a sequence of *fields*. We write $\overline{X}$ as shorthand for sequences, $X_1, \ldots, X_n$, and we write tuples $(X_1, \ldots, X_n)$. f_n is a string representing the name of the given field, and T is its type.

Υ represents the pickling information for a class or some other object type. That is, an Υ for type T contains all of the information required to pickle instances of type T, including all necessary static info for pickling its fields provided by F.

Υ_{opt} is an optional Υ; a missing Υ is represented using ϵ.

In our implementation the IR types are represented using case classes. For example, the following case class represents Υs:

```
case class ClassIR(
  tpe: Type,
  parent: ClassIR,
  fields: List[FieldIR]
) extends PickleIR
```

[1] TypeTags are part of the mainline Scala compiler since version 2.10. They replace the earlier concept of Manifests, providing a faithful representation of Scala types at runtime.

[2] Traits in Scala can be thought of as a more flexible form of Java-style interfaces that allow concrete members, and that support a form of multiple inheritance (mix-in composition) that is guaranteed to be safe based on a linearization order.

We go on to define a number of useful IR combinators, which form the basis of our model of inheritance.

Definition 3.4.2. (IR Combinators - Type Definitions)

We begin by defining the types of our combinators before we define the combinators themselves.

Type Definitions

$$
\begin{aligned}
\mathit{concat} &: (F, F) \Rightarrow F \\
\mathit{extended} &: (\Upsilon, \Upsilon) \Rightarrow \Upsilon \\
\mathit{linearization} &: T \Rightarrow \overline{T} \\
\mathit{superIRs} &: T \Rightarrow \overline{\Upsilon} \\
\mathit{compose} &: \Upsilon \Rightarrow \Upsilon \\
\mathit{flatten} &: \Upsilon \Rightarrow \Upsilon
\end{aligned}
$$

We write function types $X \Rightarrow Y$, indicating a function from type X to type Y.

The *linearization* function represents the host language's semantics for the linearized chain of supertypes.[3]

Definition 3.4.3. (IR Combinators - Function Defns)

Function Definitions

$$
\begin{aligned}
\mathit{concat}(\overline{f}, \overline{g}) &= \overline{f}, \overline{g} \\
\mathit{extended}(C, D) &= (T, C, \mathit{fields}(T)) \\
&\qquad \text{where } D = (T, \_, \_) \ \wedge T <: C.1 \\
\mathit{superIRs}(T) &= [(S, \epsilon, \mathit{fields}(S)) \mid S \in \mathit{linearization}(T)] \\
\mathit{compose}(C) &= \mathit{reduce}(\mathit{superIRs}(C.1), \mathit{extended}) \\
\mathit{flatten}(C) &= \begin{cases} (C.1, C.2, \mathit{concat}(C.3, \mathit{flatten(C.2).3})), \\ \qquad \text{if } C.2 \neq \epsilon \\ C, \quad \text{otherwise} \end{cases}
\end{aligned}
$$

The function *concat* takes two sequences as arguments. We denote concatenation of sequences using a comma. We introduce the *concat* function for clarity in the definition

[3] For example, in Scala the linearization is defined for classes mixing in multiple traits [Odersky, 2013, Odersky and Zenger, 2005]; in Java, the linearization function would simply return the chain of superclasses, not including the implemented interfaces.

of *flatten* (see below); it is simply an alias for sequence concatenation.

The function *extended* takes two ϒs, C and D, and returns a new ϒ for the type of D such that C is registered as its super ϒ. Basically, *extended* is used to combine a completed ϒ C with an incomplete ϒ D yielding a completed ϒ for the same type as D. When combining the ϒs of a type's supertypes, the *extended* function is used for reducing the linearization sequence yielding a single completed ϒ.

The function *superIRs* takes a type T and returns a sequence of the IRs of T's supertypes in linearization order.

The function *compose* takes an ϒ C for a type $C.1$ and returns a new ϒ for type $C.1$ which is the composition of the IRs of all supertypes of $C.1$. The resulting ϒ is a chain of super IRs according to the linearization order of $C.1$.

The function *flatten*, given an ϒ C produces a new ϒ that contains a concatenation of all the fields of each nested ϒ. Given these combinators, the ϒ of a type T to-be-pickled is obtained using $\Upsilon = flatten(compose((T, \epsilon, [])))$.

The above IR combinators have direct Scala implementations in scala/pickling. For example, function $superIRs$ is implemented as follows:

```
private val f3 = (c: C) =>
  c.tpe.baseClasses
       .map(superSym => c.tpe.baseType(superSym))
       .map(tp => ClassIR(tp, null, fields(tp)))
```

Here, method `baseClasses` returns the collection of superclass symbols of type `c.tpe` in linearization order. Method `baseType` converts each symbol to a type which is, in turn, used to create a `ClassIR` instance. The semantics of the `fields` method is analogous to the above $fields$ function.

3.4.3 Pickler Generation Algorithm

The pickler generation is driven by the IR (see Section 3.4.2) of a type to-be-pickled. We describe the generation algorithm in two steps. In the first step, we explain how to generate a pickler for static type T assuming that for the dynamic type S of the object to-be-pickled, *erasure*$(T) =:= S$. In the second step, we explain how to extend the generation to dynamic picklers which do not require this assumption.

Pickle Format

The pickling logic that we are going to generate contains calls to a pickle *builder* that is used to incrementally construct a pickle. Analogously, the unpickling logic contains calls to a pickle *reader* that is used to incrementally read a pickle. Importantly, the pickle format that determines the precise persisted representation of a completed pickle is not fixed. Instead, the pickle format to be used is selected at compile time– efficient binary formats, and JSON are just some examples. This selection is done via implicit parameters which allows the format to be flexibly selected while providing a default binary format which is used in case no other format is imported explicitly.

The pickle format provides an interface which plays the role of a simple, lower-level "backend". Besides a pickle template that is generated inline as part of the pickling logic, methods provided by pickle builders aim to do as little as possible to minimize runtime overhead. For example, the JSON `PickleFormat` included with scala/pickling simply uses an efficient string builder to concatenate JSON fragments (which are just strings) in order to assemble a pickle.

The interface provided by `PickleFormat` is simple: it basically consists of two methods (a) for creating an empty builder, and (b) for creating a reader from a pickle:[3]

```
def createBuilder(): PBuilder
def createReader(pickle: PickleType): PReader
```

The `createReader` method takes a pickle of a specific `PickleType` (which is an abstract type member in our implementation); this makes it possible to ensure that, say, a pickle encapsulating a byte array is not erroneously attempted to be unpickled using the JSON pickle format. Moreover, pickle builders returned from `createBuilder` are guaranteed to produce pickles of the right type.

```
class PBuilder {
  def beginEntry(obj: Any): PBuilder
  def putField(n: String, pfun: PBuilder => Unit): PBuilder
  def endEntry(): Unit
  def result(): Pickle
}
```

In the following we're going to show how the `PBuilder` interface is used by generated picklers; the `PReader` interface is used by generated unpicklers in an analogous way. The above example summarizes a core subset of the interface of `PBuilder` that the presented generation algorithm

[3] In our actual implementation the `createReader` method takes an additional parameter which is a "mirror" used for runtime reflection; it is omitted here for simplicity.

is going to use.[4] The `beginEntry` method is used to indicate the start of a pickle for the argument `obj`. The field values of a class instance are pickled using `putField` which expects both a field name and a lambda encapsulating the pickling logic for the object that the field points to. The `endEntry` method indicates the completion of a (partial) pickle of an object. Finally, invoking `result` returns the completed `Pickle` instance.

Tree Generation

The objective of the generation algorithm is to generate the body of `SPickler`'s `pickle` method:

```
def pickle(obj: T, builder: PBuilder): Unit = ...
```

As mentioned previously, the actual pickling logic is synthesized based on the IR. Importantly, the IR determines which fields are pickled and how. A lot of the work is already done when building the IR; therefore, the actual tree generation is rather simple:

- Emit `builder.beginEntry(obj)`.
- For each field `fld` in the IR, emit `builder.putField(${fld.name},b => pbody)` where `${fld.name}` denotes the splicing of `fld.name` into the tree. `pbody` is the logic for pickling `fld`'s value into the builder `b`, which is an alias of `builder`. `pbody` is generated as follows:
  1. Emit the field getter logic: `val v: ${fld.tpe} = obj.${fld.name}`. The expression `${fld.tpe}` splices the type of `fld` into the generated tree; `${fld.name}` splices the name of `fld` into the tree.
  2. Recursively generate the pickler for `fld`'s type by emitting either `val fldp = implicitly[DPickler[${fld.tpe}]]` or `val fldp = implicitly[SPickler[${fld.tpe}]]`, depending on whether `fld`'s type is effectively final or not.
  3. Emit the logic for pickling `v` into `b`: `fldp.pickle(v, b)`

A practical implementation can easily be refined to support various extensions of this basic model. For example, support for avoiding pickling fields marked as *transient* is easy with this model of generation– such fields can simply be left out of the IR. Or, based on the static types of the picklee and its fields, we can emit hints to the builder to enable various optimizations.

[4] It is not necessary that `PBuilder` is a class. In fact, in our Scala implementation it is a trait. In Java, it could be an interface.

For example, a field whose type T is *effectively final, i.e.,* it cannot be extended, can be optimized as follows:

- Instead of obtaining an implicit pickler of type `DPickler[T]`, it is sufficient to obtain an implicit pickler of type `SPickler[T]`, which is more efficient, since it does not require a dynamic dispatch step like `DPickler[T]`
- The field's type does not have to be pickled, since it can be reconstructed from its owner's type.

Pickler generation is compositional; for example, the generated pickler for a class type with a field of type `String` re-uses the `String` pickler. This is achieved by generating picklers for parts of an object type using invocations of the form `implicitly[DPickler[T]]`. This means that if there is already an implicit value of type `DPickler[T]` in scope, it is used for pickling the corresponding value. Since the lookup and binding of these implicit picklers is left to a mechanism outside of pickler generation, what's actually generated is a *pickler combinator* which returns a *pickler* composed of *existing picklers* for parts of the object to-be-pickled. More precisely, pickler generation provides the following composability property:

Property 3.4.1. (Composability) A generated pickler `p` is composed of implicit picklers of the required types that are in scope at the point in the program where `p` is generated.

Since the picklers that are in scope at the point where a pickler is generated are under programmer control, it is possible to import manually written picklers which are transparently picked up by the generated pickler. Our approach thus has the attractive property that it is an "open-world" approach, in which it is easy to add new custom picklers for selected types at exactly the desired places while integrating cleanly with generated picklers.

Dispatch Generation

So far, we have explained the generation of the pickling logic of static picklers. Dynamic picklers require an additional dispatch step to make sure subtypes of the static type to-be-pickled are pickled properly. The generation of a `DPickler[T]` is triggered by invoking `implicitly[DPickler[T]]` which tries to find an implicit of type `DPickler[T]` in the current implicit scope. Either there is already an implicit value of the right type in scope, or the only matching implicit is an implicit def provided by the pickling framework which generates a `DPickler[T]` on-the-fly. The generated dispatch logic has the following shape:

```
val clazz = if (picklee != null) picklee.getClass else null
val pickler = clazz match {
  case null => implicitly[SPickler[NullTpe]]
```

```
  case c1 if c1 == classOf[S1] => implicitly[SPickler[S1]]
  ...
  case cn if cn == classOf[Sn] => implicitly[SPickler[Sn]]
  case _ => genPickler(clazz)
}
```

The types $S_1, \ldots, S_n$ are known subtypes of the picklee's type T. If T is a sealed class or trait with final subclasses, this set of types is always known at compile time. However, in the presence of separate compilation it is, generally, possible that a picklee has an unknown runtime type; therefore, we include a default case (the last case in the pattern match) which dispatches to a runtime pickler that inspects the picklee using (runtime) reflection.

If the static type T to be pickled is annotated using the `@pickleable` annotation, all subclasses are guaranteed to extend the predefined `PickleableBase` interface trait. Consequently, a more optimal dispatch can be generated in this case:

```
val pickler =
  if (picklee != null) {
    val pbase = picklee.asInstanceOf[PickleableBase]
    pbase.pickler.asInstanceOf[SPickler[T]]
  }
  else implicitly[SPickler[NullTpe]]
```

3.4.4 Runtime Picklers

One goal of our framework is to generate as much pickling code at compile time as possible. However, due to the interplay of subclassing with both separate compilation and generics, we provide a runtime fall back capability to handle the cases that cannot be resolved at compile time.

Subclassing and separate compilation A situation arises where it's impossible to statically know all possible subclasses. In this case there are three options: (1) provide a custom pickler, and (2) use an annotation which is described in Section 3.2.2. In the case where neither a custom pickler nor an annotation is provided, our framework can inspect the instance to-be-pickled at runtime to obtain the pickling logic. This comes with some runtime overhead, but in Section 5.5 we present results which suggest that this overhead is not necessary in many cases.

For the generation of runtime picklers our framework supports two possible strategies:

- Runtime interpretation of a type-specialized pickler

- Runtime compilation of a type-specialized pickler

Interpreted runtime picklers. If the runtime type of an object is unknown at compile time, *e.g.*, if its static type is `Any`, it is necessary to carry out the pickling based on inspecting the type of the object to-be-pickled at runtime. We call picklers operating in this mode "interpreted runtime picklers" to emphasize the fact that the pickling code is not partially evaluated in this case. An interpreted pickler is created based on the runtime class of the picklee. From that runtime class, it is possible to obtain a runtime type descriptor:

- to build a static intermediate representation of the type (which describes all its fields with their types, etc.)
- to determine in which way the picklee should be pickled (as a primitive or not).

In case the picklee is of a primitive type, there are no fields to be pickled. Otherwise, the value and runtime type of each field is obtained, so that it can be written to the pickle.

3.4.5 Generics and Arrays

Subclassing and generics. The combination of subclassing and generics poses a similar problem to that introduced above in Section 3.4.4. For example, consider a generic class `C`,

```
class C[T](val fld: T) { ... }
```

A `Pickler[C[T]]` will not be able to pickle the field `fld` if its static type is unknown. To support pickling instances of generic classes, our framework falls back to using runtime picklers for pickling fields of generic type. So, when we have access to the runtime type of field `fld`, we can either look up an already-generated pickler for that runtime type, or we can generate a suitable pickler dynamically.

Arrays. Scala arrays are mapped to Java arrays; the two have the same runtime representation. However, there is one important difference: Java arrays are covariant whereas Scala arrays are invariant. In particular, it is possible to pass arrays from Java code to Scala code. Thus, a class `C` with a field `f` of type `Array[T]` may have an instance at runtime that stores an `Array[S]` in field `f` where `S` is a subtype of `T`. Pickling followed by unpickling must instantiate an `Array[S]`. Just like with other fields of non-final reference type, this situation requires writing the dynamic (array) type name to the pickle. This is possible, since array types are not erased on the JVM (unlike generic types). This allows instantiating an array with the expected dynamic type upon unpickling.

3.4.6 Object Identity and Sharing

Object identity enables the existence of complex object graphs, which themselves are a cornerstone of object-oriented programming. While in Section 3.6.7 we show that pickling *flat* object graphs is most common in big data applications, a general pickling framework for use with an object-oriented language must not only support flat object graphs, it must also support cyclic object graphs.

Supporting such cyclic object graphs in most object-oriented languages, however, typically requires sophisticated runtime support, which is known to incur a significant performance hit. This is due to the fact that pickling graphs with cycles requires tracking object identities at runtime, so that pickling terminates and unpickling can faithfully reconstruct the graph structure.

To avoid the overhead of tracking object identities unanimously for all objects, "runtime-based" serialization frameworks like Java or Kryo have to employ reflective/introspective checks to detect whether identities are relevant.[5]

Scala/pickling, on the other hand, employs a hybrid compile-time/runtime approach. This makes it possible to avoid the overhead of object identity tracking in cases where it is statically known to be safe, which we show in Section 3.6.7 is typically common in big data applications.

The following Section 3.4.6 outlines how object identity is tracked in scala/pickling. It also explains how the management of object identities enables a *sharing* optimization. This sharing optimization is especially important for persistent data structures, which are commonly used in Scala. Section 3.4.6 explains how compile-time analysis is used to reduce the amount of runtime checking in cases where object graphs are statically known to be acyclic.

Object Tracking

During pickling, a pickler keeps track of all objects that are part of the (top-level) object to-be-pickled in a table. Whenever an object that's part of the object graph is pickled, a hash code based on the identity of the object is computed. The pickler then looks up whether that object has already been pickled, in which case the table contains a unique integer ID as the entry's value. If the table does not contain an entry for the object, a unique ID is generated and inserted, and the object is pickled as usual. Otherwise, instead of pickling the object again, a special `Ref` object containing the integer ID is written to the pickle.[6] During unpickling, the above process is reversed by maintaining a mapping[7] from integer IDs to unpickled heap objects.

[5] With Kryo, some of this overhead can be avoided when using custom, handwritten serializers.

[6] Several strategies exist to avoid preventing pickled objects from being garbage collected. Currently, for each top-level object to-be-pickled, a new hash table is created.

[7] This can be made very efficient by using a map implementation which is more efficient for integer-valued keys, such as a resizable array.

This approach to dealing with object identities also enables sharing, an optimization which in some big data applications can improve system throughput by reducing pickle size. Scala's immutable collections hierarchy is one example of a set of data structures which are persistent, which means they make use of sharing. That is, object subgraphs which occur in multiple instances of a data structure can be shared which is more efficient than maintaining multiple copies of those subgraphs.

Scala/pickling's management of object identities benefits instances of such data structures as follows. First, it reduces the size of the computed pickle, since instead of pickling the same object instance many times, compact references (`Ref` objects) are pickled. Second, pickling time also has the potential to be reduced, since shared objects have to be pickled only once.

Static Object Graph Analysis

When generating a pickler for a given type `T`, the IR is analyzed to determine whether the graph of objects of type `T` may contain cycles. Both `T` and the types of `T`'s fields are examined using a breadth-first traversal. Certain types are immediately excluded from the traversal, since they cannot be part of a cycle. Examples are primitive types, like `Double`, as well as certain immutable reference types that are final, like `String`. However, the static inspection of the IR additionally allows scala/pickling to traverse sealed class hierarchies.

For example, consider this small class hierarchy:

```
final class Position(p: Person, title: String)
sealed class Person(name: String, age: Int)
final class Firefighter(name: String, age: Int, salary: Int)
  extends Person(name, age)
final class Teacher(name: String, age: Int, subject: String)
  extends Person(name, age)
```

In this case, upon generating the pickler for class `Position`, it is detected that no cycles are possible in the object graphs of instances of type `Position`. While `Position`'s `p` field has a reference type, it cannot induce cycles, since `Person` is a sealed class that has only final subclasses; furthermore, `Person` and its subclasses have only fields of primitive type.

In addition to this analysis, our framework allows users to disable all identity tracking programmatically (by importing an implicit value), in case it is known that the graphs of (all) pickled objects are acyclic. While this switch can boost performance, it also disables opportunities for sharing (see above), and may thus lead to larger "pickles".

3.5 Implementation

The presented framework has been fully implemented in Scala. The object-oriented pickler combinators presented in Section 3.3, including their implicit selection and composition, can be implemented using stable versions of the standard, open-source Scala distribution. The extension of our basic model with automatic pickler generation has been implemented using the experimental macros feature introduced in Scala 2.10.0. Macros can be thought of as a more regularly structured, localized, and more stable alternative to compiler plugins. To simplify tree generation, our implementation leverages a quasiquoting library for Scala's macros [Shabalin et al., 2013].

3.6 Experimental Evaluation

In this section we present first results of an experimental evaluation of our pickling framework. Our goals are

1. to evaluate the performance of automatically-generated picklers, analyzing the memory usage compared to other serialization frameworks, and
2. to provide a survey of the properties of data types that are commonly used in distributed computing frameworks and applications.

In the process, we are going to evaluate the performance of our framework alongside two popular and industrially-prominent serialization frameworks for the JVM, Java's native serialization, and Kryo.[8]

3.6.1 Experimental Setup

The following benchmarks were run on a MacBook Pro with a 2.6 GHz Intel Core i7 processor with 16 GB of memory running Mac OS X version 10.8.4 and Oracle's Java HotSpot(TM) 64-Bit Server VM version 1.6.0_51. In all cases we used the following configuration flags: `-XX:MaxPermSize=512m -XX:+CMSClassUnloadingEnabled -XX:ReservedCodeCacheSize=192m -XX:+UseConcMarkSweepGC -Xms512m -Xmx2g`. Each benchmark was run on a warmed-up JVM. The result shown is the median of 9 such "warm" runs.

3.6.2 Microbenchmark: Collections

In the first microbenchmark, we evaluate the performance of our framework when pickling standard collection types. We compare against three other serialization frameworks: Java's

[8] We select Kryo and Java because, like scala/pickling, they both are "automatic". That is, they require no schema or extra compilation phases, as is the case for other frameworks such as Apache Avro and Google's Protocol Buffers.

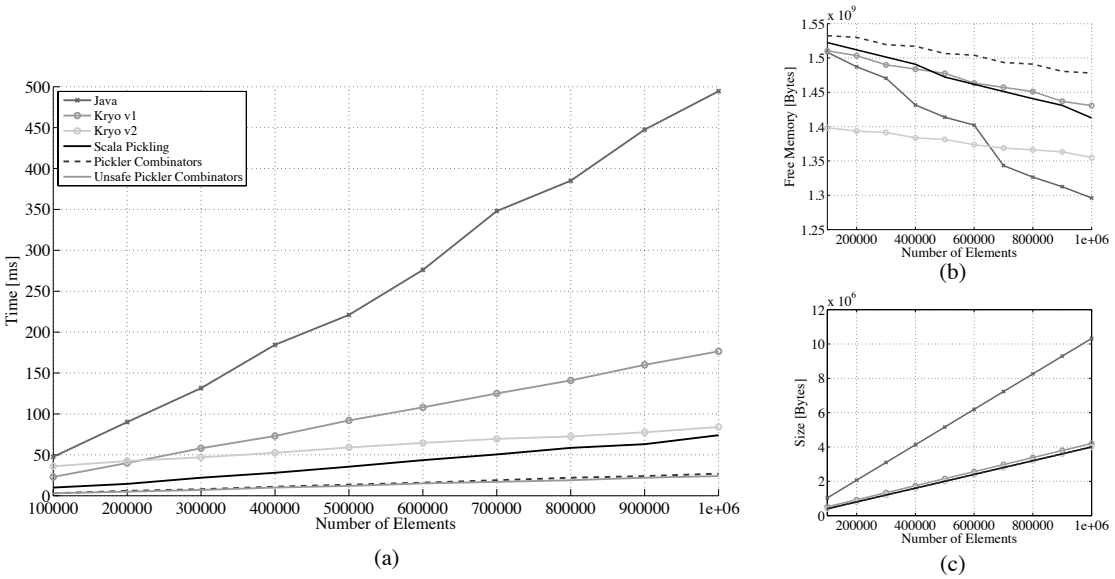


Figure 3.4 – Results for pickling and unpickling an immutable `Vector[Int]` using different frameworks. Figure 3.4(a) shows the roundtrip pickle/unpickle time as the size of the `Vector` varies. Figure 3.4(b) shows the amount of free memory available during pickling/unpickling as the size of the `Vector` varies. Figure 3.4(c) shows the pickled size of `Vector`.

native serialization, Kryo, and a combinator library of naive handwritten pickler combinators. All benchmarks are compiled and run using a current milestone of Scala version 2.10.3.

The benchmark logic is very simple: an immutable collection of type `Vector[Int]` is created which is first pickled (or serialized) to a byte array, and then unpickled. While `List` is the prototypical collection type used in Scala, we ultimately chose `Vector` as Scala's standard `List` type could not be serialized out-of-the-box using Kryo,[9] because it is a recursive type in Scala. In order to use Scala's standard `List` type with Kryo, one must write a custom serializer, which would sidestep the objective of this benchmark, which is to compare the speed of *generated* picklers.

The results are shown in Figure 3.4 (a). As can be seen, Java is slower than the other frameworks. This is likely due to the expensive runtime cost of the JVM's calculation of the runtime transitive closure of the objects to be serialized. For 1,000,000 elements, Java finishes in 495ms while scala/pickling finishes in 74ms, or a factor 6.6 faster. As can be seen, the performance of our prototype is clearly faster than Kryo for small to moderate-sized collections; even though it remains faster throughout this benchmark, the gap between Kryo and scala/pickling shrinks for larger collections. For a `Vector[Int]` with 100,000 elements, Kryo v2 finishes in 36ms while scala/pickling finishes in 10ms–a factor of 3.6 in favor of scala/pickling. Conversely, for a Vector of 1,000,000 elements, Kryo finishes in 84ms whereas scala/pickling finishes in 74ms. This result clearly demonstrates the benefit of our hybrid compile-time/runtime approach:

[9] We register each class with Kryo, an optional step that improves performance.

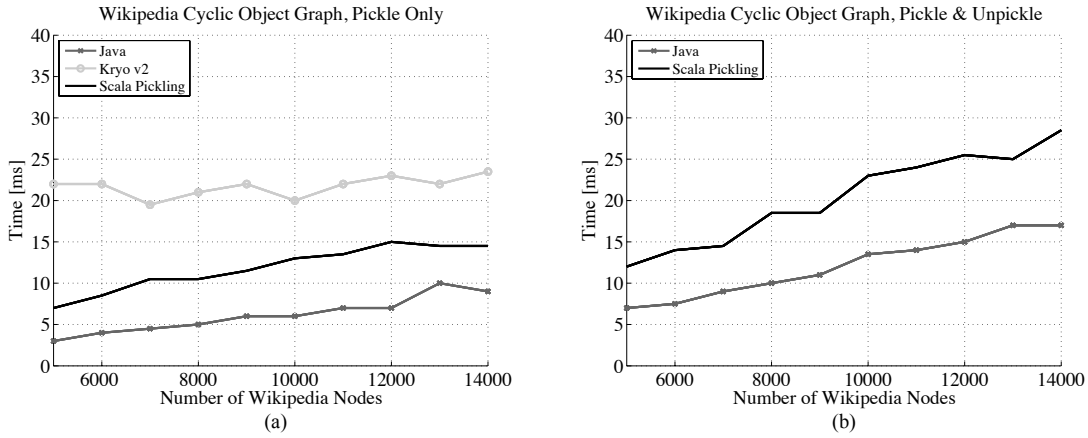


Figure 3.5 – Results for pickling/unpickling a partition of Wikipedia, represented as a graph with many cycles. Figure 3.5(a) shows a "pickling" benchmark across scala/pickling, Kryo, and Java. In Figure 3.5(b), results for a roundtrip pickling/unpickling is shown. Here, Kryo is removed because it crashes during unpickling.

while scala/pickling has to incur the overhead of tracking object identity in the case of general object graphs, in this case, the compile-time pickler generation is able to detect that object identity does not have to be tracked for the pickled data types. Moreover, it is possible to provide a size hint to the pickle builder, enabling the use of a fixed-size array as the target for the pickled data. We have found that those two optimizations, which require the kind of static checking that scala/pickling is able to do, can lead to significant performance improvements. The performance of manually written pickler combinators, however, is still considerably better. This is likely due to the fact that pickler combinators require no runtime checks whatsoever– pickler combinators are defined per type, and manually composed, requiring no such check. In principle, it should be possible to generate code that is as fast as these pickler combinators in the case where static picklers can be generated.
Figure 3.4 (b) shows the corresponding memory usage; on the y-axis the value of `System.freeMemory` is shown. This plot reveals evidence of a key property of Kryo, namely (a) that its memory usage is quite high compared to other frameworks, and (b) that its serialization is stateful because of internal buffering. In fact, when preparing these benchmarks we had to manually adjust Kryo buffer sizes several times to avoid buffer overflows. It turns out the main reason for this is that Kryo reuses buffers whenever possible when serializing one object after the other. In many cases, the newly pickled object is simply appended at the current position in the existing buffer which results in unexpected buffer growth. Our framework does not do any buffering which makes its behavior very predictable, but does not necessarily maximize its performance.

Finally, Figure 3.4 (c) shows the relative sizes of the serialized data. For a `Vector[Int]` of 1,000,000 elements, Java required 10,322,966 bytes. As can be seen, all other frameworks perform on par with another, requiring about 40% of the size of Java's binary format. Or, in

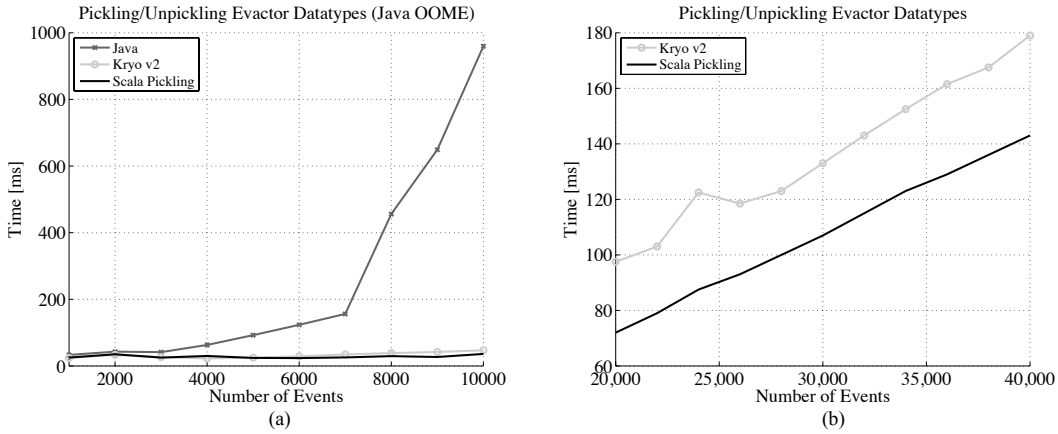


Figure 3.6 – Results for pickling/unpickling evactor datatypes (numerous tiny messages represented as case classes containing primitive fields.) Figure 3.6(a) shows a benchmark which pickles/unpickles up to 10,000 evactor messages. Java runs out of memory at this point. Figure 3.6(b) removes Java and scales up the benchmark to more evactor events.

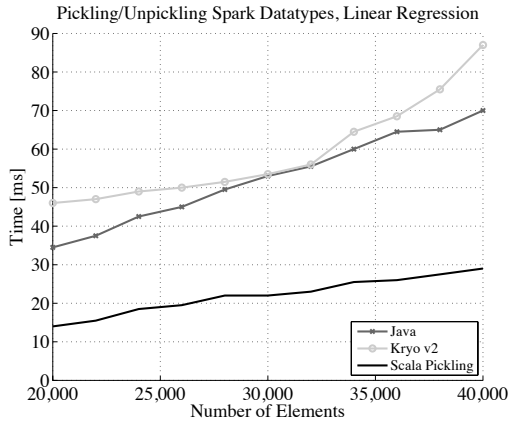


Figure 3.7 – Results for pickling/unpickling data points from an implementation of linear regression using Spark.

order of largest to smallest; Kryo v1 - 4,201,152 bytes; Kryo v2 - 4,088,570 bytes; scala/pickling 4,000,031 bytes; and Pickler Combinators 4,000,004 bytes.

3.6.3 Wikipedia: Cyclic Object Graphs

In the second benchmark, we evaluate the performance of our framework when pickling object graphs with cycles. Using real data from the Wikipedia project, the benchmark builds a graph where nodes are Wikipedia articles and edges are references between articles. In this benchmark we compare against Java's native serialization and Kryo. Our objective was to measure the full round-trip time (pickling and unpickling) for all frameworks. However, Kryo consistently crashed in the unpickling phase despite several work-around attempts. Thus, we

include the results of two experiments: (1) “pickle only”, and (2) “pickle and unpickle”. The results show that Java’s native serialization performs particularly well in this benchmark. In the “pickle only” benchmark of Figure 3.5 between 12000 and 14000 nodes, Java takes only between 7ms and 10ms, whereas scala/pickling takes around 15ms. Kryo performs significantly worse, with a time between 22ms and 24ms. In the “pickle and unpickle” benchmark of Figure 3.5, the gap between Java and scala/pickling is similar to the “pickle only” case: Java takes between 15ms and 18ms, whereas scala/pickling takes between 25ms and 28ms.

3.6.4 Microbenchmark: Evactor

The Evactor benchmark evaluates the performance of pickling a large number of small objects (in this case, events exchanged by actors). The benchmark creates a large number of events using the datatypes of the Evactor complex event processor; all created events are inserted into a collection and then pickled, and finally unpickled. As the results in Figure 3.6 show, Java serialization struggles with extreme memory consumption and crashes with an out-of-memory error when a collection with more than 10000 events is pickled. Both Kryo and scala/pickling handle this very high number of events without issue. To compare Kryo and scala/pickling more closely we did another experiment with an even higher number of events, this time leaving out Java. The results are shown on the right-hand side of Figure 3.6. At 40000 events, Kryo finishes after about 180ms, whereas scala/pickling finishes after about 144ms–a performance gain of about 25%.

3.6.5 Microbenchmark: Spark

Spark is a popular distributed in-memory collections abstraction for interactively manipulating big data. The Spark benchmark compares performance of scala/pickling, Java, and Kryo when pickling data types from Spark’s implementation of linear regression.

Over the course of the benchmark, frameworks pickle and unpickle an `ArrayBuffer` of data points that each consist of a double and an accompanying `spark.util.Vector`, which is a specialized wrapper over an array of 10 `Doubles`. Here we use a mutable buffer as a container for data elements instead of more typical lists and vectors from Scala’s standard library, because that’s the data structure of choice for Spark to internally partition and represent its data.

The results are shown in Figure 3.7, with Java and Kryo running in comparable time and scala/pickling consistently outperforming both of them. For example, for a dataset of 40000 points, it takes Java 68ms and Kryo 86ms to perform a pickling/unpickling roundtrip, whereas scala/pickling completes in 28ms, a speedup of about 2.4x compared to Java and about 3.0x compared to Kryo.

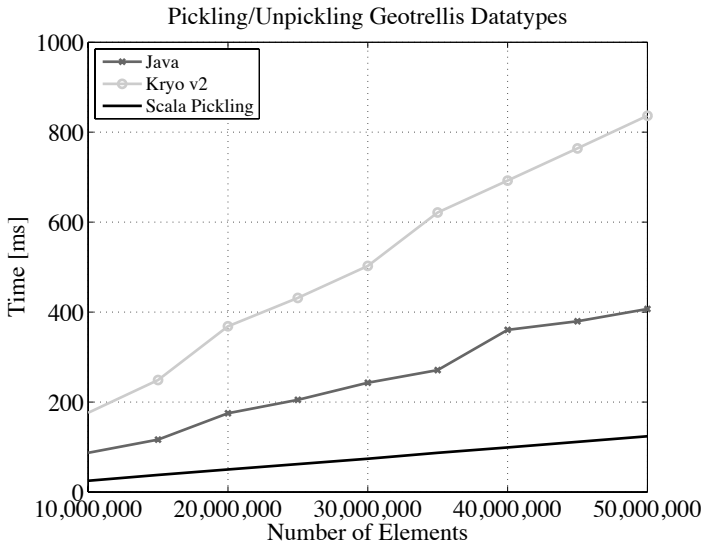


Figure 3.8 – Results for pickling/unpickling geotrellis datatypes (case classes and large primitive arrays).

	primitives/ primitive arrays	value-like types	collections	case classes	type descriptor	generics	subtyping polymorphism
GeoTrellis (Akka)	●	○	○	●	○	○	○
Evactor (Akka)	●	◐	◐	●	○	○	◐
Spark	●	●	●	◐	○	○	○
Storm	○	●	●	N/A	○	○	◐
Twitter Chill	○	◐	●	◐	◐	◐	◐

Legend: ●: Heavy Use ◐: Light Use ○: No Use

Figure 3.9 – Scala types used in industrial distributed frameworks and applications.

3.6.6 Microbenchmark: GeoTrellis

GeoTrellis [Azavea, 2010] is a geographic data processing engine for high performance applications used by the US federal government among others.

In this benchmark one of the main message classes used in GeoTrellis is pickled. The class is a simple case class containing a primitive array of integers (expected to be large). Figure 3.8 shows the time it takes to pickle and unpickle an instance of this case class varying the size of the contained array.

The plot shows that Java serialization performs, compared to Kryo, surprisingly well in this benchmark, *e.g.*, a roundtrip for 50000000 elements takes Java 406ms, whereas Kryo is more than two times slower at 836ms. It is likely that modern JVMs support arrays of primitive types well, which is the dominating factor in this case. Scala/pickling is still significantly faster with 124ms, since the static type of the array is final, so that efficient array-pickling code can be

generated at compile time.

3.6.7 Data Types in Distributed Frameworks and Applications

Figure 3.9 shows a summary of the most important data types used in popular distributed computing frameworks like Spark [Zaharia et al., 2012] and Storm [Nathan Marz and James Xu and Jason Jackson et al., 2012]. The fully shaded circles in the table representing "heavy use" means either (a) a feature is used frequently in application-level data types or (b) a feature is used frequently in data types that the framework registers with its underlying serialization system. Half-shaded circles in the table representing "light use" mean a feature is used only infrequently in the data types used in applications or registered by frameworks. We categorize the data types shown in this table into two groups.

In the first group at the top are distributed *applications* using data types suitable for distributed event processing and message passing. We consider two representative open-source applications: GeoTrellis and Evactor. Both applications use Akka [Typesafe, 2009], an event-driven middleware for distributed message passing. However, the properties of the exchanged messages are markedly different. Messages in GeoTrellis typically contain large amounts of geographic raster data, stored in arrays of primitives. Messages in Evactor represent individual events which typically contain only a few values of primitive types. Both applications make use of Scala's case classes which are most commonly used as message types in actor-based applications.

The second group in the bottom half of Figure 3.9 consists of distributed computing frameworks. What this table suggests is that the majority of distributed computing frameworks and applications requires pickling collections of various types. Interestingly, application-level data types tend to use arrays with primitive element type; a sign that there is a great need to provide easier ways to process "big data" efficiently. From the table it is also clear that case classes tend to be primarily of interest to application code whereas frameworks like Spark tend to prefer the use of simple collections of primitive type internally. What's more, the demand for pickling generics seems to be lower than the need to support subtyping polymorphism (our framework supports both, though). At least in one case (Twitter's Chill [Oscar Boykin and Mike Gagnon and Sam Ritchie, 2012]) a framework explicitly serializes manifests, type descriptors for Scala types, which are superceded by type tags. The shaded area (which groups "heavily-used" features across applications/frameworks) shows that collections are often used in distributed code, in particular with primitive element types. This motivates the choice of our collections micro benchmark.

3.7 Related Work

Some OO languages like Java and runtime environments like the JVM or .NET provide serialization for arbitrary types, provided entirely by the underlying virtual machine. While this

approach is very convenient for the programmer, there are also several issues: (a) the pickling format cannot be exchanged (Java), (b) serialization relies on runtime reflection which hits performance, and (c) existing classes that do not extend a special marker interface are not serializable, which often causes oversights resulting in software engineering costs. In functional languages, pickler combinators [Elsman, 2005, Kennedy, 2004] can reduce the effort of manually writing pickling and unpickling functions to a large extent. However, existing approaches do not support object-oriented concepts such as subtyping polymorphism. Moreover, it is not clear whether local type inference as required in OO languages would yield a comparable degree of conciseness, acceptable to programmers used to Java-style serialization. Nonetheless, our approach builds on pickler combinators, capitalizing on their powerful composability.

Our approach of retrofitting existing types with pickling support builds on implicits in Scala [Oliveira et al., 2010] and is reminiscent of other type-class-like mechanisms, such as JavaGI [Wehr and Thiemann, 2011] or C++ Concepts [Reis and Stroustrup, 2006].

Additionally, in an effort to further reduce the boilerplate required to define or compose picklers using existing picklers, we present a framework for automatically generating picklers for compound types based on picklers for their component types. Given the close relationship of our implicit picklers to type classes, this generation mechanism is related to Haskell's *deriving* mechanism [Magalhães et al., 2010]. One of the main differences is that our mechanism is faithful to subtyping. So far, as presented in this chapter, our mechanism is specialized for pickling; an extension to a generic mechanism for composing type class instances is described in Chapter 4.

Pickling in programming languages has a long history dating back to CLU [Herlihy and Liskov, 1982] and Modula-3 [Cardelli et al., 1989]. The most closely-related contemporary work is in two areas. First, pickling in object-oriented languages, for example, in Java (see the Java Object Serialization Specification [Oracle, Inc., 2011]), in .NET, and in Python [van Rossum, 2007]; second, work on pickler combinators in functional languages which we have already discussed in the introduction. The main difference of our framework compared to pickling, or serialization, in widespread OO languages is that our approach does not require special support by the underlying runtime. In fact, the core concepts of object-oriented picklers as presented in this chapter can be realized in most OO languages with generics.

While work on pickling is typically focused on finding optimally compact representations for data [Vytiniotis and Kennedy, 2010], not all work has focused only on distribution and persistence of ground values. Pickling has also been used to distribute and persist code to implement module systems [Rossberg, 2007, Roy, 1999]. Similar to our approach, but in a non-OO context, AliceML's HOT pickles [Rossberg et al., 2007] are universal in the sense that any value can be pickled. While HOT pickles are deeply integrated into language and runtime, scala/pickling exists as a macro-based library, enabling further extensibility, *e.g.*, user-defined pickle formats can be interchanged.

There is a body of work on maximizing sharing of runtime data structures [Appel and Gonçalves, 1993, Elsman, 2005, Tack et al., 2006] which we believe could be applied to the pickler combinators presented in Section 3.3; however, a complete solution is beyond the scope of the present work.

3.8 Conclusion

We have introduced a model of pickler combinators which supports core concepts of object-oriented programming including subtyping polymorphism with open class hierarchies. Furthermore, we have shown how this model can be augmented by a composable mechanism for static pickler generation which is effective in reducing boilerplate and in ensuring efficient pickling. Thanks to a design akin to an object-oriented variation of type classes known from functional programming, the presented framework enables retrofitting existing types and third-party libraries with pickling support. Experiments suggest that static generation of pickler combinators can outperform state-of-the-art serialization frameworks and significantly reduce memory usage.

4 Static and Extensible Datatype Generic Programming

In the previous chapter, we covered *object oriented picklers*, and we had a glimpse of an associated mechanism for the automatic generation of these type class-based picklers. In this chapter, we generalize our generation technique from picklers to arbitrary type class instances.

4.1 Introduction

Defining functionality that should apply to a large set of types is a common problem faced by both language designers and normal users. One common approach is to provide specialized functionality across arbitrary types at the level of the compiler or runtime. For example, in Java, every object is synthetically provided with a few methods; `toString`, `equals`, `clone`, and `hashCode`. Serialization, on the other hand, is also an ubiquitously needed functionality, but unlike the above, Java does not ensure that serialization functionality exists for every type. Instead, serialization in Java is opt-in; if a class implements a `Serializable` interface then instances of that class are automatically serializable by the JVM. While compiler/runtime-integrated approaches such as Java's serialization are typically easy to use (no boilerplate required), they are inflexible and are often impossible to customize. For example, it is not possible to adapt Java serialization to work with other formats (such as JSON or XML).

Library-based approaches to generic programming which require *type classes* [Wadler and Blott, 1989] as a language feature are a lot more flexible. Type classes provide a mechanism where a certain functionality can be captured in an interface. When programmers need certain types of values to support a given functionality, they can implement an *instance* of a type class. Type classes support *retroactive extensibility* [Lämmel and Ostermann, 2006]; functionality can be implemented *after* the type or class has been defined. This is in contrast with conventional OO programming, where all methods (such as `toString` or `equals`) are implemented together with the definition of the class. Retroactive extensibility enables flexibility and the possibility to customize behavior. As a result, several authors have argued for the software engineering benefits of using type classes [Lämmel and Ostermann, 2006, Oliveira et al., 2010], and Scala has embraced them [Miller et al., 2013, Odersky and Moors, 2009, Oliveira et al., 2010].

When comparing type classes to baked-in functionalities (*e.g.,* Java serialization), it's clear that an approach for adding functionality based on type classes is more general, since most any functionality (including serialization) can be modeled as a type class. However, an approach based on type classes is not without challenges. To provide functionality across a large number of types, users are required to implement many type class instances manually, one by one. To reduce this vast amount of boilerplate, there have been a number of proposals for *datatype-generic programming* (DGP) [Hinze et al., 2007, Rodriguez et al., 2008].

DGP is an advanced form of *generic programming* [Musser and Stepanov, 1989], where generic functions can be defined by inspecting the structure of types. DGP approaches are typically library-based, and as such they typically introduce many run-time representations, thus typically incurring significant performance penalties [Adams and DuBuisson, 2012]. Furthermore, the vast majority of DGP approaches has been developed for Haskell, and are thus fundamentally limited when ported to mainstream OO languages, due to their lack of support for subtyping or object identity. A DGP approach appropriate for use in Scala must account for such features.

Baked-in compiler-based approaches to adding functionality is at odds with library-based approaches using type classes. On the one hand, language-integrated approaches can be more powerful in the sense that they can do a great deal of static analysis, and because they are so specialized, typically require no boilerplate to programmers. However this is done at the cost of customizability and extensibility – users typically can't override or customize statically-added behavior. On the other hand, with type class-based approaches, one must contend with an enormous amount of boilerplate or pay a non-negligible performance penalty[1]. In all cases, however, type class-based approaches offer no way to statically restrict runtime behavior.

Perhaps the most important limitation of DGP approaches in the context of mainstream languages is the lack of support for pervasively used object-oriented features such as subtyping and object identity, which so far have not been addressed, except for specialized functionality [Miller et al., 2013].

4.1.1 Design Constraints

This chapter details an approach that strikes a sweet spot in the design space. The approach is guided by the following principles:

- **Extensibility and customizability**. Like for type class-based approaches, retroactive extensibility and type-based customization should be supported.
- **Little boilerplate**. Like language-integrated approaches, usage of generic code should *feel* built-in. Users shouldn't have to define type class instances or provide a lot of scaffolding.

[1] Some approaches trade type-safety for performance [Adams and DuBuisson, 2012].

- **Performance**. Generic functions written by library authors or library users should have the same or better performance than approaches with compiler/runtime support.
- **Generality**. In addition to generic functions, lightweight static analysis capabilities should be supported.

The pickling framework, scala/pickling [Miller et al., 2013], presesnted in the previous chapter, sought to achieve many of these goals for one particular application: serialization. scala/pickling is based on type classes which are generated and composed at compile time, according to their type signatures. Due to its compile-time properties, serialization code is fast and inlined, without requiring any boilerplate. Due to the fact that it is completely based upon type classes, flexibility and extensibility come for free. However, the approach is specialized on providing type class instances for only the `Pickling` type class. Other type classes or generic functions are not supported.

4.1.2 Contributions

This chapter presents *Self Assembly*, a general technique or pattern for:

- Defining generic operations or properties that operate over a large class of types with little boilerplate and good performance (these operations are statically generated). Importantly, the technique supports many features of mainstream OO languages such as subtyping, object identity, and separate compilation.
- Defining additional lightweight static type checking via *generic properties*. Such lightweight static checks can guarantee that a certain property (checked by tying together other static anaysis frameworks with the help of type classes), *e.g.*, deep immutability, holds. In this case, if a class is immutable, the immutability checker generates a type class instance for that class, which certifies that property.

The DGP-related contributions of this thesis include:

- **Self-Assembly**, a general technique for defining generic operations or properties that operate over a large class of types that requires little boilerplate; shares the extensibility and customizability properties of type classes; and, due to compile-time code generation, provides high performance. It allows defining generic functions in a statically type-safe way.
- **A full-featured DGP approach for OOP.** `self-assembly` enables the definition of datatype-generic functions that support features present in production OO languages, including subtyping, object identity, and generics.
- **Support for generic properties.** `self-assembly` enables the definition of custom lightweight static type checks to guarantee that certain static properties hold at runtime, *e.g.*, immutability.

- **The `self-assembly` library**, a complete and full-featured implementation of our technique in and for Scala. The library includes several auxiliary definitions, such as generic queries and transformations, that help define new lightweight static checks of generic properties. Importantly, `self-assembly` doesn't require any extension to the language or compiler.
- **A case study on basing scala/pickling on `self-assembly`.** We evaluate the expressivity and performance of `self-assembly` by porting a full-featured serialization framework, keeping the same published performance numbers while reducing the code size for type class instance generation by 56%.

4.2 Type Classes and a Boilerplate Problem

This section provides an introduction to type classes [Wadler and Blott, 1989] and reviews how to encode them in Scala using implicits and conventional OO features [Oliveira et al., 2010]. This section also observes that type class instances for various types tend to require code that follows a common pattern. The pattern can be viewed as a source of code boilerplate, since similar code needs to be repeated throughout several definitions. The remainder of the chapter aims at showing how to capture the pattern as reusable code and generate type class instances automatically from that code.

4.2.1 Implicits

Implicit Parameters. In Scala, it is possible to select values automatically based on type. These capabilities are enabled when using the `implicit` keyword. For example, a method `log` with multiple parameter lists may annotate their last parameter list using the `implicit` keyword.

```
def log(msg: String)(implicit o: PrintStream) =
  o.println(msg)
```

This means that in an invocation of `log`, the implicit argument list may be omitted if, for each parameter of that list, there is exactly one value of the right type in the *implicit scope*. The implicit scope is an adaptation of the regular variable scope. Imported implicits, or implicits declared in an enclosing scope are contained in the implicit scope of a method invocation.

```
implicit val out = System.out
log("Does not compute!")
```

In the above example, the implicit val `out` is in the implicit scope of the invocation of `log`. Since `out` has the right type, it is automatically selected as an implicit argument.

Implicit Conversions. Implicit conversions can be thought of as methods which, like implicit parameters, can be implicitly selected (*i.e.,* invoked) based upon their type, and whether or not they are present in implicit scope. As with implicit parameters, implicit conversions also carry the `implicit` keyword before their declaration.

```
implicit def intWrapper(x: Int): Message =
   new Message {
     def message: String = "secret message!"
   }
```

In the example above, assuming there exists an abstract class `Message` with abstract method `message`, the implicit conversion `intWrapper` will be triggered when a method called `message` is called on an `Int`. That is, simply calling `39.message` will result in "secret message!" being returned. Since the implicit conversion has the effect of adding a "new" method to type `Int`, `message` is typically called an *extension method*. In our framework we use implicit conversions, for example, for adding a `pickle` method to arbitrary objects.

4.2.2 Type Classes

Type classes are a language mechanism that provide a disciplined alternative to ad-hoc polymorphism. They have been popularized by Haskell. Type classes allow functions to be defined over a set of types. If values of a type `T` should provide a certain functionality then that functionality can be specified as an *instance* of a type class.

```
trait Show[T] {def show(visitee : T) : String}

implicit object IntInstance extends Show[Int] {
  def show(o : Int) = o.toString()
}
```

Figure 4.1 – |Show| type class and corresponding instance for integers.

In Scala type classes can be implemented using a combination of standard OO features (traits, classes and objects) and implicits [Oliveira et al., 2010]. The Scala encoding of type classes is essentially a *design pattern* [Gamma et al., 1995]: instead of having built-in language concepts for type classes, Scala uses general language features to model type classes. A type class is simply an interface that provides operations over one (or more) generic types. Such interfaces can be modeled as traits in Scala. An example of a type class is shown in Figure 4.1. The trait `Show[T]` models a type class that provides pretty printing functionality for some type `T` via a method `show`.

The main conceptual difference between standard OO methods and type-class methods is that the later are provided *externally* to objects. Suppose that we wanted to add pretty printing

functionality to integers. To do this we create an instance of the type class `Show` where the generic type parameter `T` is instantiated to `Int`. The `object IntInstance` in Figure 4.1 models such instance in Scala using regular objects. In that object, the `show` method takes an argument `o` of type `Int` an invokes the `toString()` method on `o`.

Type-Directed Resolution of Instances An interesting aspect of type classes is that instances can be automatically determined using a type-directed resolution mechanism. This type-directed resolution mechanism allows type classes to be used from client code through a mechanism similar to overloading. This is achieved in Scala using an implicit parameter:

```
def ishow[T](v : T)(implicit showT : Show[T]) =
  showT.show(v)
```

In `ishow` the idea is that the method takes two parameters, with the last of these (`showT`) being implicit. As we have seen in Section 4.2.1 this means that the second parameter can be automatically determined by the compiler. For example if we wanted to use `show` on integers we could simply write a program such as:

```
def test1 = ishow(5)
```

Provided that an `implicit` value of type `Show[Int]` is in the implicit scope (for example `IntInstance` from Figure 4.1), the second parameter is automatically inferred by the compiler.

Context Bounds Type classes are pervasively used in Scala. Because of this Scala offers an alternative convinient syntax sugar called *context bounds*. Context bounds allows code using type classes to be written more compactly and arguably more intuitively. With context bounds, instead of writting `ishow` we could write:

```
def show[T : Show](v : T) =
  implicitly[Show[T]].show(v)
```

The idea of context bounds comes from the fact that type classes can also be seen as a generic programming mechanism [Musser and Stepanov, 1989], which allows generic parameters to be constrained. In this case the type of `show` can be read as a generic method where the generic type argument must be an instance of `Show`. A small problem with context bounds there is no parameter name to be used in the definition of `show`. However, it is possible to *query* the implicit scope for a value of a certain type using a simple auxiliary method called `implicitly`:

```
sealed trait Tree
case class Fork(left : Tree, right : Tree)
  extends Tree
case class Leaf(elem : Int) extends Tree

implicit object TreeInst extends Show[Tree] {
 def show(visitee : Tree) : String = visitee match {
   case Fork(l,r) =>
    "Fork(" + show(l) + ", " + show(r) + ")"
   case Leaf(x) => "Leaf(" + x.toString() + ")"
 }}
```

Figure 4.2 – Trees of integers and corresponding Show instance.

```
def implicitly[T](implicit x : T) : T = x
```

This precludes the need for having to have the name of the implicit argument in hand in order to use it. From the client perspective, using show is similar to using ishow.

4.2.3 Pretty Printing Complex Structures

Of course it is also possible to apply type classes to more complex structures. For example consider a simple type of binary trees with integers at the leafs. Figure 4.2 shows how to model such trees in Scala using *case classes* [Emir et al., 2007] and *sealed traits*. The keyword `sealed` in Scala means that the trait can only be implemented by definitions in the existing compilation unit. Together with case classes this allows modeling *algebraic datatypes*, which are a well-known concept from functional programming. The `Tree` trait is the type of trees. The case class `Fork` models the binary nodes of the tree, wheres the case class `Leaf` models the leaves containing an integer value.

To define pretty printing for `Tree` using the `Show` type class we create an `object TreeInst`. This object provides a definition for the `show` method that pattern matches on the two tree constructors (cases) of `Tree`. The implementation of the two cases is unremarkable: both cases print the constructors names and the arguments.

A simple test program illustrating the use of `TreeInst` is shown next. The value `tree` defines a simple tree and the definition `test3` pretty prints that tree.

```
val tree : Tree = Fork(Fork(Leaf(3),Leaf(4)),Leaf(5))
def test3 = show(tree)
```

```
sealed trait PTree[A]
case class Branch[A](x: A, l: PTree[A], r: PTree[A])
  extends PTree[A]
case class Empty[A] extends PTree[A]

implicit def PTreeInst[A : Show] : Show[PTree[A]] =
  new Show[PTree[A]] {
    def show(visitee : PTree[A]) = visitee match {
      case Branch(x,l,r) =>
        "Branch(" + implicitly[Show[A]].show(x) +
        ", " + show(l) + ", " + show(r) + ")"
      case Empty() => "Empty()"
  }}
```

Figure 4.3 – Parametrized trees and corresponding Show instance.

Recursive Resolution and Compositionality of Instances Another interesting aspect of type classes is that they provide a highly compositional way to define instances. Lets consider a variant of trees, shown in Figure 4.3, which is parametrized by some element type A. The type these trees is PTree[A] and there are two types of nodes: Branch nodes with an element of type A and two branches; and Empty nodes with no content.

Like other types it is possible to define an instance (PTreeInst) for the type PTree[A]. However in order to pretty print such trees it is necessary to know how to print the elements of type A as well. To accomplish this we require that the generic type parameter A has a Show instance using a context bound. To print the elements in the Branch case, the instance can be retrieved from the implicit scope using implicitly and then used to print the element. With this instance it is possible to print trees with integer elements, such as:

```
val ptree : PTree[Int] = Branch(5,Empty,Empty)
def test4 = show(ptree)
```

However, more interestingly, it is also possible to print trees where for any element type that has a Show instance. For example:

```
val ptree2 : PTree[PTree[Tree]] =
  Branch(Branch(tree,Empty,Empty),Empty,Empty)
def test5 = show(ptree2)
```

Here ptree2 has elements of type PTree[Tree]. To print ptree2 the instance for PTree is used twice: once for values of type PTree[PTree[Tree]]; and another time for values of type PTree[Tree]. In fact it is possible to use arbitrarely many instances of the various types

(possible multiple times) during type-directed resolution, which makes the process very compositional. This is possible because the type-directed resolution mechanism is recursive.

4.2.4 A Boilerplate Problem

Although type classes are nice, they often require similar code for different instances. For example consider the two instances in Figures 4.2 and 4.3. The code that is needed in both instances is quite similar and it follows a common pattern: for each case the constructor name and parameters are printed. Therefore code tends to be quite similar across instances. This code can be viewed as a form of boilerplate since we could hope that it could be mechanically generated.

4.3 Type-Safe Meta-Programming in Scala

Scala macros [Burmako, 2013, Burmako and Odersky, 2012] enable a form of type-safe meta-programming. Macros are methods that are invoked at compile time. Instead of runtime values, macros operate on and return typed expression trees. In the following we provide an overview of macros, type checking, and properties.

4.3.1 Definition

Macro defs are methods that are transparently loaded by the compiler and executed (or expanded) during compilation. A macro is defined like any normal method, but it is linked using the `macro` keyword to an additional method that provides its implementation, which operates on expression trees.

```
def assert(x: Boolean, msg: String): Unit =
  macro assert_impl
def assert_impl(c: Context)
  (x: c.Expr[Boolean], msg: c.Expr[String]):
                            c.Expr[Unit] = ...
```

In the above example, the parameters of `assert_impl` are typed expression trees, which the body of `assert_impl` operates on, itself returning an expression of type `Expr[Unit]`. `assert_impl` is evaluated at compile time, and its result is inlined at the call site of `assert`. Note that expression trees are typed, *i.e.*, `assert`'s parameter of type `Boolean` corresponds to a typed expression tree of type `Expr[Boolean]`.

In the type-safe subset of macros that we consider in this chapter, expression trees are built using `reify/splice`:

```
val expr: c.Expr[Boolean] = reify {
  if (x.splice > 10) x.splice
  else true
}
```

Here, the body of `reify` consists of regular Scala code. Expressions in the enclosing scope are spliced into the result expression using the `splice` method. Importantly, the code within `reify` is type-checked at its definition site. This means, for the above code, Scala's type checker reports type errors not in terms of the generated code, but in terms of the high-level user-written code.

Due to limitations in the reify API, we use quasiquotes (typechecked during macro expansion) to circumvent the above type-checking in a small trusted core of `self-assembly`, shielded from users. However, we never lose soundness, since, unlike MetaML [Taha and Sheard, 2000], all splicing is done at compile time, and generated expressions are always re-type-checked after expansion.

4.3.2 Properties

Constant Type Signatures In this work, we focus on one of two macro def varieties: "black-box" macros. In this case, the type signature of the macro provides all information necessary for type-checking all of its invocations. That is, the macro does not have to be expanded prior to type-checking. This has important software engineering benefits, namely that abstract, type-based reasoning about programs is maintained independently of the macro's corresponding implementation. This is particularly useful when reasoning about the result type of a macro. For blackbox macros, the implementation (and expansion) is not required to determine the result type.

Local Expansion Since macros are simply methods that are invoked at compile time, they are expanded and inlined at invocation site. For this reason, we consider macro defs to be "local compiler extensions." They cannot change the compiler's global symbol table. Thus, they cannot introduce new top-level type definitions.

4.4 Basic Self-Assembly

Section 4.2 showed how to write type classes like `Show[T]` manually, pointing out a source of significant boilerplate code. In section 4.4.1, we outline the basic usage of the `self-assembly` library, which allows defining type classes desired in a way where the required boilerplate for defining such type classes is automatically generated. Section 4.4.2 explains the mechanics of the automatic type class generation implemented in the `self-assembly` library. Section 4.4.3 outlines how one can customize the generation of type classes for specific types.

```
object Show extends Query[String] {
  def mkTrees[C <: SContext](c: C) = new Trees(c)

  class Trees[C <: SContext](override val c: C)
      extends super.Trees(c) {
    import c.universe._
    type SExpr = c.Expr[String]

    def combine(left: SExpr, right: SExpr) =
      reify { left.splice + right.splice }

    def delimit(tpe: c.Type) = {
      val start = constant(tpe.toString + "(")
      (start, reify(", "), reify(")"))
    } }

  implicit def generate[T]: Show[T] =
    macro genQuery[T, this.type]
}
```

Figure 4.4 – Implementing the Show type class using `self-assembly`.

4.4.1 Basic Usage

The `self-assembly` library allows implementing type classes instances automatically on demand at compile time. This main idea is introduced using the simple Show type class in Figure 4.1. Section 4.6 shows how our approach extends to different forms of type classes, commonly referred to as queries and transformations [Lämmel and Peyton Jones, 2003].

Generating Instances for Show Suppose a user wants to provide instances of `Show[T]` for as many types as possible. Using `self-assembly` we can create a singleton object that extends a library-provided trait, and that implements two factory methods, `generate` and `mkTrees`. Figure 4.4 shows the Show companion object,[2] which extends the `Query` trait. The `mkTrees` factory method, abstract in `Query`, creates a new `Trees` instance; `Trees[C]` provides a number of methods that are invoked by the `self-assembly` library at *compile time* to obtain AST fragments that are inlined in the generated code. The Show type class converts objects to strings; thus, the query has to define how to assemble result strings, based on an associative combination operator (`combine`), begin/end delimiters (`first`/`last`), and a separator. As mentioned in Section 4.3, the syntax `reify { ... }` creates a typed expression based on Scala code. `left.splice` splices the expression `left` into the result expression. The compiler type-checks `reify` blocks at their definition site.

[2] A companion object is a singleton object with the same name as a trait.

Apart from implementing a subclass of `Trees[C]`, the `Show` singleton object also needs to define a generic implicit method (here, `generate`) that invokes the generation macro `genQuery`. The `genQuery` macro is provided by our library.[3]

Result With the `Show` singleton object defined as in Figure 4.4 it is no longer necessary for the user to define a type class instance for every single type manually. Instead, whenever an instance of type, say, `Show[MyClass]`, is required (typically, using an implicit parameter), Scala's type checker automatically inserts a call to the *implicit def* `generate[MyClass]`; this implicit def generates a suitable implementation of the searched type class instance on-the-fly. As a result, type class instances do not have to be defined manually.

4.4.2 Generation Mechanism

We illustrate the general idea of our generation technique through a simple example based solely on closed ADT-style datatypes in Scala. Such datatypes consist of either sealed traits or case classes extending such traits. In subsequent sections, we generalize this view to richer types.

Our treatment is centered on an example, in which, our goal is to automatically "derive" type class instances that "show" information about a given type. Think of it as a `toString` method that traverses the structure of a type, and nicely prints information about all of the fields of that type.

We structure our treatment into three distinct steps: (1) in Section 4.4.2, we show how our generation is triggered; (2) in Section 4.4.2, we explain our macro-based generation technique; (3) in Section 4.4.2, we show some example type class instances that result from our generation technique, and relate them to the type class pattern introduced in Section 4.2.2.

Triggering Generation

To be able to generate suitable instances for all possible types for which `Show[T]` can be defined, we put an implicit macro into the companion object of `Show[T]`. The fact that the implicit macro is inside the companion object means that whenever an instance `Show[S]` is requested, Scala's implicit lookup mechanism searches the members of the companion object `Show` where it finds the implicit macro:

```
object Show extends Query[String] {
  ...
  implicit def generate[T]: Show[T] =
    macro genQuery[T, this.type]
}
```

[3] The type argument `this.type` is the type of the enclosing singleton object; it is passed to `genQuery` to identify the type class and the `mkTrees` method that should be used by the library to generate instances.

```
trait Query[R] ... {
  def mkTrees[C <: Context with Singleton](c: C)
    : Trees[C]

  abstract class Trees[C <: Context with Singleton]
    (override val c: C) extends super.Trees(c) { }

  def genQuery[T:c.WeakTypeTag, S:c.WeakTypeTag]
    (c: Context): c.Tree = {
    import c.universe._
    val tpe = weakTypeOf[T]
    val stpe = weakTypeOf[S]
    val tpeOfTypeClass =
      stpe.typeSymbol.asClass.companion.asType
          .asClass.toTypeConstructor
    val qresTpe =
      tpeOfTypeClass.decls.head.asMethod.returnType

    val trees = mkTrees[c.type](c)
    ...
```

Figure 4.5 – Macro-based generation: set-up

Thus, the implicit lookup mechanism inserts an invocation of the macro method `genQuery`.

Macro-Based Generation

Being a macro, `genQuery` returns an abstract syntax tree instead of a (runtime) value. It is declared as follows:

```
def genQuery[T:c.WeakTypeTag, S:c.WeakTypeTag]
    (c: Context): c.Tree = ...
```

Note that in this declaration, the type parameters `T` and `S` are annotated with *context bounds* `c.WeakTypeTag`. First, the macro collects information about the types and the type class for which an instance should be generated. Second, the macro creates an instance of the user-provided `Trees` class by invoking the `mkTrees` factory method. These steps are shown in Figure 4.5.

The body of the type class is generated using:

```
    val tpe = weakTypeOf[T] // see Fig. 5
    ...
    val (first, separator, last) =
      trees.delimit(tpe)
    val body = trees.combine(
      fieldsExpr(first, separator), last)
```

To create the result expression, the macro utilizes the `trees` instance (of type `Trees`) that we initialize in the set-up phase (see Figure 4.5). Calling `delimit` returns three expressions ("delimiters") of type `Expr[R]` based on the reified type `tpe`. Recall that `tpe` corresponds to type parameter `T`, which is the type for which the macro generates a type class instance. The `fieldsExpr` method creates an `Expr[R]` by folding the `Expr[R]`s obtained for each field (see below) using the user-overridden `combine` method:

```
if (paramFields.size < 2)
  ...
else
  paramFields.tail.foldLeft(first) { (acc, sym) =>
    val withSep = trees.combine(acc, separator)
    trees.combine(withSep, fieldValue(sym))
  }
```

For example, Figure 4.4 shows that the definition of `combine` for `Show` is just string concatenation. As a result, this code concatenates the string values of all fields separated with `separator`.

The expression tree `fieldValue(sym)` is obtained as follows. For each field declared in type `tpe`, the following subexpression is generated:

```
val symTp = sym.typeSignatureIn(tpe)
val fieldName = sym.name.toString.trim
trees.fieldValueExpr(visitee, fieldName,
  symTp, tpeOfTypeClass)
```

The invocation of `fieldValueExpr` expands to (a) a nested look-up of a type class instance for the field, and (b) an invocation of the type class method:

```
    def fieldValueExpr(visitee: c.Expr[T], name: String,
      tpe: c.Type, tpeOfTypeClass: c.Type): c.Expr[R] =
    c.Expr[R](
      q"""
        implicitly[${appliedType(tpeOfTypeClass, tpe)}]
          .apply($visitee.${TermName(name)})
      """)
```

```
① implicit object CShowInstance extends Show[C] {
  ② def show(visitee: C): String = {
       var result = "C("

     ③ val inst_1 = implicitly[Show[D1]]
     ④ result += inst_1.show(visitee.p_1)
  ⑤    ...
       val inst_n = implicitly[Show[DN]]
       result += inst_n.show(visitee.p_n)
       result += ")"
     }
  }
```

Figure 4.6 – Basic generation of type classes.

The syntax `q"""..."""` indicates the use of a quasiquote to create an *untyped* tree that is cast to an `Expr[R]`, effectively forming part of a small trusted core of `self-assembly`. The main reason for creating an untyped tree at this point is that the value of field "name" is obtained using only the field's name–the selection `$visitee.${TermName(name)}` must fundamentally be untyped. It is clear, though, that the result will be of type `R`, since that's the result type of all type class instances of type `tpeOfTypeClass`.

Generated Type Class Instances

The generation technique explained in the previous section produces implicit (singleton) objects which correspond to the type class instances portion of the type class pattern introduced in Section 4.2.2.

Let's say the datatype that we'd like to call `show` on is the `Tree` type in Figure 4.2. In order to create a type class instance of type `Show[Tree]`, we also create type class instances for `Tree`'s two subclasses, `Fork` and `Leaf`. `Fork` and `Leaf` are case classes with the general shape:

```
case class C(p_1: D_1, ..., p_n: D_n)
  extends E_1 with ... with E_m { ... }
```

An arbitrary type class instance (implicit singleton object) can be generated using the technique described in the previous section. Figure 4.6 shows the general structure that is generated for an arbitrary shape `C`. The implicit object (1) is exactly the same as in the manual type class pattern described in Section 4.2.2. (2) is the implementation of the single abstract method of the type class (the `show` method of the `Show` trait). (3) is the result of expanding the `implicitly` invocation within the method `fieldValueExpr` above. (4) corresponds to the accumulation logic which itself results from the fold of `paramFields` above (to simplify the presentation we use the `result` accumulator variable instead of a deeply nested tree). Finally, (5) corresponds to `first` and `last` in the body of the macro-generated implementation of `Show`'s single abstract method, `show`.

4.4.3 Customization

Generation as provided by `self-assembly` is convenient, but in some cases it is desirable to have full control over the type class instances for specific types (one strength of the type class pattern as introduced in Section 4.2.2). When using the `self-assembly` library, customization is still possible. It is sufficient to define custom instances for selected types manually; these custom instances are then transparently picked up and chosen in place of automatically-generated ones. It is even possible to use Scala's scoping and implicit precedence rules to prioritize certain instances over others.

4.5 Self-Assembly for Object Orientation

A cornerstone of the design of `self-assembly` is its support for features of mainstream OO languages. The following Section 4.5.1 explains how our approach supports subtyping polymorphism in the context of open class hierarchies (Section 4.5.1) and separate compilation (Section 4.5.1). In Section 4.5.2 we discuss how `self-assembly` handles cyclic object graphs, which are easily created using mutable objects with identity.

4.5.1 Subtyping

Object-oriented languages like Java or Scala enable the definition of a *subtyping relation* based on class hierarchies. Given the pervasive use of subtyping in typical object-oriented programs, our approach is designed to account for *subtyping polymorphism*. In addition, we provide mechanisms that enable the object-oriented features even in a setting where modules/packages are separately compiled.

Open Hierarchies

Classes defined in languages like Java are by default "open," which means that they can have an unbounded number of subclasses spread across several compilation units. By contrast, *final classes* cannot have subclasses at all. In addition, *sealed classes* in Scala can only have subclasses defined within the same compilation unit.
Our approach enables the generation of type class instances even for open classes. For example, consider the class hierarchy shown in Figure 4.7. The `self-assembly` library can automatically generate an instance for type `Person`:

```
val em = Employee("Dave", 35, 80000)
val ff = Firefighter("Jim", 40, 2004)
val inst = implicitly[Show[Person]]
println(inst.show(em))
// prints: Employee(Dave, 35, 80000)
println(inst.show(ff))
// prints: Firefighter(Jim, 40, 2004)
```

```
// File PersonA.scala:
abstract class Person {
  def name: String
  def age: Int
}
case class Employee(n: String, a: Int, s: Int)
  extends Person {
  def name = n
  def age = a
}

// File PersonB.scala:
case class Firefighter(n: String, a: Int, s: Int)
  extends Person {
  def name = n
  def age = a
  def since = s
}
```

Figure 4.7 – Open class hierarchy

Note that we are using the same `Show` instance to convert both objects to strings.

Generation Concrete instances of a classtype, such as `Person` in Figure 4.7, in general have subtypes (dynamically). One approach to account for subtypes is by building the logic for all possible subtypes into the type class instance for the supertype, like is shown in Figure 4.2 in Section 4.2.3. However, such an approach does not support open class hierarchies, where new subclasses can be added in additional compilation units.

To support open class hierarchies, the generation of type class instances for open classes adds a *dispatch step*. For a class like `Person` in Figure 4.7, a dynamic dispatch is generated to select a specific type class instance based on the runtime classtype of the object that the type class is applied to (`visitee`):[4]

```
  implicit object PersonInst extends Show[Person] {
    def show(visitee: Person): String =
      visitee match {
        case v1: Employee =>
          implicitly[Show[Employee]].show(v1)
        case v2: Firefighter =>
          implicitly[Show[Firefighter]].show(v2)
```

[4] Simplified; handling of `null` values is omitted for simplicity.

```
    }
  }
```

Separate Compilation

To support subtyping polymorphism not only across different compilation units, but also across separately-compiled modules,[5] `self-assembly` provides *dynamic instance registries*. In the case of separately-compiled modules, subclasses for which we would like to generate instances are in general only discovered at link time. To be able to discover such subclasses, `self-assembly` allows registering generated instances with an *instance registry* at runtime. A reference to such an instance registry can then be shared across separately-compiled modules.

For example, module A could create a registry and populate it with a number of instances:

```
implicit val reg = new SimpleRegistry[Show]
reg.register(classOf[Employee],
             implicitly[Show[Employee]])
reg.register(classOf[Firefighter],
             implicitly[Show[Firefighter]])
...
```

Note that the registry `reg` is defined as an *implicit value;* as we explain in the following, this is required to enable registry look-ups when dispatching to type class instances based on runtime types.

With the instance registry set up in this way, another separately-compiled module B is then able to dispatch to instances registered by module A:

```
implicit val localReg = getRegistryFrom(moduleA)
localReg.register(classOf[Judge],
                  implicitly[Show[Judge]])
...
```

Importantly, when module B invokes the `show` method of an instance `instP` of type `Show[Person]`, passing an object with dynamic type `Employee`, the generated instance `instP` dispatches to the correct type class instance of type `Show[Employee]` through a look-up in registry `localReg`.

Generation To enable registry look-ups, we augment the dispatch logic with a default case:[6]

[5] The Scala ecosystem distributes modules in separate "JAR files" typically.

[6] Minimally simplified; the actual code also keeps track of object identities as discussed further below.

```
case _ => {
  val reg$1 = implicitly[Registry[Show]]
  val lookup$2: Option[Show[_]] = reg$1.get(clazz)
  lookup$2.get.asInstanceOf[Show[Person]]
              .show(visitee)
}
```

4.5.2 Object Identity

In object-oriented languages like Scala, it is important to take *object identity* into account. Simple datatypes such as case classes already permit cycles in object graphs via re-assignable fields (using the `var` modifier). It is therefore important to keep track of objects that have already been visited to avoid infinite recursion.

To enable the detection of cycles in object graphs, we keep track of all "visited" objects during the object graph traversal performed by a type class instance. However, it is not sufficient to maintain a single, global set of visited objects, since implementations of one type class might depend on other type classes; different type class instances could therefore interfere with each other when accessing the same global set (yielding nonsensical results). Thus, it is preferable to pass this set of visited objects on the call stack. With the mechanics introduced so far, this is not possible.

To enable passing an additional context (the set of visited objects) on the call stack, we require type classes to extend
`Queryable[T, R]`:

```
trait Queryable[T, R] {
  def apply(visitee: T, visited: Set[Any]): R
}
```

The `Queryable[T, R]` trait declares an `apply` method with an additional `visited` parameter (compared to the trait of the type class), which is passed the set of visited objects. This extra method allows us to distinguish between top-level invocations of type class methods and inner invocations (of `apply`). The only downside is that custom type class instances are slightly more verbose to define, although the implementation of `apply` can typically be a trivial forwarder.

For example, consider the `Show[T]` type class, now extending `Queryable[T, String]`:

```
trait Show[T] extends Queryable[T, String] {
  def show(visitee: T): String
}
```

A type class instance for integers can be implemented as follows:

```
implicit val intHasShow = new Show[Int] {
  def show(visitee: Int): String = "" + x
  def apply(visitee: Int, visited: Set[Any]) =
    show(visitee)
}
```

Note that the implementation of apply is trivial.

Generation To enable the detection of cycles in object graphs it is necessary to adapt the implementation of the implicit object as follows.

```
implicit object CShowInstance extends Show[C] {
  def show(visitee: C): String =
    apply(visitee, Set[Any]())
  def apply(visitee: C, visited: Set[Any]) =
    ...
}
```

Note that an invocation of show is treated as a *top-level invocation* forwarding to apply passing an empty set of visited objects. Crucially, when applying the type class instances for the class parameters of C, instead of invoking show directly, we invoke apply passing the visited set extended with the current object (visitee).

```
var result: String = ""
if (!visited(visitee.p_1)) {
  val inst_1 = implicitly[Show[D_1]]
  result = result +
   inst_1.apply(visitee.p_1, visited + visitee)
}
...
if (!visited(visitee.p_n)) {
  val inst_n = implicitly[Show[D_n]]
  result = result +
   inst_n.apply(visitee.p_n, visited + visitee)
}
```

4.6 Transformations

The library provides a set of traits for expressing generic functions that are either (a) queries or (b) transformations. Basically, a query generates type class instances that traverse an object

graph and return a single result of a possibly different type. In contrast, a transformation generates type class instances that perform a deep copy of an object graph, applying transformations to objects of selected types. While Sections 4.4-4.5 were focused on generic queries, this section provides an overview of generic transformations.

Example Suppose we would like to express a generic transformation, which clones object graphs, except for subobjects of a certain type, which are transformed. An example for such a transformation is a generic "scale" function that scales all integers in an object graph by a given factor. The `self-assembly` library lets us write the "scale" function in two steps: first, the definition of a suitable type class; second, the implementation of a subclass of the library-provided `Transform` class. A suitable type class is easily defined:

```
trait Scale[T] extends Queryable[T, T] {
  def scale(visitee: T): T
}
```

Note that the input and output types of `Queryable` are the same in this case, since `scale` transforms any input object into an object of the same type. The actual transformation is defined as follows:

```
object Scale extends Transform {
  def mkTrees[C <: SContext](c: C) = new Trees(c)

  class Trees[C <: SContext](override val c: C)
    extends super.Trees(c)

  implicit def generate[T]: Scale[T] =
    macro genTransform[T, this.type]
}
```

This transformation is not very interesting yet: it simply creates a deep clone of the input object. To specify how, in our case, integers are scaled, it is necessary to define a custom type class instance:

```
def intScale(factor: Int) = new Scale[Int] {
  def scale(x: Int) = x * factor
  def apply(x: Int, visited: Set[Any]) = scale(x)
}
implicit val intInst = intScale(myFactor)
```

For convenience, we can introduce a generic `gscale` function:

```
def gscale[T](obj: T)(implicit inst: Scale[T]): T =
  inst.scale(obj)
```

gscale is then invoked as follows:

```
implicit val inst = intScale(10)
val scaled = gscale(obj)
```

Transformations in self-assembly The genTransform macro is based on traversals similar to those of generic queries. However, the crucial difference is that the macro generates code to *clone* visited objects (based on techniques used in scala/pickling [Miller et al., 2013]). Interestingly, the implementations of queries and transformations share a substantial number of generic building blocks.

4.7 Generic Properties: Custom Lightweight Static Checks

In this section we show how our approach supports the definition of custom lightweight static checking, similar to pluggable type system extensions, that go beyond object-oriented DGP as discussed in the previous sections. In particular, the self-assembly library allows defining generic type-based properties that can be checked by the existing Scala type checker.

The key to support both object-oriented DGP and type properties is the fact that our approach is based on generic programming *at compile time*. In addition to having access to query and transformation facilities provided by the library, users also have (a) access to full static type information and (b) Scala's meta-programming API, enabling one to generatively define such generic type properties.

The enabled static checks are lightweight in the sense that they cannot extend the existing syntax or change Scala's existing type-checking. Instead, they can be thought of as pluggable type system extensions [Bracha, 2004] in that without changing the existing typechecker, additional properties can be checked. As a result, our approach supports added checking such as (transitive) type-based immutability checking, which goes beyond standard DGP.

In the following Section 4.7.1, we first provide a more precise definition of the supported generic properties. Section 4.7.2 presents a complete example of a non-trivial generic property, immutable types. Finally, in Section 4.7.3, we discuss key aspects of our implementation in the self-assembly library.

4.7.1 Generic Properties: Definition

The generic properties supported in self-assembly are unary type relations. Oliveira et al. [Oliveira et al., 2010] show how to define custom type relations in Scala using implicits

(see Section 4.2.1). However, unary type relations defined using implicits are incapable of expressing properties that depend on structural type information that's inaccessible through simple type bounds. Our approach builds on Oliveira et al.'s foundation, and extends it to deep structural type information using type-safe meta-programming.

In the following, we summarize the definition of type relations using implicits and present a high-level overview of our added lightweight static checks. We then show how `self-assembly` is augmented with meta-programming facilities in order to enable the definition of deeper structural properties.

Defining Unary Type Relations via Type Classes Using implicits a unary type relation can be defined in Scala using an arbitrary generic type constructor, say, `TC`. A type `T` can be declared to be an element of this relation, by defining an *implicit* of type `TC[T]`:

```
implicit val tct = new TC[T] {}
```

This way, an arbitrary *bounded* unary type relation can be defined. The membership of a type `U` in the relation `TC` can be checked by requiring evidence for it using an implicit parameter:

```
def m[U](implicit ev: TC[U]): ...
```

(Classes, and thereby constructors, can also have such implicit parameters.) Only if there exists an implicit value of type `TC[U]` can an invocation of method `m[U]` be type-checked.

Polymorphic implicit methods allow defining a certain class of unbounded type relations by returning values of type `TC[V]` for an arbitrary type `V` that satisfies given type bounds. For example, the following implicit method declares all types that are equal to or subtypes of type `Person` to be elements of relation `TC`:

```
implicit def belowPerson[S <: Person]: TC[S] =
  new TC[S] {}
```

However, without meta-programming the domain of the relation can only be restricted using type bounds; this is not enough for rich properties such as immutability since it requires deep checking to determine whether fields are re-assignable or not.

More Powerful Type Relations via Type-Safe Meta-Programming We extend the above-described type class-based approach so as to be able to define relations that take deep structural type information into account. Our approach provides the following benefits for library authors defining new type relations (such as the immutable property):

1. Library authors are provided with a safe, read-only view of the static type info corresponding to types we test for membership in the relation. The provided type information is not restricted to subtyping tests, rather, all functionality for analyzing type information is provided by Scala's meta-programming API.

2. Boilerplate for library authors is minimized using the generation approach that we outlined in Section 4.4.2. Analogous to queries and transformations, the `self-assembly` library provides a set of reusable abstractions, in turn making the generation mechanism easily accessible to library authors.

Safety Static meta-programming has a reputation for being ad-hoc, untyped, and "anything-goes." However, in our approach the use of macros is fairly restricted. First, we restrict ourselves to a type-safe subset of Scala's macro system (except for a small trusted core), and macro implementations are guaranteed to conform to their type signatures. As a result, these macros are easy to reason about and are well-behaved citizens in the tooling ecosystem. Second, and perhaps most importantly, the `self-assembly` library encapsulates all code generation capabilities internally; library authors defining new generic properties are provided with only a very restricted API. The API is limited to a read-only view of static type information and the possibility to define a predicate on this information controlling type class instance generation.

4.7.2 Example: Immutable Types

This section presents a complete example of a generic property as defined by a library author using `self-assembly`: a type property for deep immutability. The implementation of this property is shown in Figure 4.8.

The goal of the defined generic property is to traverse the full structure of a given type, and to ensure (a) that there are no re-assignable fields and (b) that all field types satisfy this property recursively. Therefore, the property is guaranteed *transitively* (all reachable objects are immutable). To guard against subclasses with re-assignable fields, the implementation assumes references of non-final class type potentially refer to mutable objects.

Elements like trait `Property` and the `genQuery` macro are provided by the library. The idea is that when the `genQuery` macro derives an instance of `Immutable[T]` it (a) creates an instance of class `Trees` at compile time, and (b) uses this to check that type T (accessible at compile time as `tpe`) does not contain re-assignable fields (`vars`) and it is possible to derive `Immutable` instances for all its fields (in turn guaranteeing that they are all deeply immutable).

The example also shows that it is possible to add custom type class instances manually (in the example, for types `Int` and `String`). In general, this means that the checks of the generic property can be overridden for specific types. While providing an escape hatch (*e.g.,* in situations where lightweight static checking is not powerful enough to prove a desired property

```
trait Immutable[T] {}

object Immutable extends Property[Unit] {
  def mkTrees[C <: Context with Singleton](c: C) =
    new Trees(c)

  class Trees[C <: Context with Singleton]
    (override val c: C) extends super.Trees(c) {
    def check(tpe: c.Type): Unit = {
      import c.universe._

      if (tpe.typeSymbol.isClass &&
          !tpe.typeSymbol.asClass.isFinal &&
          !tpe.typeSymbol.asClass.isCaseClass) {
        c.abort(c.enclosingPosition, """instances
of non-final or non-case class not
guaranteed to be immutable""")
      } else {
        // if tpe has var, abort
        val allAccessors =
          tpe.decls collect {
            case sym: MethodSymbol
              if sym.isAccessor ||
              sym.isParamAccessor => sym }
        val varGetters =
          allAccessors collect {
            case sym if sym.isGetter &&
              sym.accessed != NoSymbol &&
              sym.accessed.asTerm.isVar => sym }
        if (varGetters.nonEmpty)
          c.abort(c.enclosingPosition,
                  "not immutable")
      }
    }
  }

  implicit def generate[T]: Immutable[T] =
    macro genQuery[T, this.type]

  implicit val intIsImm: Immutable[Int] =
    new Immutable[Int] {}

  implicit val stringIsImm: Immutable[String] =
    new Immutable[String] {}
}
```

Figure 4.8 – Deep immutability checking using `self-assembly`

for some type), this capability can also be used to subvert the checking of the generic property, of course. However, existing type checking of the Scala compiler remains unaffected in all cases.

4.7.3 Generic Properties as Implemented in `self-assembly`

The `self-assembly` library implements generic properties as extensions of generic queries. Note that library authors defining new type properties are not exposed to the implementation discussed in the following.

Let us consider a sketch of `self-assembly`'s implementation of the simple generic `Property` trait used in the previous example:

```
trait Property[R] extends AcyclicQuery[R] {
  abstract class Trees[C <: SContext]
    (override val c: C) extends super.Trees(c) {
    def check(tpe: c.Type): Unit
    override def delimit(tpe: c.Type) = {
      check(tpe)
      (reify({}), reify({}), reify({}))
    }
    ...
  } }
```

The trait introduces a new abstract `check` method that must be implemented by the library author who wishes to define concrete properties such as `Immutable[T]` above. Moreover, the `delimit` method that the generic query invokes for all types encountered in a traversal is overridden to invoke the user-defined `check` method. Otherwise, `delimit` only returns trivial expression trees, since they are (essentially) unused.

4.8 Implementation and Case Study

We have implemented our approach in the `self-assembly` Scala library.[7] The library has been developed and tested using the current stable release of Scala version 2.11. No extension of the Scala language or compiler is required by the library. The library is comprised of ≈ 1,150 LOC.

Case Study: Scala Pickling To evaluate both expressivity and performance, we have ported the pickling framework presented in the previous chapter, scala/pickling [Miller et al., 2013], to `self-assembly`[8].

[7] See https://github.com/phaller/selfassembly.

[8] https://github.com/phaller/selfassembly/tree/master/src/main/scala/selfassembly/examples/pickling

scala/pickling is a popular open-source project; on the social code hosting platform GitHub, the project has more than 630 "stars". To achieve its high performance, scala/pickling leverages macros for compile-time code generation. Our port of scala/pickling to `self-assembly` supports already about 90% of the features of the original; notably, subtyping, object identity, separate compilation, and pluggable pickle formats. Currently, the port lacks picklers based on run-time reflection.

Framework	Performance Change	LOC reduction
scala/pickling	< 1%	56%

Table 4.1 – Results of porting scala/pickling to `self-assembly`

In terms of efficiency, `self-assembly` compares favorably to the original library: execution time of the "Evactor" benchmark [Miller et al., 2013] remains within 1% of scala/pickling. At the same time, the `self-assembly`-based code is significantly simpler, shorter, and more maintainable. The use of `self-assembly` reduced the code size for macro-based type class instance generation by about 56%.

4.9 Related Work

DGP in Functional Languages The idea of DGP originated in the Functional Programming community. There are several approaches for writing datatype-generic programs. Early approaches were based on programming languages with built-in support for DGP. These approaches include PolyP [Jansson and Jeuring, 1997], and Generic Haskell [Clarke and Löh, 2003]. Later approaches were based on small language extensions for general purpose languages like Haskell. Examples include Scrap Your Boilerplate [Lämmel and Peyton Jones, 2003], Template Haskell [Sheard and Peyton Jones, 2002] and Generic Clean [Alimarine and Plasmeijer, 2002].

More recently, researchers have realized that by using advanced type system features DGP could be implemented directly as libraries. Extensive surveys of various approaches to DGP in Haskell (mostly focused on libraries) document various approaches [Hinze et al., 2007, Rodriguez et al., 2008]. A large majority of these library based approaches use *run-time* type representations, as well as, isomorphisms that convert between specific datatypes and generic type representations. Without further optimizations this has a significant impact on performance. To improve performance several approaches use techniques such as partial-evaluation [Alimarine and Smetsers, 2004] or inlining [Magalhães et al., 2010]. Approaches based on partial-evaluation require language support, which makes them more difficult to adopt. Inlining is simpler to adopt since it is readily available in many compilers. Good results optimizing some generic functions have been reported in the GHC compiler. However inlining is not very predictable and some generic functions do not optimize well.

Approaches that use meta-programming techniques like Template Haskell (TH) [Adams and

DuBuisson, 2012] to do DGP are closest to our work. The use of TH is very often motivated by performance considerations, to avoid the costs of run-time type representations. However, published proposals using TH are based on its *untyped* macro system. (TH itself has recently been upgraded to allow type-safe macros.) Although type errors are still detected at compile time even using the untyped system, they are given in terms of the generated code instead of the macro code. In `self-assembly` we do not need to make such a trade-off, because we only use the type-safe subset of Scala's macros (apart from a small, internal trusted core, as is common in DGP approaches).

In contrast to `self-assembly` none of the functional DGP approaches deal with OO features like subtyping or object identity.

DGP in OO Languages Adaptive Object-Oriented Programming (AOOP) [Lieberherr, 1996] can be considered a DGP approach. In AOOP there is a domain-specific language for selecting parts of a structure that should be visited. This is useful to do traversals on complex structures and focus only on the interesting parts of the structure relevant for computing the final output. DJ is an implementation of AOOP for Java using reflection [Orleans and Lieberherr, 2001]. More recently, inspired by AOOP, DemeterF [Chadwick and Lieberherr, 2010] improved on previous approaches by providing support for safe traversals, generics and data-generic function generation. Compared to `self-assembly` most AOOP approaches are not type-safe. Only in DemeterF a custom type system was designed to ensure type-safety of generic functions. However DemeterF requires a new language and it is unclear wether issues like object identity are considered, since they take a more functional approach than other AOOP approaches. DemeterF is a language approach to DGP (much like Generic Haskell, for example); whereas we view `self-assembly` as a library based approach.

There has also been some work porting existing functional DGP approaches to Scala. Moors et al. [Moors et al., 2006] did a port of "origami"-based DGP [Gibbons, 2006]. Oliveira and Gibbons [Oliveira and Gibbons, 2010] picked up on this line of work and have shown how several other DGP approaches can be ported and improved in Scala. In particular they have shown some approaches that for doing DGP with type classes, which has a similar flavour to `self-assembly`. However none of these ports attempt to deal with OO features like subtyping or object identity. Moreover all approaches are based on run-time type representations, which is in contrast to our compile-time approach.

Pluggable Type Systems and Language Extensions There are several approaches for providing pluggable type system extensions for statically-typed OO languages [Chin et al., 2005, Dietl et al., 2011, Papi et al., 2008], but unlike `self-assembly`, they do not provide DGP capabilities. Furthermore, `self-assembly` provides lightweight added type checks, which can cannot extend program syntax (like, *e.g.,* SugarJ [Erdweg et al., 2011]) or change Scala's built-in type checking.

Our approach is in some sense complementary to staging for embedded DSLs (*e.g.,* , LMS [Rompf and Odersky, 2012]): however, rather than providing staged expressions that are type-checked by the host language, we piggy-back on a macro system for the definition of new type relations. Implicit macros generate type class instances, which, in turn, refine type-checking of *unstaged* programs in the host language. Furthermore, `self-assembly` doesn't require any extensions to the host language.

4.10 Conclusion

This chapter detailed a general mechanism, called `self-assembly`, for defining generic operations or properties that operate over a large class of types with little boilerplate and good performance, and for defining additional lightweight static typechecking via generic properties This mechanism has the extensibility and customization advantages of type classes; and it has the automatic implementation advantages of mechanisms like Java's serialization mechanism. The key idea is to provide automatic implementations of type classes using type-safe macros. This allows programmers to define their own generic functionality, such as serialization, pretty printing, or equality; and it also allows the definition of generic properties such as immutability checking. To demonstrate the usefulness of `self-assembly` in practice, we implemented an industry-ready serialization framework for Scala.

5 Spores

In Chapter 3, we covered *object oriented picklers* and scala/pickling, a framework for automatically generating them. Throughout the presentation of scala/pickling, it was noted that serializing function closures, a first-class language construct in Scala, was beyond its capabilities. In this chapter, we see why serializing function closures is nontrivial, and introduce *spores*, an abstraction which enables closures to be statically analyzed and serialized.

5.1 Introduction

With the rise of big data analytics, and our ongoing migration to mobile applications and "the cloud", distributed programming has entered the mainstream. Popular paradigms in software engineering such as software as a service (SaaS), RESTful services, or the rise of a multitude of systems for big data processing and interactive analytics evidence this trend.

At the same time, functional programming has also been gaining traction, as is evidenced by the ongoing trend of traditionally object-oriented or imperative languages being extended with functional features, such as lambdas in Java 8 [Goetz, 2013], C++11 [International Standard ISO/IEC 14882:2011, 2011], and Visual Basic 9 [Meijer, 2007], the perceived importance of functional programming in general empirical studies on software developers [Meyerovich and Rabkin, 2013], and the popularity of functional programming massively online open courses (MOOCs) [Miller et al., 2014b].

One reason for the rise in popularity of functional programming languages and features within object-oriented communities is the basic philosophy of transforming immutable data by applying first-class functions, and the observation that this functional style simplifies reasoning about data in parallel, concurrent, and distributed code. A popular and well-understood example of this style of programming for which many popular frameworks have come to fruition is functional data-parallel programming (FDP). Examples of FDP across functional and object-oriented paradigms include Java 8's monadic-style optionally parallel collections [Goetz, 2013], Scala's parallel [Prokopec et al., 2011] and concurrent dataflow [Prokopec et al., 2012a] collec-

tions, Data Parallel Haskell [Chakravarty et al., 2007], CnC [Budimlić et al., 2010], Nova [Collins et al., 2013], and Haskell's `Par` monad [Marlow et al., 2011] to name a few.

In the context of distributed programming, data-parallel frameworks like MapReduce [Dean and Ghemawat, 2008] and Spark [Zaharia et al., 2012] are designed around functional patterns where closures are transmitted across cluster nodes to large-scale persistent datasets. As a result of the "big data" revolution, these frameworks have become very popular, in turn further highlighting the need to be able to reliably and safely serialize and transmit closures over the network.

However, there's trouble in paradise. For both object-oriented and functional languages, there still exist numerous hurdles at the language-level for even these most basic functional building blocks, closures, to overcome in order to be reliable and easy to reason about in a concurrent or distributed setting.

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:

```
sendFunc :: SendPort (Int -> Int) -> Int -> ProcessM ()
sendFunc p x = sendChan p (\y -> x + y + 1)
```

In this example, in function `sendFunc` we are sending the lambda `(\y -> x + y + 1)` on channel `p`. The lambda captures variable `x`, a parameter of `sendFunc`. Serializing the lambda requires serializing also its captured variables. However, when looking up a serializer for the lambda, only the type of the lambda is taken into account; however, it doesn't tell us anything about the types of its captured variables, which makes it impossible in Haskell to look up serializers for them.

In object-oriented languages like Java or C#, serialization is solved differently – the runtime environment is designed to be able to serialize any object, reflectively. While this "universal" serialization might seem to solve the problem of languages like Haskell that cannot rely on such a mechanism, serializing closures nonetheless remains surprisingly error-prone. For example, attempting to serialize a closure with transitive references to objects that are not marked as serializable will crash at runtime, typically with no compile-time checks whatsoever. The kicker is that it is remarkably easy to accidentally and unknowingly create such a problematic transitive reference, especially in an object-oriented language.

For example, consider the following use of a distributed collection in Scala with higher-order functions `map` and `reduce` (using Spark):

```
class MyCoolRddApp {
  val log = new Log(...)
  def shift(p: Int): Int = ...
  ...
  def work(rdd: RDD[Int]) {
    rdd.map(x => x + shift(x)).reduce(...)
  }
}
```

In this example, the closure `(x => x + shift(x))` is passed to the `map` method of the distributed collection `rdd` which requires serializing the closure (as, in Spark, parts of the data structure reside on different machines). However, calling `shift` inside the closure invokes a method on the enclosing object `this`. Thus, the closure is capturing, and must therefore serialize, `this`. If `Log`, a field of `this`, is not serializable, this will fail at runtime.

In fact, closures suffer not only from the problems shown in these two examples; there are numerous more hazards that manifest *across programming paradigms*. To provide a glimpse, closure-related hazards related to concurrency and distribution include:

- accidental capture of non-serializable variables (including `this`);
- language-specific compilation schemes, creating implicit references to objects that are not serializable;
- transitive references that inadvertently hold on to excessively large object graphs, creating memory leaks;
- capturing references to mutable objects, leading to race conditions in a concurrent setting;
- unknowingly accessing object members that are not constant such as methods, which in a distributed setting can have logically different meanings on different machines.

Given all of these issues, exposing functions in public APIs is a source of headaches for authors of concurrent or distributed frameworks. Framework users who stumble across any of these issues are put in a position where it's unclear whether or not the encountered issue is a problem on the side of the user or the framework, thus often adversely hitting the perceived reliability of these frameworks and libraries.

We argue that solving these problems in a principled way could lead to more confidence on behalf of library authors in exposing functions in APIs, thus leading to a potentially wide array of new frameworks.

This chapter takes a step towards more principled *function-passing style* by introducing a type-based foundation for closures, called *spores*. Spores are a closure-like abstraction and type

system which is designed to avoid typical hazards of closures. By including type information of captured variables in the type of a spore, we enable the expression of type-based constraints for captured variables, making spores safer to use in a concurrent or distributed setting. We show that this approach can be made practical by automatically synthesizing refinement types using macros, and by leveraging local type inference. Using type-based constraints, spores allow expressing a variety of "safe" closures.

To express safe closures with transitive properties such as guaranteed serializability, or closures capturing only deeply immutable types, spores support type constraints based on type classes which enforce transitive properties. In addition, implicit macros in Scala enable integration with type systems that enforce transitive properties using generics or annotated types. Spores also support user-defined type constraints. Finally, we argue that by principle of a type-based approach, spores can potentially benefit from optimization, further safety via type system extensions, and verification opportunities.

5.1.1 Design Constraints

The design of spores is guided by the following principles:

- **Type-safety.** Spores should be able to express type-based properties of captured variables in a statically safe way. Including type information of captured variables in the type of a spore creates a number of previously impossible opportunities; it facilitates the verification of closure-heavy code; it opens up the possibility for IDEs to assist in safe closure creation, advanced refactoring, and debugging support; it enables compilers to implement safe transformations that can further simplify the use of safe closures, and it makes it possible for spores to integrate with type class-based frameworks like scala/pickling [Miller et al., 2013].

- **Extensibility.** Given types which include information about what a closure captures, libraries and frameworks should be able to restrict the types that are captured by spores. Enforcing these *type constraints* should not be limited to serializability, thread-safety, or other pre-defined properties, however; spores should enable customizing the semantics of variable capture based on user-defined types. It should be possible to use existing type-based mechanisms to express a variety of user-defined properties of captured types.

- **Ease of Use.** Spores should be lightweight to use, and be able to integrate seamlessly with existing practice. It should be possible to capitalize on the benefits of precise types while at the same time ensuring that working with spores is never too verbose, thanks to the help of automatic type synthesis and inference. At the same time, frameworks like Spark, for which the need for controlled capture is central, should be able to use spores, meanwhile requiring only minimal changes in application code.

- **Practicality.** Spores should be practical to use in general, as well as be practical for inclusion in the full-featured Scala language. They should be practical in a variety of real-world scenarios (for use with Spark, Akka, parallel collections, and other closure-heavy code). At the same time, to enable a robust integration with the host language, existing type system features should be reused instead of extended.
- **Reliability for API Designers.** Spores should enable library authors to confidently release libraries that expose functions in user-facing APIs without concern of runtime exceptions or other dubious errors falling on their users.

5.1.2 Contributions

This chapter outlines the following contributions:

- We introduce a closure-like abstraction and type system, called "spores," which avoids typical hazards when using closures in a concurrent or distributed setting through controlled variable capture and customizable user-defined constraints for captured types.
- We introduce an approach for type-based constraints that can be combined with existing type systems to express a variety of properties from the literature, including, but not limited to, serializability and thread-safety/immutability. Transitive properties can be lifted to spore types in a variety of ways, *e.g.*, using type classes.
- We present a formalization of spores with type constraints and prove soundness of the type system.
- We present an implementation of spores in and for the full Scala language.[1]
- We (a) demonstrate the practicality of spores through a small empirical study using a collection of real-world Scala programs, and (b) show the power of the guarantees spores provide through case studies using parallel and distributed frameworks.

5.2 Spores

Spores are a closure-like abstraction and type system which aims to give users a principled way of controlling the environment which a closure can capture. This is achieved by (a) enforcing a specific syntactic shape which dictates how the environment of a spore is declared, and (b) providing additional type-checking to ensure that types being captured have certain properties. A crucial insight of spores is that, by including type information of captured variables in the type of a spore, type-based constraints for captured variables can be composed and checked,

[1] https://github.com/scala/spores

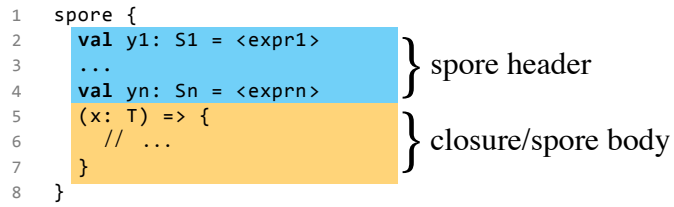


Figure 5.1 – The syntactic shape of a spore.

making spores safer to use in a concurrent, distributed, or in an arbitrary settings where closures must be controlled.

Below, we describe the syntactic shape of spores, and in Section 5.2.2 we describe the `Spore` type. In Section 5.2.4 we informally describe the type system, and how to add user-defined constraints to customize what types a spore can capture.

5.2.1 Spore Syntax

A spore is a closure with a specific shape that dictates how the environment of a spore is declared. The shape of a spore is shown in Figure 5.1. A spore consists of two parts:

- **the spore header**, composed of a list of value definitions.
- **the spore body** (sometimes referred to as the "spore closure"), a regular closure.

The characteristic property of a spore is that the *spore body* is only allowed to access its parameter, the values in the spore header, as well as top-level singleton objects (public, global state). In particular, the spore closure is not allowed to capture variables in the environment. Only an expression on the right-hand side of a value definition in the spore header is allowed to capture variables.

By enforcing this shape, the environment of a spore is always declared explicitly in the spore header, which avoids accidentally capturing problematic references. Moreover, importantly for object-oriented languages, it's no longer possible to accidentally capture the `this` reference.

Evaluation Semantics

The evaluation semantics of a spore is equivalent to a closure obtained by leaving out the `spore` marker, as shown in Figure 5.2. In Scala, the block shown in Figure 5.2a first initializes all value definitions in order and then evaluates to a closure that captures the introduced local variables `y1, ..., yn`. The corresponding spore, shown in Figure 5.2b has the exact same evaluation semantics. Interestingly, this closure shape is already used in production systems such as Spark in an effort to avoid problems with accidentally captured references,

```
{
  val y1: S1 = <expr1>
  ...
  val yn: Sn = <exprn>
  (x: T) => {
    // ...
  }
}
```

(a) A closure block.

```
spore {
  val y1: S1 = <expr1>
  ...
  val yn: Sn = <exprn>
  (x: T) => {
    // ...
  }
}
```

(b) A spore.

Figure 5.2 – The evaluation semantics of a spore is equivalent to that of a closure, obtained by simply leaving out the spore marker.

```
trait Function1[-A, +B] {
  def apply(x: A): B
}
```

(a) Scala's arity-1 function type.

```
trait Spore[-A, +B]
extends Function1[A, B] {
  type Captured
  type Excluded
}
```

(b) The arity-1 `Spore` type.

Figure 5.3 – The `Spore` type.

such as `this`. However, in systems like Spark, the above shape is merely a convention that is not enforced.

5.2.2 The Spore Type

Figure 5.3 shows Scala's arity-1 function type and the arity-1 spore type. Functions are

contravariant in their argument type `A` (indicated using `-`) and covariant in their result type `B` (indicated using `+`). The `apply` method of `Function1` is abstract; a concrete implementation applies the body of the function that is being defined to the parameter `x`.

Individual spores have *refinement types* of the base `Spore` type, which, to be compatible with normal Scala functions, is itself a subtype of `Function1`. Like functions, spores are contravariant in their argument type `A`, and covariant in their result type `B`. Unlike a normal function, however, the `Spore` type additionally contains information about *captured* and *excluded* types. This information is represented as (potentially abstract) `Captured` and `Excluded` type members. In a concrete spore, the `Captured` type is defined to be a tuple with the types of all captured variables. Section 5.2.4 introduces the `Excluded` type member.

```
val s = spore {
  val y1: String = expr1;
  val y2: Int = expr2;
  (x: Int) => y1 + y2 + x
}
```

(a) A spore `s` which captures a `String` and an `Int` in its spore header.

```
Spore[Int, String] {
  type Captured = (String, Int)
}
```

(b) `s`'s corresponding type.

Figure 5.4 – An example of the `Captured` type member.
Note: we omit the `Excluded` *type member for simplicity; we detail it later in Section 5.2.4.*

5.2.3 Basic Usage

Definition

A spore can be defined as shown in Figure 5.4a, with its corresponding type shown in Figure 5.4b. As can be seen, the types of the environment listed in the spore header are represented by the `Captured` type member in the spore's type.

Using Spores in APIs

Consider the following method definition:

```
def sendOverWire(s: Spore[Int, Int]): Unit = ...
```

In this example, the `Captured` (and `Excluded`) type member is not specified, meaning it is left abstract. In this case, so long as the spore's parameter and result types match, a spore type is always compatible, regardless of which types are captured.

Using spores in this way enables libraries to enforce the use of spores instead of plain closures, thereby reducing the risk for common programming errors (see Section 5.6 for detailed case studies), even in this very simple form. Later sections show more advanced ways in which library authors can control the capturing semantics of spores.

Composition

Like normal functions, spores can be composed. By representing the environment of spores using refinement types, it is possible to preserve the captured type information (and later, constraints) of spores when they are composed.

For example, assume we are given two spores `s1` and `s2` with types:

```
s1: Spore[Int, String] { type Captured = (String, Int) }
s2: Spore[String, Int] { type Captured = Nothing }
```

The fact that the `Captured` type in `s2` is defined to be `Nothing` means that the spore does not capture anything (`Nothing` is Scala's bottom type). The composition of `s1` and `s2`, written `s1 compose s2`, would therefore have the following refinement type:

```
Spore[String, String] { type Captured = (String, Int) }
```

Note that the `Captured` type member of the result spore is equal to the `Captured` type of `s1`, since it is guaranteed that the result spore does not capture more than what `s1` already captures. Thus, not only are spores composable, but so are their (refinement) types.

Implicitly Converting Functions to Spores

The design of spores was guided in part by a desire to make them easy to use, and easy to integrate in already closure-heavy code. Spores, as so far proposed, introduce considerable verbosity in pursuit of the requirement to explicitly define the spore's environment.

Therefore, it is also possible to use function literals as spores if they satisfy the spore shape constraints. To support this, an implicit conversion[2] macro[3] is provided which converts regular functions to spores, but only if the converted function is a literal: only then is it possible to enforce the spore shape.

For-Comprehensions

Converting functions to spores opens up the use of spores in a number of other situations; most prominently, for-comprehensions (Scala's version of Haskell's do-notation) in Scala are desugared to invocations of the higher-order `map`, `flatMap`, and `filter` methods, each of which take normal functions as arguments.[4]

In situations where for-comprehension closures capture variables, preventing them from being converted implicitly to spores, we introduce an alternative syntax for capturing variables in spores: an object that is referred to using a so-called "stable identifier" `id` can additionally be captured using the syntax `capture(id)`.[5]

This enables the use of spores in for-comprehensions, since it's possible to write:

[2] In Scala, implicit conversions can be thought of as methods which can be implicitly invoked based upon their type, and whether or not they are present in implicit scope.

[3] In Scala, macros are methods that are transparently loaded by the compiler and executed (or expanded) during compilation. A macro is defined like a normal method, but it is linked using the `macro` keyword to an additional method that operates on abstract syntax trees.

[4] For-comprehensions are desugared before implicit conversions are inserted; thus, no change to the Scala compiler is necessary.

[5] In Scala, a stable identifier is basically a selection p.x where p is a path and x is an identifier (see Scala Language Specification [Odersky, 2013], Section 3.1).

```
for (a <- gen1; b <- capture(gen2)) yield capture(a) + b
```

Note that superfluous `capture` expressions are not harmful. Thus, it is legal to write:

```
for (a <- capture(gen1); b <- capture(gen2)) yield capture(a) + capture(b)
```

This allows the use of `capture` in a way that does not require users to know how for-comprehensions are desugared. In Section 5.6 we show how capture and the implicit conversion of functions to spores enables the use of for-comprehensions in the context of distributed programming with spores.

5.2.4 Advanced Usage and Type Constraints

In this section, we describe two different kinds of "type constraints" which enable more fine-grained control over closure capture semantics; *excluded types* which prevent certain types from being captured, and *context bounds* for captured types which enforce certain type-based properties for all captured variables of a spore. Importantly, all of these different kinds of constraints compose, as we will see in later subsections.

Throughout this chapter, we use as a motivating example hazards that arise in concurrent or distributed settings. However, note that the system of type constraints described henceforth is general, and can be applied to very different applications and sets of types.

Excluded Types

Libraries and frameworks for concurrent and distributed programming, such as Akka [Typesafe, 2009] and Spark, typically have requirements to avoid capturing certain types in closures that are used together with library-provided objects and methods. For example, when using Akka, one should not capture variables of type `Actor`; in Spark, one should not capture variables of type `SparkContext`.

Such restrictions can be expressed in our system by excluding types from being captured by spores, using refinements of the `Spore` type presented in Section 5.2.2. For example, the following refinement type forbids capturing variables of type `Actor`:

```
type SporeNoActor[-A, +B] = Spore[A, B] {
  type Excluded <: No[Actor]
}
```

Note the use of the auxiliary type constructor `No` (defined as `trait No[-T]`): it enables the exclusion of multiple types while supporting desired sub-typing relationships.

For example, exclusion of multiple types can be expressed as follows:

```
type SafeSpore = Spore[Int, String] {
  type Excluded = No[Actor] with No[Util]
}
```

Given Scala's sub-typing rules for refinement types, a spore refinement excluding a superset of types excluded by an "otherwise type-compatible" spore is a subtype. For example, `SafeSpore` is a subtype of `SporeNoActor[Int, String]`.

Subtyping Using some frameworks typically user-defined subclasses are created that extend framework-provided types. However, the extended types are sometimes not safe to be captured. For example, in Akka, user-created closures should not capture variables of type `Actor` and any subtypes thereof. To express such a constraint in our system we define the `No` type constructor to be contravariant in its type parameter; this is the meaning of the `-` annotation in the type declaration `trait No[-T]`.

As a result, the following refinement type is a supertype of type `SporeNoActor[Int, Int]` defined above (we assume `MyActor` is a subclass of `Actor`):

```
type MySpore = Spore[Int, Int] {
  type Excluded <: No[MyActor]
}
```

It is important that `MySpore` is a supertype and not a subtype of `SporeNoActor[Int, Int]`, since an instance of `MySpore` could capture some other subclass of `Actor` which is not itself a subclass of `MyActor`. Thus, it would not be safe to use an instance of `MySpore` where an instance of `SporeNoActor[Int, Int]` is required. On the other hand, an instance of `SporeNoActor[Int, Int]` is safe to use in place of an instance of `MySpore`, since it is guaranteed not to capture `Actor` or any of its subclasses.

Reducing `Excluded` ***Boilerplate*** Given that the design of spores was guided in part by a desire to make them easy to use, and easy to integrate in already closure-heavy code with minimal changes, one might observe that the `Spore` type with `Excluded` types introduces considerable verbosity. This is easily solved in practice by the addition of a macro `without[T]` which takes a type parameter `T` and rewrites the spore type to take into consideration the excluded type `T`. Thus, in the case of the `SafeSpore` example, the same spore refinement type can easily be synthesized inline in the definition of a spore value:

```
val safeSpore = spore {
  val a = ...
  val b = ...
  (x: T) => { ... }
}.without[Actor].without[Util]
```

Context Bounds for Captured Types

The fact that for spores a certain shape is enforced is very useful. However, in some situations this is not enough. For example, a common source of race conditions in data-parallel frameworks manifests itself when users capture mutable objects. Thus, a user might want to enforce that closures only capture immutable objects. However, such constraints cannot be enforced using the spore shape alone (captured objects are stored in constant values in the spore header, but such constants might still refer to mutable objects).

In this section, we introduce a form of type-based constraints called "context bounds" which enforce certain type-based properties for all captured variables of that spore.[6]

Taking another example, it might be necessary for a spore to require the availability of instances of a certain type class for the types of all of its captured variables. A typical example for such a type class is `Pickler`: types with an instance of the `Pickler` type class can be pickled using a new type-based pickling framework for Scala [Miller et al., 2013]. To be able to pickle a spore, it's necessary that all its captured types have an instance of `Pickler`.[7]

Spores allow expressing such a requirement using a notion of implicit *properties*. The idea is that if there is an implicit value[8] of type `Property[Pickler]` in scope at the point where a spore is created, then it is enforced that all captured types in the spore header have an instance of the `Pickler` type class:

```
import spores.withPickler

spore {
  val name: String = <expr1>
  val age: Int = <expr2>
  (x: String) => { ...}
}
```

While an imported property does not have an impact on how a spore is constructed (besides

[6] The name "context bound" is used in Scala to refer to a particular kind of implicit parameter that is added automatically if a type parameter has declared such a context bound. Our proposal essentially adds context bounds to type members.

[7] A spore can be pickled by pickling its environment and the fully-qualified class name of its corresponding function class.

[8] An implicit value is a value in *implicit scope* that is statically selected based on its type.

the property import), it has an impact on the result type of the spore macro. In the above example, the result type would be a refinement of the `Spore` type:[9]

```
Spore[String, Int] {
  type Captured = (String, Int)
  implicit val ev$0 = implicitly[Pickler[Captured]]
}
```

For each property that is imported, the resulting spore refinement type contains an implicit value with the corresponding type class instance for type `Captured`.

Expressing context bounds in APIs Using the above types and implicits, it's also possible for a method to require argument spores to have certain context bounds. For example, requiring argument spores to have picklers defined for their captured types can be achieved as follows:

```
def m[A, B](s: Spore[A, B])(implicit p: Pickler[s.Captured]) = ...
```

Defining Custom Properties

Properties can be introduced using the
`Property` trait (provided by the spores library): `trait Property[C[_]]`

As a running example, we will be defining a custom property for immutable types. A custom property can be introduced using a generic trait, and an implicit "property" object that mixes in the above `Property` trait:

```
object safe {
  trait Immutable[T]
  implicit object immutableProp extends Property[Immutable]
  ...
}
```

The next step is to mark selected types as immutable by defining an implicit object extending the desired list of types, each type wrapped in the `Immutable` type constructor:

[9] In the code example, `implicitly[T]` returns the uniquely-defined implicit value of `T` which is in scope at the invocation site.

```
object safe {
  ...
  import scala.collection.immutable.{Map, Set, Seq}
  implicit object collections extends Immutable[Map[_, _]] with
    Immutable[Set[_]] with Immutable[Seq[_]] with ...
}
```

The above definitions allow us to create spores that are guaranteed to capture only types `T` for which an implicit of type `Immutable[T]` exists.

It's also possible to define compound properties by mixing in multiple traits into an implicit property object:

```
implicit object myProps extends Property[Pickler] with Property[Immutable]
```

By making this compound property available in a scope within which spores are created (for example, using an `import`), it is enforced that those spores have both the context bound `Pickler` and the context bound `Immutable`.

Composition

Now that we've introduced type constraints in the form of excluded types and context bounds, we present generalized composition rules for the types of spores with such constraints.

To precisely describe the composition rules, we introduce the following notation: the function *Excluded* returns, for a given refinement type, the set of types that are excluded; the function *Captured* returns, for a given refinement type, the list of types that are captured. Using these two mathematical functions, we can precisely specify how the type members of the resulting spore refinement type are computed. (We use the syntax *.type* to refer to the singleton types of the argument spores and the result, respectively.)

1. $Captured(res.type) = Captured(s1.type), Captured(s2.type)$
2. $Excluded(res.type) = \{\, T \in Excluded(s1.type) \cup Excluded(s2.type) \mid T \notin Captured(s1.type), Captured(s2.type) \,\}$

The first rule expresses the fact that the sequence of captured types of the resulting refinement type is simply the concatenation of the captured types of the argument spores. The second rule expresses the fact that the set of excluded types of the result refinement type is defined as the set of all types that are excluded by one of the argument spores, but that are not captured by any of the argument spores.

For example, assume two spores `s1` and `s2` with types:

```
Spore[Int, String] {
  type Captured = (Int, Util)
  type Excluded = No[Actor]
}
```

(a) Type of spore `s1`.

```
Spore[String, Int] {
  type Captured = (String, Int)
  type Excluded = No[Actor] with No[Util]
}
```

(b) Type of spore `s2`.

The result of composing the two spores, `s1 compose s2`, thus has the following type:

```
Spore[String, String] {
  type Captured = (Int, Util, String, Int)
  type Excluded = No[Actor]
}
```

Loosening constraints Given that type constraints compose, it's evident that as spores compose, type constraints can monotonically increase in number. Thus, it's important to note that it's also possible to soundly loosen constraints using regular type widening.

Let's say we have a spore with the following (too elaborate) refinement type:

```
val s2: Spore[String, Int] {
  type Captured = (String, Int)
  type Excluded = No[Actor] with No[Util]
}
```

Then we can soundly drop constraints by using a supertype such as `MySafeSpore`:

```
type MySafeSpore = Spore[String, Int] {
  type Captured
  type Excluded <: No[Actor]
}
```

5.2.5 Transitive Properties

Transitive properties like picklability or immutability are not enforced through the spores type system. Rather, spores were designed for extensibility; we ensure that deep checking can be applied to spores as follows.

An initial motivation was to be able to require type class instances for captured types, *e.g.,* picklability; spores integrate seamlessly with scala/pickling [Miller et al., 2013].

Transitive properties expressed using known techniques, *e.g.*, generics (Zibin etal's OIGJ system [Zibin et al., 2010] for transitive immutability) or annotated types, can be enforced for captured types using custom spore properties. Instead of merely tagging types, implicit macros can generate type class instances for all types satisfying a predicate. For example, using OIGJ we can define an implicit macro:

```
implicit def isImmutable[T: TypeTag]: Immutable[T]
```

which returns a type class instance for all types of the shape `C[0, Immut]` that is deeply immutable (analyzing the `TypeTag`). Custom spore properties requiring type classes constructed in such a way enable transitive checking for a variety of such (pluggable) extensions, including compositions thereof (*e.g.*, picklability/immutability).

5.3 Formalization

$t ::= x$	variable
$\mid (x : T) \Rightarrow t$	abstraction
$\mid t\ t$	application
$\mid \texttt{let}\ x = t\ \texttt{in}\ t$	let binding
$\mid \{\overline{l = t}\}$	record construction
$\mid t.l$	selection
$\mid \texttt{spore}\ \{\ \overline{x : T = t}\ ; \overline{pn}; (x : T) \Rightarrow t\ \}$	spore
$\mid \texttt{import}\ pn\ \texttt{in}\ t$	property import
$\mid t\ \texttt{compose}\ t$	spore composition
$v ::= (x : T) \Rightarrow t$	abstraction
$\mid \{\overline{l = v}\}$	record value
$\mid \texttt{spore}\ \{\ \overline{x : T = v}\ ; \overline{pn}; (x : T) \Rightarrow t\ \}$	spore value
$T ::= T \Rightarrow T$	function type
$\mid \{\overline{l : T}\}$	record type
$\mid \mathscr{S}$	
$\mathscr{S} ::= T \Rightarrow T\ \{\ \texttt{type}\ \mathscr{C} = \overline{T}\ ;\ \overline{pn}\ \}$	spore type
$\mid T \Rightarrow T\ \{\ \texttt{type}\ \mathscr{C}\ ;\ \overline{pn}\ \}$	abstract spore type
$P \in pn \to \mathscr{T}$	property map
$\mathscr{T} \in \mathscr{P}(T)$	type family
$\Gamma ::= \overline{x : T}$	type environment
$\Delta ::= \overline{pn}$	property environment

Figure 5.6 – Core language syntax

We formalize spores in the context of a standard, typed lambda calculus with records. Apart from novel language and type-systematic features, our formal development follows a well-

known methodology [Pierce, 2002]. Figure A.2 shows the syntax of our core language. Terms are standard except for the `spore`, `import`, and `compose` terms. A `spore` term creates a new spore. It contains a list of variable definitions (the spore header), a list of property names, and the spore's closure. A property name refers to a type family (a set of types) that all captured types must belong to.

An illustrative example of a property and its associated type family is a type class: a spore satisfies such a property if there is a type class instance for all its captured types.

An `import` term imports a property name into the property environment within a lexical scope (a term); the property environment contains properties that are registered as requirements whenever a spore is created. This is explained in more detail in Section 5.3.2. A `compose` term is used to compose two spores. The core language provides spore composition as a built-in feature, because type checking spore composition is markedly different from type checking regular function composition (see Section 5.3.2).

The grammar of values is standard except for spore values; in a spore value each term on the right-hand side of a definition in the spore header is a value.

The grammar of types is standard except for spore types. Spore types are refinements of function types. They additionally contain a (possibly-empty) sequence of captured types, which can be left abstract, and a sequence of property names.

5.3.1 Subtyping

Figure 5.7 shows the subtyping rules; rules S-REC and S-FUN are standard [Pierce, 2002].

The subtyping rule for spores (S-SPORE) is analogous to the subtyping rule for functions with respect to the argument and result types. Additionally, for two spore types to be in a subtyping relationship either their captured types have to be the same ($M_1 = M_2$) or the supertype must be an abstract spore type ($M_2 = \texttt{type}\ \mathscr{C}$). The subtype must guarantee at least the properties of its supertype, or a superset thereof. Taken together, this rule expresses the fact that a spore type whose type member $\mathscr{C}$ is not abstract is compatible with an abstract spore type as long as it has a superset of the supertype's properties. This is important for spores used as first-class values: functions operating on spores with arbitrary environments can simply demand an abstract spore type. The way both the captured types and the properties are modeled corresponds to (but simplifies) the subtyping rule for refinement types in Scala (see Section 5.2.4).

Rule S-SPOREFUN expresses the fact that spore types are refinements of their corresponding function types, giving rise to a subtyping relationship.

$$\textsc{S-Rec}\ \frac{\overline{l'} \subseteq \overline{l} \qquad l_i = l'_i \to T_i <: T'_i \wedge T'_i <: T_i}{\{\overline{l : T}\} <: \{\overline{l' : T'}\}} \qquad \textsc{S-Fun}\ \frac{T_2 <: T_1 \qquad R_1 <: R_2}{T_1 \Rightarrow R_1 <: T_2 \Rightarrow R_2}$$

$$\textsc{S-Spore}\ \frac{T_2 <: T_1 \qquad R_1 <: R_2 \qquad \overline{pn'} \subseteq \overline{pn} \qquad M_1 = M_2 \vee M_2 = \texttt{type}\ \mathscr{C}}{T_1 \Rightarrow R_1\ \{\ M_1\ ;\ \overline{pn}\ \} <: T_2 \Rightarrow R_2\ \{\ M_2\ ;\ \overline{pn'}\ \}}$$

$$\textsc{S-SporeFun}\ \frac{}{T_1 \Rightarrow R_1\ \{\ M\ ;\ \overline{pn}\ \} <: T_1 \Rightarrow R_1}$$

Figure 5.7 – Subtyping

$$\textsc{T-Var}\ \frac{x : T \in \Gamma}{\Gamma;\Delta \vdash x : T} \qquad \textsc{T-Sub}\ \frac{\Gamma;\Delta \vdash t : T' \quad T' <: T}{\Gamma;\Delta \vdash t : T} \qquad \textsc{T-Abs}\ \frac{\Gamma, x : T_1;\Delta \vdash t : T_2}{\Gamma;\Delta \vdash (x : T_1) \Rightarrow t : T_1 \Rightarrow T_2}$$

$$\textsc{T-App}\ \frac{\Gamma;\Delta \vdash t_1 : T_1 \Rightarrow T_2 \quad \Gamma;\Delta \vdash t_2 : T_1}{\Gamma;\Delta \vdash (t_1\ t_2) : T_2} \qquad \textsc{T-Let}\ \frac{\Gamma;\Delta \vdash t_1 : T_1 \quad \Gamma, x : T_1;\Delta \vdash t_2 : T_2}{\Gamma;\Delta \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : T_2}$$

$$\textsc{T-Rec}\ \frac{\Gamma;\Delta \vdash \overline{t : T}}{\Gamma;\Delta \vdash \{\overline{l = t}\} : \{\overline{l : T}\}} \qquad \textsc{T-Sel}\ \frac{\Gamma;\Delta \vdash t : \{\overline{l : T}\}}{\Gamma;\Delta \vdash t.l_i : T_i} \qquad \textsc{T-Imp}\ \frac{\Gamma;\Delta, pn \vdash t : T}{\Gamma;\Delta \vdash \texttt{import}\ pn\ \texttt{in}\ t : T}$$

$$\textsc{T-Spore}\ \frac{\forall s_i \in \overline{s}.\ \Gamma;\Delta \vdash s_i : S_i \qquad \overline{y : S}, x : T_1;\Delta \vdash t_2 : T_2 \qquad \forall pn \in \Delta, \Delta'.\ \overline{S} \subseteq P(pn)}{\Gamma;\Delta \vdash \texttt{spore}\ \{\ \overline{y : S = s}\ ;\Delta';(x : T_1) \Rightarrow t_2\ \} :\quad T_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}\ ;\ \Delta, \Delta'\ \}}$$

$$\textsc{T-Comp}\ \frac{\begin{array}{c}\Gamma;\Delta \vdash t_1 : T_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}\ ;\ \Delta_1\ \} \\ \Gamma;\Delta \vdash t_2 : U_1 \Rightarrow T_1\ \{\ \texttt{type}\ \mathscr{C} = \overline{R}\ ;\ \Delta_2\ \} \qquad \Delta' = \{pn \in \Delta_1 \cup \Delta_2 \mid \overline{S} \subseteq P(pn) \wedge \overline{R} \subseteq P(pn)\}\end{array}}{\Gamma;\Delta \vdash t_1\ \texttt{compose}\ t_2 : U_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}, \overline{R}\ ;\ \Delta'\ \}}$$

Figure 5.8 – Typing rules

5.3.2 Typing rules

Typing derivations use a judgement of the form $\Gamma;\Delta \vdash t : T$. Besides the standard variable environment Γ we use a property environment Δ which is a sequence of property names that have been imported using `import` expressions in enclosing scopes of term t. The property environment is reminiscent of the implicit parameter context used in the original work on implicit parameters [Lewis et al., 2000]; it is an environment for names whose definition sites "just happen to be far removed from their usages."

In the typing rules we assume the existence of a global property mapping P from property names pn to type families $\mathscr{T}$. This technique is reminiscent of the way some object-oriented core languages provide a global class table for type-checking. The main difference is that our core language does not include constructs to extend the global property map; such constructs are left out of the core language for simplicity, since the creation of properties is not essential to our model. We require P to follow behavioral subtyping:

Definition 5.3.1. (Behavioral subtyping of property mapping) *If* $T <: T'$ *and* $T' \in P(pn)$, *then* $T \in P(pn)$

The typing rules are standard except for rules T-IMP, T-SPORE, and T-COMP, which are new. Only these three type rules inspect or modify the property environment Δ. Note that there is no rule for spore application, since there is a subtyping relationship between spores and functions (see Section 5.3.1). Using the subsumption rule T-SUB spore application is expressed using the standard rule for function application (T-APP).

Rule T-IMP imports a property pn into the property environment within the scope defined by term t.

Rule T-SPORE derives a type for a spore term. In the spore, all terms on right-hand sides of variable definitions in the spore header must be well-typed in the same environment $\Gamma;\Delta$ according to their declared type. The body of the spore's closure, t_2, must be well-typed in an environment containing only the variables in the spore header and the closure's parameter, one of the central properties of spores. The last premise requires all captured types to satisfy both the properties in the current property environment, Δ, as well as the properties listes in the spore term, Δ'. Finally, the resulting spore type contains the argument and result types of the spore's closure, the sequence of captured types according to the spore header, and the concatenation of properties Δ and Δ'. The intuition here is that properties in the environment have been explicitly imported by the user, thus indicating that all spores in the scope of the corresponding import should satisfy them.

Rule T-COMP derives a result type for the composition of two spores. It inspects the captured types of both spores ($\overline{S}$ and $\overline{R}$) to ensure that the properties of the resulting spore, Δ, are satisfied by the captured variables of both spores. Otherwise, the argument and result types are analogous to regular function composition. Note that it is possible to weaken the properties of a spore through spore subtyping and subsumption (T-SUB).

5.3.3 Operational semantics

Figure 5.9 shows the evaluation rules of a small-step operational semantics for our core language. The only non-standard rules are E-APPSPORE, E-SPORE, E-IMP, and E-COMP[10]3.

[10] For the sake of brevity, here we omit the standard evaluation rules. The complete set of evaluation rules can be found in Appendix B

E-AppSpore
$$\frac{\forall pn \in \overline{pn}.\ \overline{T} \subseteq P(pn)}{\texttt{spore}\ \{\ \overline{x : T = v}; \overline{pn}; (x' : T) \Rightarrow t\ \}\, v' \rightarrow \overline{[x \mapsto v]}[x' \mapsto v']\, t}$$

E-Spore
$$\frac{t_k \rightarrow t_k'}{\begin{array}{c}\texttt{spore}\ \{\ \overline{x : T = v}, x_k : T_k = t_k, \overline{x' : T' = t'}\ ; (x : T) \Rightarrow t\ \} \rightarrow \\ \texttt{spore}\ \{\ \overline{x : T = v}, x_k : T_k = t_k', \overline{x' : T' = t'}\ ; (x : T) \Rightarrow t\ \}\end{array}}$$

E-Imp
$$\texttt{import}\ pn\ \texttt{in}\ t \rightarrow insert(pn, t)$$

E-Comp1
$$\frac{t_1 \rightarrow t_1'}{t_1\ \texttt{compose}\ t_2 \rightarrow t_1'\ \texttt{compose}\ t_2}$$

E-Comp2
$$\frac{t_2 \rightarrow t_2'}{v_1\ \texttt{compose}\ t_2 \rightarrow v_1\ \texttt{compose}\ t_2'}$$

E-Comp3
$$\frac{\Delta = \{p \mid p \in \overline{pn}, \overline{qn}.\ \overline{T} \subseteq P(p) \wedge \overline{S} \subseteq P(p)\}}{\begin{array}{c}\texttt{spore}\ \{\ \overline{x : T = v}; \overline{pn}; (x' : T') \Rightarrow t\ \}\ \texttt{compose spore}\ \{\ \overline{y : S = w}; \overline{qn}; (y' : S') \Rightarrow t'\ \} \rightarrow \\ \texttt{spore}\ \{\ \overline{x : T = v}, \overline{y : S = w}; \Delta; (y' : S') \Rightarrow \texttt{let}\ z' = t'\ \texttt{in}\ [x' \mapsto z']\, t\}\end{array}}$$

Figure 5.9 – Operational Semantics[10]

H-InsSpore1
$$\frac{\forall t_i \in \overline{t}.\ insert(pn, t_i) = t_i' \qquad insert(pn, t) = t'}{insert(pn, \texttt{spore}\ \{\ \overline{x : T = t}; \overline{pn}; (x' : T) \Rightarrow t\ \}) = \qquad \texttt{spore}\ \{\ \overline{x : T = t'}; \overline{pn}, pn; (x' : T) \Rightarrow t'\ \}}$$

H-InsSpore2
$$\frac{insert(pn, t) = t'}{insert(pn, \texttt{spore}\ \{\ \overline{x : T = v}; \overline{pn}; (x' : T) \Rightarrow t\ \}) = \qquad \texttt{spore}\ \{\ \overline{x : T = v}; \overline{pn}, pn; (x' : T) \Rightarrow t'\ \}}$$

H-InsApp
$$insert(pn, t_1\ t_2) = insert(pn, t_1)\ insert(pn, t_2)$$

H-InsSel
$$insert(pn, t.l) = insert(pn, t).l$$

Figure 5.10 – Helper function *insert*

Rule E-AppSpore applies a spore literal to an argument. The differences to regular function application (E-AppAbs) are (a) that the types in the spore header must satisfy the properties of the spore dynamically, and (b) that the variables in the spore header must be replaced by their values in the body of the spore's closure. Rule E-Spore is a congruence rule. Rule E-Imp is a computation rule that is always enabled. It adds property name pn to all spore terms within the body t. The $insert$ helper function is defined in Figure 5.10 (we omit rules for `compose` and `let`; they are analogous to rules H-InsApp and H-InsSel).

Rule E-COMP3 is the computation rule for spore composition. Besides computing the composition in a way analogous to regular function composition, it defines the spore header of the result spore, as well as its properties. The properties of the result spore are restricted to those that are satisfied by the captured variables of both argument spores.

5.3.4 Soundness

This section presents a soundness proof of the spore type system. The proof is based on a pair of progress and preservation theorems [Wright and Felleisen, 1994]. A complete proof of soundness appears in Appendix B. In addition to standard lemmas, we also prove a lemma specific to our type system, Lemma 5.3.1, which ensures types are preserved under property import. Soundness of the type system follows from Theorem 5.3.1 and Theorem 5.3.2.

Theorem 5.3.1. (Progress) *Suppose t is a closed, well-typed term (that is, $\vdash t : T$ for some T). Then either t is a value or else there is some t' with $t \to t'$.*

Proof. By induction on a derivation of $\vdash t : T$. The only three interesting cases are the ones for spore creation, application, and spore composition. □

Lemma 5.3.1. (Preservation of types under import) *If* $\Gamma; \Delta, pn \vdash t : T$ *then* $\Gamma; \Delta \vdash insert(pn, t) : T$

Proof. By induction on a derivation of $\Gamma; \Delta, pn \vdash t : T$. □

Lemma 5.3.2. (Preservation of types under substitution) *If* $\Gamma, x : S; \Delta \vdash t : T$ *and* $\Gamma; \Delta \vdash s : S$, *then* $\Gamma; \Delta \vdash [x \mapsto s]t : T$

Proof. By induction on a derivation of $\Gamma, x : S; \Delta \vdash t : T$. □

Lemma 5.3.3. (Weakening) *If* $\Gamma; \Delta \vdash t : T$ *and* $x \notin dom(\Gamma)$, *then* $\Gamma, x : S; \Delta \vdash t : T$.

Proof. By induction on a derivation of $\Gamma; \Delta \vdash t : T$. □

Theorem 5.3.2. (Preservation) *If* $\Gamma; \Delta \vdash t : T$ *and* $t \to t'$, *then* $\Gamma; \Delta \vdash t' : T$.

Proof. By induction on a derivation of $\Gamma; \Delta \vdash t : T$. □

5.3.5 Relation to spores in Scala

The type soundness proof (see Section 5.3.4) guarantees several important properties for well-typed programs which closely correspond to the pragmatic model in Scala:

1. Application of spores: for each property name pn, it is ensured that the dynamic types of all captured variables are contained in the type family pn maps to ($P(pn)$).

2. Dynamically, a spore only accesses its parameter(s) and the variables in its header.

3. The properties computed for a composition of two spores is a safe approximation of the properties that are dynamically required.

$$
\begin{array}{lll}
t ::= \ldots & & \text{terms} \\
\quad \mid \texttt{spore}\,\{\,\overline{x : T = t}\,;\overline{T};\overline{pn};(x:T) \Rightarrow t\,\} & & \text{spore} \\
 & & \\
v ::= \ldots & & \text{values} \\
\quad \mid \texttt{spore}\,\{\,\overline{x : T = v}\,;\overline{T};\overline{pn};(x:T) \Rightarrow t\,\} & & \text{spore value} \\
 & & \\
\mathscr{S} ::= T \Rightarrow T\,\{\,\texttt{type}\,\mathscr{C} = \overline{T}\,;\,\texttt{type}\,\mathscr{E} = \overline{T}\,;\,\overline{pn}\,\} & & \text{spore type} \\
\quad \mid T \Rightarrow T\,\{\,\texttt{type}\,\mathscr{C}\,;\,\texttt{type}\,\mathscr{E} = \overline{T}\,;\,\overline{pn}\,\} & & \text{abstract spore type}
\end{array}
$$

Figure 5.11 – Core language syntax extensions

5.3.6 Excluded types

This section shows how the formal model can be extended with excluded types as described above (see Section 5.2.4). Figure 5.11 shows the syntax extensions: first, spore terms and values are augmented with a sequence of excluded types; second, spore types and abstract spore types get another member $\texttt{type}\,\mathscr{E} = \overline{T}$ specifying the excluded types.

Figure 5.12 shows how the subtyping rules for spores have to be extended. Rule S-ESPORE requires that for each excluded type T' in the supertype, there must be an excluded type T in the subtype such that $T' <: T$. This means that by excluding type T, subtypes like T' are also prevented from being captured.

Figure 5.13 shows the extensions to the operational semantics. Rule E-EAPPSPORE additionally requires that none of the captured types $\overline{T}$ are contained in the excluded types $\overline{U}$. Rule E-ECOMP3 computes the set of excluded types of the result spore in the same way as in the corresponding type rule (T-ECOMP).

Figure 5.14 shows the extensions to the typing rules. Rule T-ESPORE additionally requires that none of the captured types $\overline{S}$ is a subtype of one of the types contained in the excluded types $\overline{U}$. The excluded types are recorded in the type of the spore. Rule T-ECOMP computes a new set of excluded types $\overline{V}$ based on both the excluded types and the captured types of t_1 and t_2. Given that it is possible that one of the spores captures a type that is excluded in the other spore, the type of the result spore excludes only those types that are guaranteed not be captured.

S-ESPORE
$$\frac{T_2 <: T_1 \qquad R_1 <: R_2 \qquad \overline{pn'} \subseteq \overline{pn} \qquad M_1 = M_2 \vee M_2 = \texttt{type}\ \mathscr{C} \qquad \forall T' \in \overline{U'}.\ \exists T \in \overline{U}.\ T' <: T}{T_1 \Rightarrow R_1\ \{\ M_1\ ;\ \texttt{type}\ \mathscr{E} = \overline{U}\ ;\ \overline{pn}\ \} \qquad <: T_2 \Rightarrow R_2\ \{\ M_2\ ;\ \texttt{type}\ \mathscr{E} = \overline{U'}\ ;\ \overline{pn'}\ \}}$$

S-ESPOREFUN
$$T_1 \Rightarrow R_1\ \{\ M\ ;\ E\ ;\ \overline{pn}\ \} <: T_1 \Rightarrow R_1$$

Figure 5.12 – Subtyping extensions

E-EAPPSPORE
$$\frac{\forall pn \in \overline{pn}.\ \overline{T} \subseteq P(pn) \qquad \forall T_i \in \overline{T}.\ T_i \notin \overline{U}}{\texttt{spore}\ \{\ \overline{x : T = v}\ ;\ \overline{U}\ ;\ \overline{pn}\ ;\ (x' : T) \Rightarrow t\ \}\ v' \rightarrow \qquad \overline{[x \mapsto v]}[x' \mapsto v']t}$$

E-ECOMP3
$$\frac{\Delta = \{p \mid p \in \overline{pn}, \overline{qn}.\ \overline{T} \subseteq P(p) \wedge \overline{S} \subseteq P(p)\} \qquad \overline{V} = (\overline{U} \setminus \overline{S}) \cup (\overline{U'} \setminus \overline{T})}{\begin{array}{c}\texttt{spore}\ \{\ \overline{x : T = v}\ ;\ \overline{U}\ ;\ \overline{pn}\ ;\ (x' : T') \Rightarrow t\ \}\ \texttt{compose} \\ \texttt{spore}\ \{\ \overline{y : S = w}\ ;\ \overline{U'}\ ;\ \overline{qn}\ ;\ (y' : S') \Rightarrow t'\ \} \rightarrow \qquad \texttt{spore}\ \{\ \overline{x : T = v}, \overline{y : S = w}\ ;\ \overline{V}\ ;\ \Delta\ ; \\ (y' : S') \Rightarrow \texttt{let}\ z' = t'\ \texttt{in}\ [x' \mapsto z']t\ \}\end{array}}$$

Figure 5.13 – Operational semantics extensions

T-ESPORE
$$\frac{\begin{array}{c}\forall s_i \in \overline{s}.\ \Gamma;\Delta \vdash s_i : S_i \qquad \overline{y : S}, x : T_1; \Delta \vdash t_2 : T_2 \\ \forall pn \in \Delta, \Delta'.\ \overline{S} \subseteq P(pn) \qquad \forall S_i \in \overline{S}.\ \forall U_j \in \overline{U}.\ \neg(S_i <: U_j)\end{array}}{\Gamma;\Delta \vdash \texttt{spore}\ \{\ \overline{y : S = s}\ ; \overline{U}; \Delta'; (x : T_1) \Rightarrow t_2\ \} : \qquad T_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}\ ;\ \texttt{type}\ \mathscr{E} = \overline{U}\ ;\ \Delta, \Delta'\ \}}$$

T-ECOMP
$$\frac{\begin{array}{c}\Gamma;\Delta \vdash t_1 : T_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}\ ;\ \texttt{type}\ \mathscr{E} = \overline{U}\ ;\ \Delta_1\ \} \\ \Gamma;\Delta \vdash t_2 : U_1 \Rightarrow T_1\ \{\ \texttt{type}\ \mathscr{C} = \overline{R}\ ;\ \texttt{type}\ \mathscr{E} = \overline{U'}\ ;\ \Delta_2\ \} \\ \Delta' = \{pn \in \Delta_1 \cup \Delta_2 \mid \overline{S} \subseteq P(pn) \wedge \overline{R} \subseteq P(pn)\} \qquad \overline{V} = (\overline{U} \setminus \overline{R}) \cup (\overline{U'} \setminus \overline{S})\end{array}}{\Gamma;\Delta \vdash t_1\ \texttt{compose}\ t_2 : U_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}, \overline{R}\ ;\ \texttt{type}\ \mathscr{E} = \overline{V}\ ;\ \Delta'\ \}}$$

Figure 5.14 – Typing extensions

5.4 Implementation

We have implemented spores as a macro library for Scala 2.10 and 2.11. Macros are an experimental feature introduced in Scala 2.10 that enable "macro defs," methods that take expression trees as arguments and that return an expression tree that is inlined at each invocation site. Macros are expanded during type checking in a way which enables macros to synthesize their result type specialized for each expansion site.

The implementation for Scala 2.10 requires in addition a compiler plug-in that provides a backport of the support for Java 8 SAM types ("functional interfaces") of Scala 2.11. SAM type support extends type inference for user-defined subclasses of Scala's standard function types which enables infering the types of spore parameters.

An expression `spore { val y: S = s; (x: T) => /* body */ }` invokes the `spore` macro which is passed the block `{ val y... }` as an expression tree. A spore without type constraints simply checks that within the body of the spore's closure, only the parameter `x` as well as the variables in the spore header are accessed according to the spore type-checking rules. The expression tree returned by the macro creates an instance of a *refinement type* of the abstract `Spore` class that implements its `apply` method (inherited from the corresponding standard Scala function trait) by applying the spore's closure. The `Captured` type member (see Section 5.2.2) is defined by the generated refinement type to be a tuple type with the types of all captured variables. If there are no type constraints the `Excluded` type member is defined to be `No[Nothing]`.

Type constraints are implemented as follows. First, invoking the generic `without` macro passing a type argument `T`, say, augments the generated `Spore` refinement type by effectivly adding the clause `with No[T]` to the definition of its `Excluded` type member. Second, the existence of additional bounds on the captured types is detected by attempting to infer an implicit value of type `Property[_]`. If such an implicit value can be inferred, a sequence of types specifying type bounds is obtained as follows. The type of the implicit value is matched against the pattern `Property[t1] with ... with Property[tn]`. For each type `ti` an implicit member of the following shape is added to the `Spore` type refinement:

```
implicit val evi: ti[Captured] = implicitly[ti[Captured]]
```

The implicit conversion (Section 5.2.3) from standard Scala functions to spores is implemented as a macro whose expansion fails if the argument function is not a literal, since in this case it is impossible for the macro to check the spore shape/capturing constraints.

5.5 Evaluation

In this section we evaluate the practicality and the benefits of using spores as an alternative to normal closures in Scala. The evaluation has two parts. In the first part we measure the impact of introducing spores in existing programs. In the second part we evaluate the utility and the syntactic overhead of spores in a large code base of applications based on the Apache Spark framework for big data analytics.

Program	LOC	#closures	#converted	LOC changed	#captured vars	
funsets	99	8	8	7	9	MOOC
forcomp	201	6	4	4	0	
mandelbrot	325	1	1	9	6	Parallel Collections
barneshut	722	7	7	8	1	
spark pagerank	64	5	5	8	0	Spark
spark kmeans	92	5	4	9	2	
Total	1503	32	29	45	18	

Figure 5.15 – Evaluating the practicality of using spores in place of normal closures

5.5.1 Using Spores Instead of Closures

In this section we measure the number of changes required to convert existing programs that crucially rely on closures to use spores. We analyze a number of real Scala programs, taken from three categories:

1. General, closure-heavy code, taken from the exercises of the popular MOOC on Functional Programming Principles in Scala; the goal of analyzing this code is to get an approximation of the worst-case effort required when consistently using spores instead of closures, in a mostly-functional code base.

2. Parallel applications based on Scala's parallel collections. These examples evaluate the practicality of using spores in a parallel code base to increase its robustness.

3. Distributed applications based on the Apache Spark cluster computing framework. In this case, we evaluate the practicality of using spores in Spark applications to make sure closures are guaranteed to be serializable.

Methodology For each program, we obtained (a) the number of closures in the program that are candidates for conversion, (b) the number of closures that could be converted to spores, (c) the changed/added number of LOC, and (d) the number of captured variables. It is important to note that during the conversion it was not possible to rely on an implicit conversion of functions to spores, since the expected types of all library methods that were invoked by the evaluated applications remained normal function types. Thus, the reported numbers are worse than they would be for APIs using spores.

Results The results are shown in Figure 5.15. Out of 32 closures 29 could be converted to spores with little effort. One closure failed to infer its parameter type when expressed as a spore. Two other closures could not be converted due to implementation restrictions of our prototype. On average, per converted closure 1.4 LOC had to be changed. This number is dominated by two factors: the inability to use the implicit conversion from functions to spores, and one particularly complex closure in "mandelbrot" that required changing 9 LOC. In our

Project	average LOC per closure	average # of captured vars	% closures that don't capture
sameeragarwal/blinkdb ★268 33 LOC 22,022	1.39	1	93.5%
freeman-lab/thunder ★89 2 LOC 2,813	1.03	1.30	23.3%
bigdatagenomics/adam ★86 16 LOC 19,055	1.90	1.44	80.2%
ooyala/spark-jobserver ★79 6 LOC 5,578	1.60	1	80.0%
Sotera/correlation-approximation ★12 2 LOC 775	4.55	1.25	63.6%
aecc/stream-tree-learn ★1 2 LOC 1,199	5.73	2	54.5%
lagerspetz/TimeSeriesSpark ★5 1 LOC 14,882	2.85	1.77	75.0%
Total LOC 66,324	2.25	1.39	67.2%

Figure 5.16 – Evaluating the impact and overhead of spores on real distributed applications. Each project listed is an active and noteworthy open-source project hosted on GitHub that is based on Apache Spark. ★ represents the number of "stars" (or interest) a repository has on GitHub, and 👥 represents the number of contributors to the project.

programs, the number of captured variables is on average 0.56. These results suggest that programs using closures in non-trivial ways can typically be converted to using spores with little effort, even if the used APIs do not use spore types.

5.5.2 Spores and Apache Spark

To evaluate both benefit and overhead of using spores in larger, distributed applications, we studied the codebases of 7 noteworthy open-source applications using Apache Spark.

Methodology We evaluated the applications along two dimensions. In the first dimension we were interested how widespread patterns are that spores could statically enforce. In the context of open-source applications built on top of the Spark framework, we counted the number of closures passed to the higher-order `map` method of the `RDD` type (Spark's distributed collection abstraction); all of these closures must be serializable to avoid runtime exceptions. (The `RDD` type has several more higher-order functions that require serializable closures such as `flatMap`; `map` is the most commonly used higher-order function, though, and is thus representative of the use of closures in Spark.) In the second dimension, we analyzed the percentage of spores that could be converted automatically to spores assuming the Spark API would use spore types instead of regular function types, thus not incurring any syntactic overhead. In cases where automatic conversion would be impossible, we analyzed the average

number of captured variables, indicating the syntactic overhead of using explicit spores.

Results Figure 5.16 summarizes our results. Of all closures passed to `RDD`'s `map` method, about 67.2% do not capture any variable; these closures could be automatically converted to spores using the implicit macro of Section 5.2.3. The remaining 32.8% of closures that do capture variables, capture on average 1.39 variables. This indicates that unchecked patterns for serializable closures are widespread in real applications, and that benefiting from static guarantees provided by spores would require only little syntactic overhead.

5.5.3 Spores and Akka

We have also verified that excluding specific types from closures is important.

The Akka event-driven middleware provides an actor abstraction for concurrency. When using futures together with actors, it is common to provide the result of a future-based computation to the sender of a message sent to an actor.

However, naive implementations of patterns such as this can problematic. To access the sender of a message, Akka's `Actor` trait provides a method `sender` that returns a reference to the actor that is the sender of the message currently being processed. There is a potential for a data race where the actor starts processing a message from a different actor than the original sender, but a concurrent future-based computation invokes the `sender` method (on `this`), thus obtaining a reference to the wrong actor.

Given the importance of combining actors and futures, Akka provides a library method `pipeTo` to enable programming patterns using futures that avoid capturing variables of type `Actor` in closures. However, the correct use of `pipeTo` is unchecked. Spores provide a new statically-checked approach to address this problem by demanding closures passed to future constructors to be spores with the constraint that type `Actor` is excluded.

Methodology To find out how often spores with type constraints could turn an unchecked pattern into a statically-checked guarantee, we analyzed 7 open-source projects using Akka (GitHub projects with 23 stars on average; more than 100 commits; 2.7 contributors on average). For each project we searched for occurrences of "pipeTo" directly following closures passed to future constructors.

Results The 7 projects contain 19 occurrences of the presented unchecked pattern to avoid capturing Actor instances within closures used concurrently. Spores with a constraint to exclude Actor statically enforce the safety of all those closures.

5.6 Case Study

Frameworks like MapReduce [Dean and Ghemawat, 2008] and Apache Spark [Zaharia et al., 2012] are designed for processing large datasets in a cluster, using well-known map/reduce computation patterns.

In Spark, these patterns are expressed using higher-order functions, like `map`, applied to the "resilient distributed dataset" (RDD) abstraction. However, to avoid unexpected runtime exceptions due to unserializable closures when passing closures to RDDs, programmers must adopt conventions that are subtle and unchecked by the Scala compiler.

The pattern shown in Figure 5.17 is a common pattern that was extracted from a code base used in production.

```
class GenericOp(sc: SparkContext, mapping: Map[String, String]) {
  private var cachedSessions: spark.RDD[Session] = ...

  def doOp(keyList: List[...], ...): Result = {
    val localMapping = mapping

    val mapFun: Session => (List[String], GenericOpAggregator) = { s =>
      (keyList, new GenericOpAggregator(s, localMapping))
    }

    val reduceFun: (GenericOpAggregator, GenericOpAggregator) =>
      GenericOpAggregator = { (a, b) => a.merge(b) }

    cachedSessions.map(mapFun).reduceByKey(reduceFun).collectAsMap
  }
}
```

Figure 5.17 – Conventions used in production to avoid serialization errors.

Here, the `doOp` method performs operations on the RDD `cachedSessions`. `GenericOp` has a parameter of type `SparkContext`, the main entry point for functionality provided by Spark, and a parameter of type `Map[String, String]`. The main computation is a chain of invocations of `map`, `reduceByKey`, and `collectAsMap`. To ensure that the argument closures of `map` and `reduceByKey` are serializable, the code follows two conventions: first, instead of defining `mapFun` and `reduceFun` as methods, they are defined using lambdas stored in local variables. Second, instead of using the `mapping` parameter directly, it is first copied into a local variable `localMapping`. The reason for the first convention is that in Scala converting a method to a function implicitly captures a reference to the enclosing object. However, `GenericOp` is not serializable, since it refers to a `SparkContext`. The reason for the second convention is that using `mapping` directly would result in `mapFun` capturing `this`.

Applying Spores

The above conventions can be enforced by the compiler, avoiding unexpected runtime exceptions, by turning `mapFun` and `reduceFun` into spores:

```
val mapFun: Spore[Session, (List[String], GenericOpAggregator)] =
  spore { val localMapping = mapping
    (s: Session) => (keyList, new GenericOpAggregator(s, localMapping)) }
val reduceFun: Spore[(GenericOpAggregator, GenericOpAggregator),
                     GenericOpAggregator] =
  spore { (a, b) => a.merge(b) }
```

The spore shape enforces the use of `localMapping` (moved into `mapFun`). Furthermore, there is no more possibility of accidentally capturing a reference to the enclosing object.

5.7 Related Work

Cloud Haskell [Epstein et al., 2011] provides statically guaranteed-serializable closures by either rejecting environments outright, or by allowing manual capturing, requiring the user to explicitly specify and pre-serialize the environment in combination with top-level functions (enforced using a new `Static` type constructor). That is, in Cloud Haskell, to create a serializable closure, one must explicitly pass the serialized environment as a parameter to the function – this requires users to have to refactor closures they wish to be made serializable. In contrast, spores do not require users to manually factor out, manage, and serialize their environment; spores require only that *what* is captured is specified, not *how*. Furthermore, spores are more general than Cloud Haskell's serializable closures; user-defined type constraints enable spores to express more properties than just serializability, like thread-safety, immutability, or any other user-defined property. In addition, spores allow restricting captured types in a way that is integrated with object-oriented concerns, such as subtyping and open class hierarchies.

C++11 [International Standard ISO/IEC 14882:2011, 2011] has introduced syntactic rules for explicit capture specifications that indicate which variables are captured and how (by reference or by copy). Since the capturing semantics is purely syntactic, a capture specification is only enforced at closure creation time. Thus, when composing two closures, the capture semantics is not preserved. Spores, on the other hand, capture such specifications at the level of types, enabling composability. Furthermore, spores' type constraints enable more general type-directed control over capturing than capture-by-value or capture-by-reference alone.

A preliminary proposal for closures in the Rust language [Matsakis, 2013] allows describing the closed-over variables in the environment using closure bounds, requiring captured types to implement certain traits. Closure bounds are limited to a small set of built-in traits to enforce properties like sendability. Spores on the other hand enable user-defined property

definition, allowing for greater customizability of closure capturing semantics. Furthermore, unlike spores, the environment of a closure in Rust must always be allocated on the stack (although not necessarily the top-most stack frame).

Java 8 [Goetz, 2013] introduces a limited type of closure which is only permitted to capture variables that are effectively-final. Like with Scala's standard closures, variable capture is implicit, which can lead to accidental captures that spores are designed to avoid. Although serializability can be requested at the level of the type system using newly-introduced intersection types in Java 8, there is no guarantee about the absence of runtime exceptions, as there is for spores. Finally, spores additionally allow specifying type-based constraints for captured variables that are more general than serializability alone.

Parallel closures [Matsakis, 2012] are a variation of closures that make data in the environment available using read-only references using a type system for reference immutability. This enables parallel execution without the possibility of data races. Spores are not limited to immutable environments, and do not require a type system extension. River Trail [Herhut et al., 2013] provides a concurrency model for JavaScript, similar to parallel closures; however, capturing variables in closures is currently not supported.

ML5 [Murphy VII et al., 2007] provides mobile closures verified not to use resources not present on machines where they are applied. This property is enforced transitively (for all values reachable from captured values), which is stronger than what plain spores provide. However, type constraints allow spores to require properties not limited to mobility. Transitive properties are supported either using type constraints based on type classes which enforce a transitive property or by integrating with type systems that enforce transitive properties. Unlike ML5, spores do not require a type system extension.

A well-known type-based representation of closures uses existential types where the existentially quantified variable represents the closure's environment, enabling type-preserving compilation of functional languages [Morrisett et al., 1999]. A spore type may have an abstract `Captured` type, effectively encoding an existantial quantification; however, captured types are typically concrete, and the spore type system supports constraints on them.

HdpH [Maier and Trinder, 2011] generalizes Cloud Haskell's closures in several aspects: first, closures can be transformed without eliminating them. Second, unnecessary serialization is avoided, e.g., when applying a closure immediately after creation. Otherwise, the discussion of Cloud Haskell in above also applies to HdpH. Delimited continuations [Rompf et al., 2009] represent a way to serialize behavior in Scala, but don't resolve any of the problems of normal Scala closures when it comes to accidental capture, as spores do.

Termite Scheme [Germain, 2006] is a Scheme dialect for distributed programming where closures and continuations are always serializable; references to non-serializable objects (like open files) are automatically wrapped in processes that are serialized as their process ID. In contrast, with spores there is no such automatic wrapping. Unlike closures in Termite

Scheme, spores are statically-typed, supporting type-based constraints. Serializable closures in a dynamically-typed setting are also the basis for [Schwendner, 2009]. Python's standard serialization module, `pickle`, does not support serializing closures. Dill [McKerns et al., 2012] extends Python's `pickle` module, adding support for functions and closures, but without constraints.

5.8 Conclusion

This chapter presented a type-based foundation for closures, called spores, designed to avoid various hazards that arise particularly in concurrent or distributed settings. We have presented a flexible type system for spores which enables composability of differently-constrained spores as well as custom user-defined type constraints. We formalize and present a full soundness proof, as well as an implementation of our approach in Scala.

A key takeaway of our approach is that including type information of captured variables in the type of the spore enables a number of previously impossible opportunities, including but not limited to controlled capture in concurrent, distributed, and other arbitrary scenarios where closures must be controlled.

Finally, we demonstrate the practicality of our approach through an empirical study, and show that converting non-trivial programs to use spores requires relatively little effort.

6 Function-Passing

In Chapter 3 and 5 we covered *pickling* and *spores*, a serialization framework and an abstraction for statically checked and serializable function closures. In this chapter, we bring these two frameworks together in the form of a new programming model that provides a principled substrate for well-typed functional distributed programming called *function-passing*.

6.1 Introduction

It is difficult to deny that data-centric programming is growing in importance. At the same time, it is no secret that the most successful systems for programming with "big data" have all adopted ideas from functional programming; *i.e.*, programming with first-class functions and higher-order functions. These functional ideas are often touted to be the key to the success of these frameworks. It is not hard to imagine why–a functional, declarative interface to data distributed over tens, or hundreds, or even thousands of nodes provides a more natural way for end-users and data scientists to reason about data.

While leveraging functional programming *concepts*, popular implementations of the MapReduce [Dean and Ghemawat, 2008] model, such as Hadoop MapReduce [Apache, 2015] for Java, have been developed without making use of functional language features such as closures. In contrast, a new generation of programming systems for large-scale data processing, such as Apache Spark [Zaharia et al., 2012], Twitter's Scalding [Twitter, 2015], and Scoobi [NICTA, 2015] build on functional language features in Scala in order to provide high-level, declarative APIs.

Due to the limited support for distribution in the languages that are used to implement these systems, even well-developed and widely used systems still encounter issues that can complicate their use or optimization. Some of these include:

- **Usage Errors** These systems' APIs cannot statically prevent *common usage errors* resulting from some language features not being designed with distribution in mind, often confronting users with hard-to-debug runtime errors. A common example is unsafe

closure serialization [Miller et al., 2014a].

- **Lost Optimization Opportunities** The absence of certain kinds of static type information precludes systems-centric optimizations. Importantly, type-based static metaprogramming enables fast serialization [Miller et al., 2013], but this is only possible if also lower layers (namely those dealing with object serialization) are statically typed. Several studies [Oracle, Inc., 2011, Philippsen et al., 2000, Pitt and McNiff, 2001, Welsh and Culler, 2000] report on the high overhead of serialization in widely-used runtime environments such as the JVM. Researchers have even found that for some jobs, as much as half of the CPU time is spent deserializing and decompressing data on a Spark cluster [Ousterhout et al., 2015]. This overhead is so important in practice that popular systems, like Spark [Zaharia et al., 2012] and Akka [Typesafe, 2009], often leverage alternative serialization frameworks such as Protocol Buffers [Google, 2008], Apache Avro [Apache, 2013], or Kryo [Nathan Sweet, 2013] to meet their performance requirements.

- **Lack of Formal Semantics** As it stands, popular system designs don't allow formal reasoning about important systems-oriented concerns such as fault recorvery due a lack of formal operational models. As a result, formal reasoning is not available for the development of these systems; *i.e.*, such systems tend not to be built upon foundations with a formal semantics.

We present a new programming model we call *function passing* (F-P) designed to overcome most of these issues by providing a more principled substrate on which to build typed, functional data-centric distributed systems. It builds upon two previous veins of work–an approach for generating type-safe and performant pickler combinators [Miller et al., 2013], and spores [Miller et al., 2014a], closures that are guaranteed to be serializable. Our model attempts to fit the paradigm of data-centric programming more naturally by extending monadic programming to the network. Our model can be thought of as somewhat of a dual to the actor model;[1] rather than keeping functionality stationary and sending data, in our model, we keep data stationary and send functionality to the data. This results in well-typed communication by design, a common pain point for builders of distributed systems in Scala. Our model is in no small part inspired by Spark, and can be thought of as a generalization of its programming model.

Our model brings together immutable, persistent data structures, monadic higher-order functions, strong static typing, and lazy evaluation–pillars of functional programming–to provide a more type-safe, and easy to reason about foundation for data-centric distributed systems. Interestingly, we found that laziness was an enabler in our model, without complicating the

[1] There are many variations and interpretations of the actor model; in saying our model is somewhat of a dual, we simply mean to highlight that programmers need not focus on programming with typically stationary message handlers. Instead, our model focuses on a monadic interface for programming with data (and sending functions instead).

ability to reason about programs. Without optimizations based on laziness, we found this model would be impractically inefficient in memory and time.

One important contribution of our model is a precise specification of the semantics of functional fault recovery. The fault-recovery mechanisms of widespread systems such as Apache Spark, MapReduce [Dean and Ghemawat, 2008] and Dryad [Isard et al., 2007] are based on the concept of a *lineage* [Bose and Frew, 2005, Cheney et al., 2009]. Essentially, the lineage of a data set combines (a) an initial data set available on stable storage and (b) a sequence of transformations applied to initial and subsequent data sets. Maintaining such lineages enables fault recovery through recomputation.

6.1.1 Contributions

The F-P-related contributions of this thesis include:

- ***A new data-centric programming model for functional processing of distributed data*** which makes important concerns like fault tolerance simple by design. The main computational principle is based on the idea of sending safe, guaranteed serializable functions to stationary data. Using standard monadic operations our model enables creating immutable DAGs of computations, supporting decentralized distributed computations. Lazy evaluation enables important optimizations while keeping programs simple to reason about.

- ***A formalization of our programming model*** based on a small-step operational semantics. To our knowledge it is the first formal account of fault recovery based on lineage in a purely functional setting. Inspired by widespread systems like Spark [Zaharia et al., 2012], our formalization is a first step towards a formal, operational account of real-world fault recovery mechanisms. The presented semantics is clearly stratified into a deterministic layer and a concurrent/distributed layer. Importantly, reasoning techniques for sequential programs are not invalidated by the distributed layer.

- ***An implementation of the programming model*** in and for Scala. We present experiments that show some of the benefits of the proposed design, and we report on a validation of spores in the context of distributed programming.

This chapter proceeds first with a description of the F-P model from a high-level, elaborating upon key benefits and trade-offs, then zooming in to make each component part of the F-P model more precise. We describe the basic model this way in Section 6.2. We go on to show in Section 6.3 how essential higher-order operations on distributed frameworks like Spark can be implemented in terms of the primitives presented in Section 6.2. We present a formalization of our programming model in Section 6.4, and an overview of its prototypical implementation in Section 6.5. Finally, we discuss related work in Section 6.6, and conclude in Section 6.7.

6.2 Overview of Model

The best way to quickly visualize the F-P model is to think in terms of a persistent functional data structure with structural sharing. A *persistent data structure* is a data structure that always preserves the previous version of itself when it is modified–such data structures are effectively immutable, as their operations do not (visibly) update the structure in-place, but instead always yield a new updated structure. Then, rather than containing pure data, imagine instead that the data structure represents a directed acyclic graph (DAG) of transformations on data that is distributed.

Importantly, since this DAG of computations is a persistent data structure itself, it is safe to exchange (copies of) subgraphs of a DAG between remote nodes. This enables a robust and easy-to-reason-about model of fault tolerance. We call subgraphs of a DAG *lineages*; lineages enable restoring the data of failed nodes through re-applying the transformations represented by their DAG. This sequence of applications must begin with data available from stable storage.

Central to our model is the careful use of laziness. Computations on distributed data are typically not executed eagerly; instead, applying a function to distributed data just creates an immutable lineage. To obtain the result of a computation, it is necessary to first "kick off" computation, or to "force" its lineage. Within our programming model, this force operation makes network communication (and thus possibilities for latency) explicit, which is considered to be a strength when designing distributed systems [Waldo et al., 1996]. Deferred evaluation also enables optimizing distributed computations through operation fusion, which avoids the creation of unnecessary intermediate data structures–which is more efficient in time as well as space. This kind of optimization is particularly important and effective in distributed systems [Chambers et al., 2010b].

> For these reasons, we believe that laziness should be viewed as an enabler in the design of distributed systems.

The F-P model consists of three main components:

- **Silos:** stationary typed data containers.
- **SiloRefs:** references to local or remote Silos.
- **Spores:** safe, serializable functions.

Silos A silo is a typed data container. It is stationary in the sense that it does not move between machines – it remains on the machine where it was created. Data stored in a silo is typically loaded from stable storage, such as a distributed file system. A program operating on data stored in a silo can only do so using a reference to the silo, a `SiloRef`.

SiloRefs Similar to a proxy object, a SiloRef represents, and allows interacting with, both local and remote silos. SiloRefs are immutable, storing identifiers to locate possibly remote silos. SiloRefs are also typed (`SiloRef[T]`) corresponding to the type of their silo's data, leading to well-typed network communication. That is, by parameterizing SiloRefs, it becomes impossible by design to apply transformations (*e.g.*, to apply a function) to that data unless the type of the function agrees with the type of the data stored in the corresponding Silo. This avoids a common pitfall of actor-based programming in Scala; since communication between actors is untyped[2] (an actor's message handler's type in Scala is `Any => Unit`) developers commonly run into hung and timed-out systems during system development due to the `Any => Unit` message handler in an actor receiving a message of a type that is not explicitly handled by the programmer. In F-P, this situation is avoided by design in that these sorts of errors are caught at compile-time rather than requiring a programmer to debug a hung system at runtime.

The SiloRef provides three primitive operations/combinators (some are lazy, some are not): `map`, `flatMap`, and `send`. `map` lazily applies a user-defined function to data pointed to by the SiloRef, creating in a new silo containing the result of this application. Like `map`, `flatMap` lazily applies a user-defined function to data pointed to by the SiloRef. Unlike `map`, the user-defined function passed to `flatMap` returns a SiloRef whose contents is transferred to the new silo returned by `flatMap`. Essentially, `flatMap` enables accessing the contents of (local or remote) silos from within remote computations. We illustrate these primitives in more detail in Section 6.2.2.

Spores As introduced in Chapter 5, spores [Miller et al., 2014a] are safe closures that are guaranteed to be serializable and thus distributable. The following is a review of the important characteristics of spores as they pertain to the F-P model.

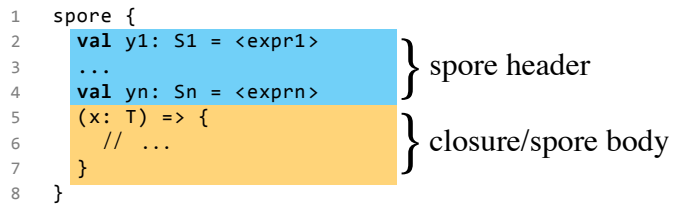
Spores are a closure-like abstraction and type system which gives authors of distributed frameworks a principled way of controlling the environment which a closure (provided by client code) can capture. This is achieved by (a) enforcing a specific syntactic shape which dictates how the environment of a spore is declared, and (b) providing additional type-checking to ensure that types being captured have certain properties.

A spore consists of two parts:

[2] There are several ongoing efforts aimed at typed communication between actors [He et al., 2014, Kuhn, 2015].

- **the spore header**, composed of a list of value definitions.
- **the spore body** (sometimes referred to as the "spore closure"), a regular closure.

This shape is illustrated below.



The characteristic property of a spore is that the spore body is only allowed to access its parameter, the values in the spore header, as well as top-level singleton objects (Scala's form of modules). The spore closure is not allowed to capture variables other than those declared in the spore header (*i.e.,* the spore *closure* may not capture variables in the environment enclosing the spore). By enforcing this shape, the environment of a spore is always declared explicitly in the spore header, which avoids accidentally capturing problematic references. Moreover, importantly for object-oriented languages like Scala, it's no longer possible to accidentally capture the `this` reference.

Spores also come with additional type-checking. Type information corresponding to captured variables are included in the type of a spore. This enables authors of distributed frameworks to customize type-checking of spores to, for example, *exclude* a certain type from being captured by user-provided spores. Authors of distributed frameworks may kick on this type-checking by simply including information about excluded types (or other type-based properties) in the signature of a method. A concrete example would be to ensure that the `map` method on `RDD`s in Spark (a distributed collection) accepts only spores which do not capture `SparkContext` (a non-serializable internal framework class).

For a deeper understanding of spores, see Chapter 5.

6.2.1 Basic Usage

We begin with a simple visual example to provide a feeling for the basics of the F-P model.

The only way to interact with distributed data stored in silos is through the use of SiloRefs. A SiloRef can be thought of as an immutable handle to the remote data contained within a corresponding silo. Users interact with this distributed data by applying functions to SiloRefs, which are transmitted over the wire and later applied to the data within the corresponding silo. As is the case for persistent data structures, when a function is applied to a piece of distributed data via a SiloRef, a SiloRef representing a new silo containing the transformed data is returned.

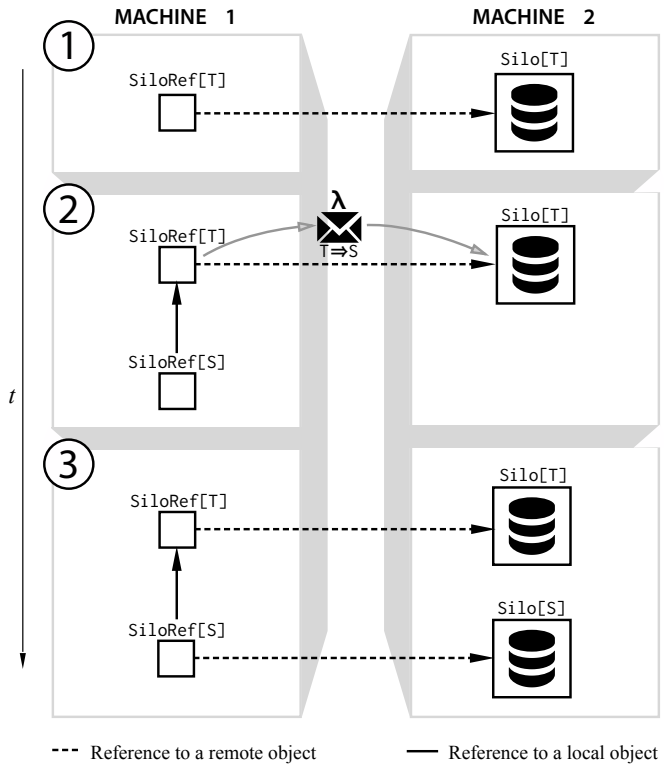


Figure 6.1 – Basic F-P model.

The simplest illustration of the model is shown in Figure 6.1 (time flows vertically from top to bottom). Here, we start with a `SiloRef[T]` which points to a piece of remote data contained within a `Silo[T]`. When the function shown as λ of type $T \Rightarrow S$ is applied to `SiloRef[T]` and "forced" (sent over the wire), a new SiloRef of type `SiloRef[S]` is immediately returned. Note that `SiloRef[S]` contains a reference to its parent SiloRef, `SiloRef[T]`. (This is how *lineages* are constructed.) Meanwhile, the function is asynchronously sent over the wire and is applied to `Silo[T]`, eventually producing a new `Silo[S]` containing the data transformed by function λ. This new `SiloRef[S]` can be used even before its corresponding silo is materialized (*i.e.,* before the data in `Silo[S]` is computed) – the F-P framework queues up operations applied to `SiloRef[S]` and applies them when `Silo[S]` is fully materialized.

Different sorts of complex DAGs can be asynchronously built up in this way. Though first, to see how this is possible, we need to develop a clearer idea of the primitive operations available on SiloRefs and their semantics. We describe these in the following section.

6.2.2 Primitives

There are four basic primitive operations on SiloRefs that together can be used to build the higher-order operations common to popular data-centric distributed systems (how to build some of these higher-order operations is described in Section 6.3). In this section we'll

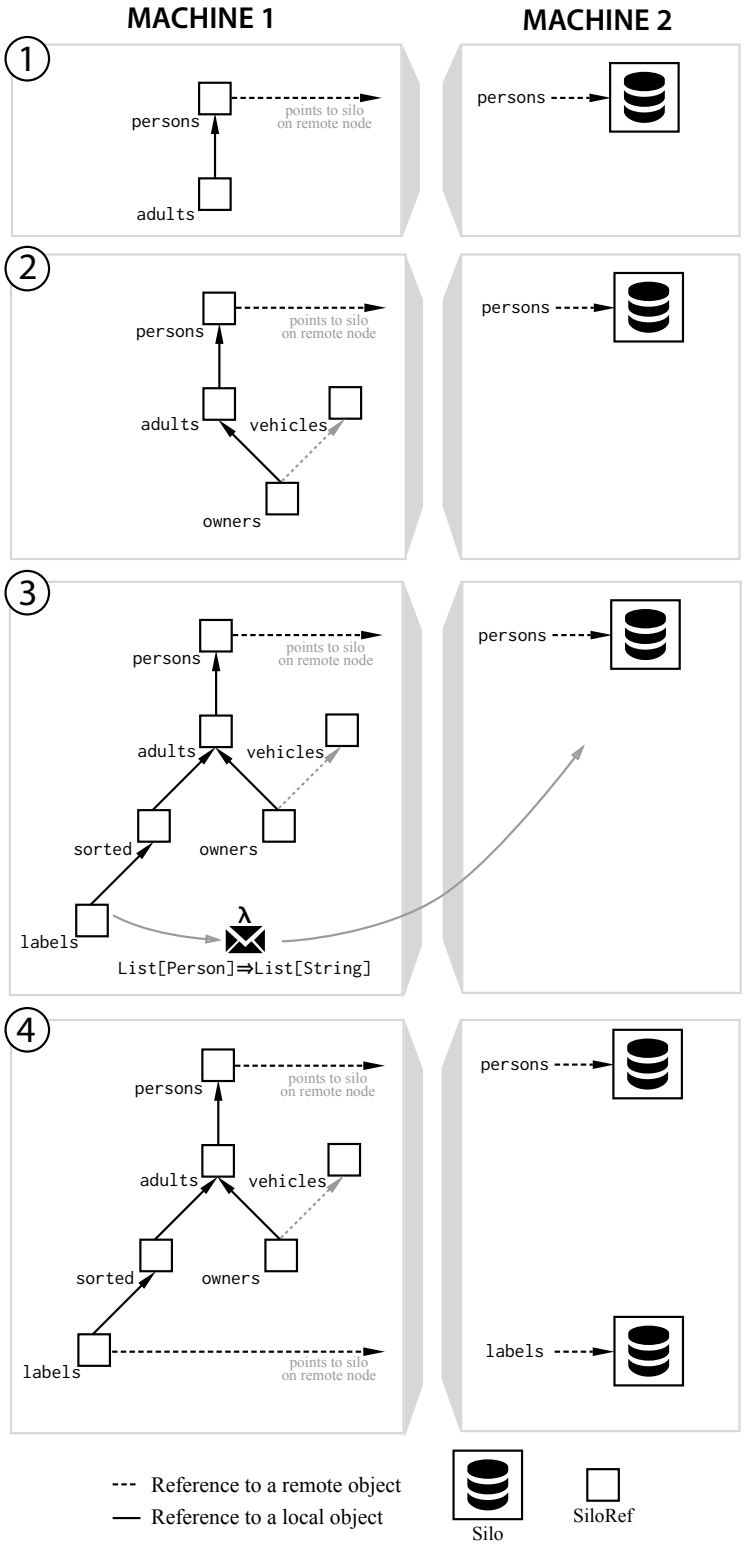


Figure 6.2 – A simple DAG in the F-P model.

introduce these primitives in the context of a running example. These primitives include:

- `map`
- `flatMap`
- `send`
- `cache`

map `def map[S](s: Spore[T, S]): SiloRef[S]`
The `map` method takes a spore that is to be applied to the data in the silo associated with the given SiloRef. Rather than immediately sending the spore across the network, and waiting for the operation to finish, the `map` method is *lazy*. Without involving any network communication, it immediately returns a SiloRef referring to a new, lazily-created silo. This new SiloRef only contains lineage information, namely, a reference to the original SiloRef, a reference to the argument spore, and the information that it is the result of a `map` invocation. As we explain below, another method, `send` or `cache`, must be called explicitly to force the materialization of the result silo.

To better understand how DAGs are created and how remote silos are materialized, we will develop a running example throughout this section. Given a silo containing a list of `Person` records, the following application of `map` defines a (not-yet-materialized) silo containing only the records of adults (graphically shown in Figure 6.2, part 1):

```
val persons: SiloRef[List[Person]] = ...
val adults =
  persons.map(spore { ps => ps.filter(p => p.age >= 18) })
```

flatMap `def flatMap[S](s: Spore[T, SiloRef[S]]): SiloRef[S]`
Like `map`, the `flatMap` method takes a spore that is to be applied to the data in the silo of the given SiloRef. However, the crucial difference is in the type of the spore argument whose result type is a SiloRef in this case. Semantically, the new silo created by `flatMap` is defined to contain the data of the silo that the user-defined spore returns. The `flatMap` combinator adds expressiveness to our model that is essential to express more interesting computation DAGs. For example, consider the problem of combining the information contained in two different silos (potentially located on different hosts). Suppose the information of a silo containing `Vehicle` records should be enriched with other details only found in the `adults` silo. In the following, `flatMap` is used to create a silo of `(Person, Vehicle)` pairs where the names of person and vehicle owner match (graphically shown in Figure 6.2, part 2):

Note that the spore passed to `flatMap` declares the capturing of the `vehicles` SiloRef in its so-called "spore header." The spore header spans all variable definitions between the spore marker and the parameter list of the spore's closure. The spore header defines the variables that the spore's closure is allowed to access. Essentially, spores limit the free variables of their

```
val vehicles: SiloRef[List[Vehicle]] = ...
// adults that own a vehicle
val owners = adults.flatMap(spore {
  val localVehicles = vehicles // spore header
  ps =>
    localVehicles.map(spore {
      val localps = ps // spore header
      vs =>
        localps.flatMap(p =>
          // list of (p, v) for a single person p
          vs.flatMap {
            v => if (v.owner.name == p.name) List((p, v)) else Nil
          }
        )
    })
})
```

closure's body to the closure's parameters and the variables declared in the spore's header. Within the spore's closure, it is necessary to read the data of the `vehicles` silo in addition to the `ps` list of `Person` records. This requires calling `map` on `localVehicles`. However, `map` returns a SiloRef; thus, invoking `map` on `adults` instead of `flatMap` would be impossible, since there would be no way to get the data out of the silo returned by `localVehicles.map(..)`. With the use of `flatMap`, however, the call to `localVehicles.map(..)` creates the final result silo, whose data is then also contained in the silo returned by `flatMap`.

Although the expressiveness of the `flatMap` combinator subsumes that of the `map` combinator (see Section 6.2.2), keeping `map` as a (lightweight) primitive enables more opportunities for optimizing computation DAGs (*e.g.*, operation fusion [Chambers et al., 2010b]).

send `def send(): Future[T]`

As mentioned earlier, the execution of computations built using SiloRefs is deferred. The `send` operation *forces* the lazy computation defined by the given SiloRef. Forcing is explicit in our model, because it requires sending the lineage to the remote node on which the result silo should be created. Given that network communication has a latency several orders of magnitude greater than accessing a word in main memory, providing an explicit send operation is a judicious choice [Waldo et al., 1996].

To enable materialization of remote silos to proceed concurrently, the `send` operation immediately returns a future [Haller et al., 2012]. This future is then asynchronously completed with the data of the given silo. Since calling `send` will materialize a silo and send its data to the current node, `send` should only be called on silos with reasonably small data (for example, in the implementation of an aggregate operation such as `reduce` on a distributed collection).

cache `def cache(): Future[Unit]`
The performance of typical data analytics jobs can be increased dramatically by caching large data sets in memory [Zaharia et al., 2012]. To do this, the silo containing the computed data set needs to be materialized. So far, the only way to materialize a silo that we have shown is using the `send` primitive. However, `send` additionally transfers the contents of a silo to the requesting node–too much if a large remote data set should merely be cached in memory remotely. Therefore, an additional primitive called `cache` is provided, which forces the materialization of the given SiloRef, returning `Future[Unit]`.

Given the running example so far, we can add another subgraph branching off of `adults`, which sorts each `Person` by age, produces a `String` gretting, and then "kicks-off" remote computation by calling `cache` and caching the result in remote memory (graphically shown in Figure 6.2, part 3 and 4):

```
val sorted =
  adults.map(spore { ps => ps.sortWith(p => p.age) })
val labels =
  sorted.map(spore { ps => ps.map(p => "Welcome, " + p.name) })
labels.cache()
```

Assuming we would also cache the `owners` SiloRef from the previous example, the resulting lineage graph would look as illustrated in Figure 6.2. Note that `vehicles` is not a regular parent in the lineage of `owners`; it is an indirect input used to compute `owners` by virtue of being *captured* by the spore used to compute `owners`.

Creating Silos

Besides a type definition for SiloRef, our framework also provides a companion singleton object (Scala's form of modules). The singleton object provides factory methods for obtaining SiloRefs referring to silos populated with some initial data:[3]

```
object SiloRef {
  def fromTextFile(host: Host)(file: File): SiloRef[List[String]]
  def fromFun[T](host: Host)(s: Spore[Unit, T]): SiloRef[T]
  def fromLineage[T](host: Host)(s: SiloRef[T]): SiloRef[T]
}
```

Each of the factory methods has a `host` parameter that specifies the target host (address/port) on which to create the silo. Note that the `fromFun` method takes a spore closure as an argument to make sure it can be serialized and sent to `host`. In each case, the returned SiloRef contains

[3] For clarity, only method signatures are shown.

its `host` as well as a host-unique identifier. The `fromLineage` method is particularly interesting as it creates a copy of a previously existing silo based on the lineage of a SiloRef `s`. Note that only the SiloRef is necessary for this operation to successfully complete; the silo originally hosting `s` might already have failed.

Expressiveness

Expressing `map` Leveraging the above-mentioned methods for creating silos, it is possible to express `map` in terms of `flatMap`:

```
def map[S](s: Spore[T, S]): SiloRef[S] =
  this.flatMap(spore {
    val localSpore = s
    (x: T) =>
      val res = localSpore(x)
      SiloRef.fromFun(currentHost)(spore {
        val localRes = res
        () => localRes
      })
  })
```

This should come as no surprise, given that `flatMap` is the monadic bind operation on SiloRefs, and `SiloRef.fromFun` is the monadic return operation. The reason why `map` is provided as one of the main operations of SiloRefs is that direct uses of `map` enable an important optimization based on operation fusion.

Expressing `cache` The `cache` operation can be expressed using `flatMap` and `send`:

```
def cache(): Future[Unit] = this.flatMap(spore {
 val localDoneSiloRef = DoneSiloRef
 res => localDoneSiloRef
}).send()
```

Here, we first use `flatMap` to create a new silo that will be completed with the trivial value of the `DoneSiloRef` singleton object (*e.g.*, `Unit`). Essentially, invoking `send` on this trivial SiloRef causes the resulting future to be completed as soon as `this` SiloRef has been materialized in memory.

```
val persons: SiloRef[List[Person]] = ...
val vehicles: SiloRef[List[Vehicle]] = ...
// copy of `vehicles` on different host `h`
val vehicles2 = SiloRef.fromFun(h)(spore {
  val localVehicles = vehicles
  () => localVehicles
})

val adults =
  persons.map(spore { ps => ps.filter(p => p.age >= 18) })

// adults that own a vehicle
def computeOwners(v: SiloRef[List[Vehicle]]) =
  spore {
    val localVehicles = v
    (ps: List[Person]) => localVehicles.map(...)
  }

val owners: SiloRef[List[(Person, Vehicle)]] =
  adults.flatMap(computeOwners(vehicles),
                 computeOwners(vehicles2))
```

6.2.3 Fault Handling

F-P includes overloaded variants of the primitives discussed so far which enable the definition of flexible fault handling semantics. The main idea is to specify fault handlers for *subgraphs of computation DAGs*. Our guiding principle is to make the definition of the failure-free path through a computation DAG as simple as possible, while still enabling the handling of faults at the fine-granular level of individual SiloRefs.

Defining fault handlers Fault handlers may be specified whenever the lineage of a SiloRef is extended. For this purpose, the introduced `map` and `flatMap` primitives are overloaded. For example, consider our previous example, but extended with a fault handler:

Importantly, in the `flatMap` call on the last line, in addition to `computeOwners(vehicles)`, the regular spore argument of `flatMap`, `computeOwners(vehicles2)` is passed as an additional argument. The second argument registers a *failure handler* for the subgraph of the computation DAG starting at `adults`. This means that if during the execution of `computeOwners(vehicles)` it is detected that the `vehicles` SiloRef has failed, it is checked whether the SiloRef that the higher-order combinator was invoked on (in this case, `adults`) has a failure handler registered. In that case, the failure handler is used as an alternative spore to compute the result of `adults.flatMap(..)`. In this example, we specified `computeOwners(vehicles2)` as the failure handler; thus, in case `vehicles` has failed, the computation is retried using `vehicles2`

instead.

6.3 Higher-Order Operations

The introduced primitives enable expressing surprisingly intricate computational patterns.

Higher-order operations such as variants of `map`, `reduce`, and `join`, operating on collections of data partitions, distributed across a set of hosts, are required when implementing abstractions like Spark's distributed collections [Zaharia et al., 2012]. Section 6.3.1 demonstrates the implementation of some such operations in terms of silos.

In addition, even more patterns are possible thanks to the decentralized nature of our programming model, which removes the limitations of master/worker host configurations. Section 6.3.2 shows examples of peer-to-peer patterns that are still fault-tolerant.

6.3.1 Higher-Order Operations

join Suppose we are given two silos with the following types:

```
val silo1: SiloRef[List[A]]
val silo2: SiloRef[List[B]]
```

as well as two hash functions computing hashes (of type `K`) for elements of type `A` and type `B`, respectively:

```
val hashA: A => K = ...
val hashB: B => K = ...
```

The goal is to compute the hash-join of `silo1` and `silo2` using a higher-order operation `hashJoin`:

```
def hashJoin[A, B, K](s1: SiloRef[List[A]],
                      s2: SiloRef[List[B]],
                      f: A => K,
                      g: B => K)
  : SiloRef[List[(K, (A, B))]] = ???
```

To implement `hashJoin` in terms of silos, the types of the two silos first have to be made equal, through initial `map` invocations:

```
val combined = s12.flatMap(spore {
  val localS22 = s22
  (triples1: List[(K, Option[A], Option[B])]) =>
    s22.map(spore {
      val localTriples1 = triples1
      (triples2: List[(K, Option[A], Option[B])]) =>
        localTriples1 ++ triples2
    })
})
```

```
val s12: SiloRef[List[(K, Option[A], Option[B])]] =
  s1.map(spore { l1 => l1.map(x => (f(x), Some(x), None)) })
val s22: SiloRef[List[(K, Option[A], Option[B])]] =
  s2.map(spore { l2 => l2.map(x => (g(x), None, Some(x))) })
```

Then, we can use `flatMap` to create a new silo which contains the elements of both silo `s12` and silo `s22`:

The combined silo contains triples of type `(K, Option[A], Option[B])`. Using an additional `map`, the collection can be sorted by key, and adjacent triples be combined, yielding a `SiloRef[List[(K, (A, B))]]` as required.

Partitioning and groupByKey A `groupByKey` operation on a group of silos containing collections needs to create multiple result silos, on each node, with ranges of keys supposed to be shipped to destination hosts. These destination hosts are determined using a partitioning function. Our goal, concretely:

```
val groupedSilos = groupByKey(silos)
```

Furthermore, we assume that `silos.size` = N where N is the number of hosts, with hosts h_1, h_2, etc. We assume each silo contains an unordered collection of key-value pairs (a multi-map). Then, `groupByKey` can be implemented as follows:

- Each host h_i applies a *partitioning function* (example: `hash(key) mod N`) to the key-value pairs in its silo, yielding N (local) silos.
- Using `flatMap`, each pair of silos containing keys of the same range can be combined and materialized on the right destination host.

Using just the primitives introduced earlier, applying the partitioning function in this way would require N `map` invocations per silo. Thus, the performance of `groupByKey` could be

```
object Utils {
  def aggregate(vs: SiloRef[List[Vehicle]],
                ps: SiloRef[List[Person]]): SiloRef[String] = ...
  def write(result: String, fileName: String): Unit = ...
}
val vehicles: SiloRef[List[Vehicle]] = ...
val persons:  SiloRef[List[Person]]  = ...
val info:     SiloRef[Info]          = ...
val fileName: String                 = "hdfs://..."
val done    = info.flatMap(spore {
  val localVehicles = vehicles
  val localPersons  = persons
  (localInfo: Info) =>
    aggregate(localVehicles, localPersons).map(spore {
      val in = localInfo
      res => combine(res, in)
    })
}).map(spore {
  val captured = fileName
  combined => Utils.write(combined, captured)
})
done.cache() // force computation
```

Figure 6.3 – Example of peer-to-peer style processing in F-P.

increased significantly using a specialized combinator, say, "mapPartition" that would apply a given partitioning function to each key-value pair, simultaneously populating N silos (where N is the number of "buckets" of the partitioning function).

6.3.2 Peer-to-Peer Patterns

To illustrate the decentralized nature of our model, consider the following example: the local host aggregates some data as soon as two silos `vehicles` and `persons` have been materialized. The aggregation result is then combined with a silo `info` on local host. The final result is written to a distributed file system, shown in Figure 6.3.

This program does not tolerate failures of the local host: if it fails before the computation is complete, the result is never written to the file. Using fault handlers, though, it is easy to introduce a backup host that takes over in case the local host fails at any point, as shown in Figure 6.4

First, the local variables `doCombine` and `doWrite` refer to the verbatim spores passed to `flatMap` and `map` above. Second, `backup` is a dummy silo on a backup host `hostb`. It is used to send a spore to the backup host in a way that allows it to detect whether the original

```
val doCombine = spore {
  val localVehicles = vehicles
  val localPersons  = persons
  (localInfo: Info) =>
    aggregate(localVehicles, localPersons).map(spore {
      val in = localInfo
      res => combine(res, in)
    })
}
val doWrite = spore {
  val captured = fileName
  combined => Utils.write(combined, captured)
}
val done      = info.flatMap(doCombine).map(doWrite)
val backup    = SiloRef.fromFun(hostb)(spore { () => true })
val recovered = backup.flatMap(
  spore {
    val localDone = done
    x => localDone
  },
  spore { // fault handler
    val localInfo      = info
    val localDoCombine = doCombine
    val localDoWrite   = doWrite
    val localHostb     = hostb
    x =>
      val restoredInfo = SiloRef.fromLineage(localHostb)(localInfo)
      restoredInfo.flatMap(localDoCombine).map(localDoWrite)
  }
)
done.cache()      // force computation on local host
recovered.cache() // force computation on backup host
```

Figure 6.4 – Using fault handlers to introduce a backup host in F-P.

host has failed. The fault handling is done by calling `flatMap` on `backup`, passing (a) a spore for the non-failure case (b) a spore for the failure case. The spore for the non-failure case simply returns the done SiloRef. The spore for the failure case is applied whenever the value of the done SiloRef could not be obtained. In this case, the lineage of the captured `info` SiloRef is used to restore its original contents in a new silo created on the backup host `hostb`. Its SiloRef is then used to retry the original computation. In case the original host failed only after the materialization of `vehicles` and `persons` completed, their cached data is reused.

$t ::= x$	variable
$\mid (x : T) \Rightarrow t$	abstraction
$\mid t\ t$	application
$\mid \texttt{let}\ x = t\ \texttt{in}\ t$	let binding
$\mid \{\overline{l = t}\}$	record construction
$\mid t.l$	selection
$\mid \texttt{spore}\ \{\ \overline{x : T = t}\ ;(x : T) \Rightarrow t\ \}$	spore
$\mid \texttt{map}(r, t[, t])$	map
$\mid \texttt{flatMap}(r, t[, t])$	flatMap
$\mid \texttt{send}(r)$	send
$\mid \texttt{await}(\iota)$	await future
$\mid r$	SiloRef
$\mid \iota$	future
$v ::= (x : T) \Rightarrow t$	abstraction
$\mid \{\overline{l = v}\}$	record value
$\mid p$	spore value
$\mid r$	SiloRef
$\mid \iota$	future
$p ::= \texttt{spore}\ \{\ \overline{x : T = v}\ ;(x : T) \Rightarrow t\ \}$	spore value
$T ::= T \Rightarrow T$	function type
$\mid \{\overline{l : T}\}$	record type
$\mid \mathscr{S}$	
$\mathscr{S} ::= T \Rightarrow T\ \{\ \texttt{type}\ \mathscr{C} = \overline{T}\ \}$	spore type
$\mid T \Rightarrow T\ \{\ \texttt{type}\ \mathscr{C}\ \}$	abstract spore type

Figure 6.5 – Core language syntax.

6.4 Formalization

We formalize our programming model in the context of a standard, typed lambda calculus with records. Figure 6.5 shows the syntax of our core language. Terms are standard except for the `spore`, `map`, `flatMap`, `send`, and `await` terms. A `spore` term creates a new spore. It contains a list of variable definitions (the spore header) and the spore's closure. A term `await`(ι) blocks execution until the future ι has been completed asynchronously. The `map`, `flatMap`, and `send` primitives have been discussed earlier.

6.4.1 Operational semantics

In the following we give a small-step operational semantics of the primitives of our language. The semantics is clearly stratified into a deterministic layer and a non-deterministic (concurrent) layer. Importantly, this means our programming model can benefit from existing

$h \in Hosts$
$i \in \mathbb{N}$

$\iota ::= (h, i)$	location
$r ::= \text{Mat}(\iota)$	materialized
$\mid \text{Mapped}(\iota, h, r, p, opt_f)$	lineage with `map`
$\mid \text{FMapped}(\iota, h, r, p, opt_f)$	lineage with `flatMap`
$E ::= \epsilon$	message queue
$\mid \text{Res}(\iota, v) :: E$	response
$\mid \text{Req}(h, r, \iota) :: E$	request
$\mid \text{ReqF}(h, r, \iota) :: E$	request (fault)

Figure 6.6 – Elements of the operational model.

$$\text{R-MAP}\quad \frac{host(r) = h' \quad i \text{ fresh} \qquad r' = \text{Mapped}((h, i), h', r, p, \text{None})}{(R[\texttt{map}(r, p)], E, S)^h \longrightarrow (R[r'], E, S)^h}$$

$$\text{R-FMAP}\quad \frac{host(r) = h' \quad i \text{ fresh} \qquad r' = \text{FMapped}((h, i), h', r, p, \text{None})}{(R[\texttt{flatMap}(r, p)], E, S)^h \longrightarrow (R[r'], E, S)^h}$$

$$\text{R-AWAIT}\quad \frac{S(\iota) = \text{Some}(v)}{(R[\texttt{await}(\iota)], E, S)^h \longrightarrow (R[v], E, S)^h} \qquad \text{R-RES}\quad \frac{E = \text{Res}(\iota, v) :: E' \quad S' = S + (\iota \mapsto v)}{(R[\texttt{await}(\iota_f)], E, S)^h \longrightarrow (R[\texttt{await}(\iota_f)], E', S')^h}$$

$$\text{R-REQLOCAL}\quad \frac{\begin{array}{c} E = \text{Req}(h', r, \iota'') :: E' \quad r = \text{Mapped}(\iota, h, r', p, \text{None}) \quad r' \neq \text{Mat}(\iota_s) \quad S(\iota) = \text{None} \\ loc(r') = \iota' \quad S(\iota') = \text{None} \qquad E'' = \text{Req}(h, r', \iota') :: E \end{array}}{(R[\texttt{await}(\iota_f)], E, S)^h \rightarrow (R[\texttt{await}(\iota_f)], E'', S)^h}$$

Figure 6.7 – Deterministic reduction.

reasoning techniques for sequential programs. Program transformations that are correct for sequential programs are also correct for distributed programs. Our programming model shares this property with some existing approaches such as [Peyton Jones et al., 1996].

Notation and conventions. We write $S' = S + (\iota \mapsto v)$ to express the fact that S' maps ι to v and otherwise agrees with S. We write $S(\iota) = \text{Some}(v)$ to express the fact that S maps ι to v. We write $S(\iota) = \text{None}$ if S does not have a mapping for ι. Reduction is defined using reduction contexts [Pierce, 2002]. We omit the definition of reduction contexts, since they are completely standard.

Configurations. The reduction rules of the deterministic layer define transitions of *host configurations* $(t, E, S)^h$ of host h where t is a term, E is a message queue, and S is a silo store. The reduction rules of the non-deterministic layer define transitions of sets H of host configurations. The reduced host configurations are chosen non-deterministically in order to express concurrency between hosts.

Fault handling. In the interest of clarity we present the reduction rules in two steps. In the first step we explain simplified rules without fault handling semantics (Sections 6.4.1 and 6.4.1). In the second step we explain how these simplified rules have to be refined in order to support the fault handling principles of our model (Section 6.4.2).

Decentralized identification

A important property of our programming model is the fact that silos are uniquely identified using *decentralized identifiers*. A decentralized identifier ι has two components: (a) the identifier of the host h that created ι, and (b) a name i created fresh on h (e.g., an integer value): $\iota = (h, i)$. Decentralized identifiers are important, since they reconcile two conflicting properties central to our model. The first property is building computation DAGs locally, without remote communication. This is possible using decentralized identifiers, since each host can generate new identifiers independently of other remote hosts. The second property is allowing SiloRefs to be freely copied between remote hosts. This is possible, since decentralized identifiers uniquely identify silos without the need for subsequent updates of their information; decentralized identifiers are immutable. This latter property is essential to enable computation DAGs that are *immutable upon construction*. In our programming model, computation DAGs are created using the standard monadic operations of SiloRefs. In particular, the `flatMap` operation (monadic bind) in general requires that its argument spore captures SiloRefs that are subsequently copied to a remote host. Hence it is essential that SiloRefs and the decentralized identifiers they contain be freely copyable between remote hosts.

Deterministic layer

We first consider the reduction rules of the deterministic layer shown in Figure 6.7. The reduction rules for `map` (R-MAP) and `flatMap` (R-FMAP) do not involve communication with other hosts. In each case, a new SiloRef r' is created that is derived from SiloRef r. The execution of the actual operation (`map` or `flatMap`, respectively) is deferred, and an object representing this derivation is returned. In both cases, the new SiloRef r' refers to a silo created on host h' by applying the spore value p to the value of silo r. The first component of the Mapped and FMapped objects, (h, i), is a fresh *location* created by host h to uniquely identify the result silo.

Most reduction rules are enabled when the current redex is an `await` term. The reduction of a term `await`(ι) only continues when store S maps location ι to value v. In all other cases, the current host removes the next message from its message queue E. As shown in Figure 6.6 there are two types of messages: requests (Req) and responses (Res). A response $\text{Res}(\iota, v)$ tells its receiver that the silo at location ι has value v. A request $\text{Req}(h, r, \iota)$ is sent on behalf of host h to request the value of silo r at location ι. The reception of a response $\text{Res}(\iota, v)$ is handled by adding a mapping $(\iota \mapsto v)$ to the store (rule R-Res). The reception of a request $\text{Req}(h', r, \iota'')$ is handled locally if materialization of the requested silo r is deferred and the parent silo r' in r's lineage has not been materialized either. In this case, the host sends a request to materialize r' to itself.

R-Send
$$\frac{host(r) = h' \quad h' \neq h \quad i \text{ fresh} \qquad \iota = (h, i) \quad m = \text{Req}(h, r, \iota)}{\{(R[\texttt{send}(r)], E, S)^h, (t, E', S')^{h'}\} \cup H \to \{(R[\iota], E, S)^h, (t, E' \cdot m, S')^{h'}\} \cup H}$$

R-Req1
$$\frac{E = \text{Req}(h', r, \iota') :: E' \quad r = \text{Mat}(\iota) \qquad S(\iota) = \text{Some}(v) \quad m = \text{Res}(\iota', v)}{\{(R[\texttt{await}(\iota_f)], E, S)^h, (t, E'', S')^{h'}\} \cup H \to \{(R[\texttt{await}(\iota_f)], E', S)^h, (t, E'' \cdot m, S')^{h'}\} \cup H}$$

R-Req2
$$\frac{\begin{array}{c} E = \text{Req}(h', r, \iota') :: E' \quad r = \text{Mapped}(\iota, h, r', p, \text{None}) \quad r' = \text{Mat}(\iota_s) \quad S(\iota) = \text{None} \\ S(\iota_s) = \text{Some}(v) \quad p(v) = v' \quad S' = S + (\iota \mapsto v') \quad m = \text{Res}(\iota', v') \end{array}}{\{(R[\texttt{await}(\iota_f)], E, S)^h, (t, E'', S'')^{h'}\} \cup H \to \{(R[\texttt{await}(\iota_f)], E', S')^h, (t, E'' \cdot m, S'')^{h'}\} \cup H}$$

R-Req3
$$\frac{\begin{array}{c} E = \text{Req}(h'', r, \iota'') :: E'' \quad r = \text{FMapped}(\iota, h, \text{Mat}(\iota_s), p, \text{None}) \quad S(\iota) = \text{None} \quad S(\iota_s) = \text{Some}(v) \\ p(v) = r' \quad loc(r') = \iota' \quad S(\iota') = \text{None} \quad host(r') = h' \quad m = \text{Req}(h, r', \iota') \quad E''' = \text{Req}(h'', r', \iota'') :: E'' \end{array}}{\{(R[\texttt{await}(\iota_f)], E, S)^h, (t, E', S')^{h'}\} \cup H \to \{(R[\texttt{await}(\iota_f)], E''', S)^h, (t, E' \cdot m, S')^{h'}\} \cup H}$$

R-Req4
$$\frac{\begin{array}{c} E = \text{Req}(h'', r, \iota'') :: E'' \quad r = \text{FMapped}(\iota, h, \text{Mat}(\iota_s), p, \text{None}) \quad S(\iota) = \text{None} \quad S(\iota_s) = \text{Some}(v) \\ p(v) = r' \quad loc(r') = \iota' \quad S(\iota') = \text{Some}(v') \quad S'' = S + (\iota \mapsto v') \quad m = \text{Res}(\iota'', v') \end{array}}{\{(R[\texttt{await}(\iota_f)], E, S)^h, (t, E', S')^{h''}\} \cup H \to \{(R[\texttt{await}(\iota_f)], E'', S'')^h, (t, E' \cdot m, S')^{h''}\} \cup H}$$

Figure 6.8 – Nondeterministic reduction.

Nondeterministic layer

All reduction rules in the nondeterministic layer, shown in Figure 6.8, involve communication between two hosts.

Reducing a term `send`(r) appends a request $\text{Req}(h, r, \iota)$ to the message queue of host h' of the requested silo r. In this case, host h creates a unique location $\iota = (h, i)$ to identify the silo subsequently. Rules R-Req1, R-Req2, and R-Req3 define the handling of request messages

that cannot be handled locally. If the request can be serviced immediately (R-REQ1), a response with the value v of the requested silo r is appended to the message queue of the requesting host h'. Rules R-REQ2 and R-REG3 handle cases where the requested silo is not already available in materialized form.

6.4.2 Fault handling

The key principles of the fault handling mechanism are:

- Whenever a message is sent to a non-local host h, it is checked whether h is alive; if it is not, any silos located on h are declared to have failed.
- Whenever the value of a silo r cannot be obtained due to another failed silo, r is declared to have failed.
- Whenever the failure of a silo r is detected, the nearest predecessor r' in r's lineage that is not located on the same host is determined. If r' has a fault handler f registered, the execution of f is requested. Otherwise, r' is declared to have failed.

These principles are embodied in the reduction as follows. First, we use the predicate failed(h) as a way to check whether it is possible to communicate with host h (e.g., an implementation could check whether it is possible to establish a socket connection). Second, failures of hosts are handled whenever communication is attempted: whenever a host h intends to send a message to a host h' where $h' \neq h$, it is checked whether failed(h'). If it is the case that failed(h'), either the corresponding location (silo or future) is declared as failed (and fault handling deferred), or a suitable fault handler is located and a recovery step is attempted. In the following we explain the extended reduction rules shown in Figure 6.9.

In rule RF-SEND, the host of the requested silo r is detected to have failed. However, the parent silos of r are all located on the same (failed) host. Thus, in this case silo r is simply declared as failed, and fault handling is delegated to other parts of the computation DAG that require the value of r (if any). Since `send` is essentially a "sink" of a DAG, no suitable fault handler can be located at this point.

This is different in rule RF-REQ4. Here, host h processes a message requesting silo r which is the result of a `flatMap` call. Materializing r requires obtaining the value of silo r', the result of applying spore p to the value v of the materialized parent r''. Importantly, if the host of r' is failed, it means the computation of the DAG defined by spore p did not result in a silo on an available host. Consequently, if the `flatMap` call deriving r specified a fault handler p_f, p_f is applied to v in order to recover from the failure. If the host of the resulting silo r_f is not failed, the original request for r is "modified" to request r_f instead. This is done by removing message Req(h'', r, ι'') from the message queue and prepending message Req(h'', r_f, ι''). Moreover, host h sends a message to itself, requesting the value of silo r_f.

$$\text{RF-SEND}\quad \frac{host(r) = h' \quad h' \neq h \quad \text{failed}(h') \qquad i \text{ fresh} \quad \iota = (h, i) \quad S'' = S + (\iota \mapsto \bot)}{\{(R[\texttt{send}(r)], E, S)^h\} \cup H \rightarrow \{(R[\iota], E, S'')^h\} \cup H}$$

$$\text{RF-REQ4}\quad \frac{\begin{array}{c} E = \text{Req}(h'', r, \iota'') :: E'' \quad r = \text{FMapped}(\iota, h, r'', p, \text{Some}(p_f)) \quad S(\iota) = \text{None} \\ loc(r'') = \iota_s \quad S(\iota_s) = \text{Some}(v) \quad p(v) = r' \quad \text{failed}(host(r')) \quad p_f(v) = r_f \quad host(r_f) = h_f \quad \neg\text{failed}(h_f) \\ loc(r_f) = \iota_f \quad S(\iota_f) = \text{None} \quad m = \text{Req}(h, r_f, \iota_f) \quad E''' = \text{Req}(h'', r_f, \iota'') :: E'' \end{array}}{\{(R[\texttt{await}(\iota_f)], E, S)^h, (t, E', S')^{h_f}\} \cup H \rightarrow \{(R[\texttt{await}(\iota_f)], E''', S)^h, (t, E' \cdot m, S')^{h_f}\} \cup H}$$

$$\text{RF-REQ5}\quad \frac{\begin{array}{c} E = \text{Req}(h'', r, \iota'') :: E' \quad r = \text{FMapped}(\iota, h, r'', p, \text{None}) \quad S(\iota) = \text{None} \quad loc(r'') = \iota_s \quad S(\iota_s) = \text{Some}(v) \\ p(v) = r' \quad \text{failed}(host(r')) \quad i_p, i_a \text{ fresh} \quad \iota_p = (h, i_p), \iota_a = (h, i_a) \quad m_p = \text{ReqF}(h, r'', \iota_p) \\ r_a = \text{FMapped}(\iota_a, h, \text{Mat}(\iota_p), p, \text{None}) \quad E'' = m_p :: \text{Req}(h'', r_a, \iota'') :: E' \end{array}}{(R[\texttt{await}(\iota_f)], E, S)^h \longrightarrow (R[\texttt{await}(\iota_f)], E'', S)^h}$$

$$\text{RF-REQF}\quad \frac{\begin{array}{c} E = \text{ReqF}(h, r, \iota') :: E' \quad r = \text{Mapped}(\iota, h, r'', p, \text{Some}(p_f)) \\ loc(r'') = \iota_s \quad S(\iota_s) = \text{Some}(v) \quad E'' = \text{Res}(\iota', p_f(v)) :: E' \end{array}}{(R[\texttt{await}(\iota_f)], E, S)^h \longrightarrow (R[\texttt{await}(\iota_f)], E'', S)^h}$$

Figure 6.9 – Fault handling.

Rule RF-REQ5 shows fault recovery in the case where the lineage of a requested silo does not specify a fault handler itself. In this case, host h creates two fresh locations ι_p, ι_a. ι_p is supposed to be eventually mapped to the result value of executing the fault handler of parent silo r''. Host h requests this value from itself using a special message $\text{ReqF}(h, r'', \iota_p)$. Finally, the original request for silo r in message queue E is replaced with a request for silo r_a. The silo r_a is created analogous to r, but using silo $\text{Mat}(\iota_p)$ as parent (eventually, location ι_p is mapped to the result of applying the parent's fault handler). As demonstrated by rule RF-REQF, ReqF messages used to request the application of the fault handler are handled in a way that is completely analogous to the way regular Req messages are handled, except that fault handlers p_f are applied as opposed to regular spores p.

6.5 Implementation

The presented programming model has been fully implemented in Scala, a functional programming language that runs on both JVMs and JavaScript runtimes. F-P is compiled and run using Scala 2.11.5, and considers only the JVM backend for now. Our implementation, which has been published as an open-source project,[4] builds on two main Scala extensions:

[4] https://github.com/heathermiller/f-p

- First, scala/pickling,[5] a type-safe and performant serialization library with an accompanying, optional macro extension that is focused on distributed programming. It is used for all serialization tasks. Our F-P implementation benefits from the maturity of Pickling, which supports pickling/unpickling a wide range of Scala type constructors. Pickling has evolved from a research prototype to a production-ready serialization framework that is now in widespread commercial use.
- Second, the programming model makes extensive use of spores, closure-like objects with explicit, typed environments. While previous work has reported on an empirical evaluation of spores, our presented programming model and implementation turned out to be an extensive validation of spores in the context of distributed programming. In addition, our implementation required a thorough refinement of the way spores are pickled.

So far, we have used our implementation to build a small Spark-like distributed collections abstraction, and example data analytics applications, such as word count and group-by-join pipelines. Our prototype has also served as an experimentation platform for type-based optimizations, which we present in more detail below.

6.5.1 Serialization in the presence of existential quantification

Initially, to serialize most message types exchanged by the network communication layer, runtime-based unpicklers had to be used (meaning unpickling code discovering the structure of a type through introspection at runtime). A major disadvantage of runtime-based unpickling is its significant impact on performance. The reason for its initial necessity was that message types are typically generic, but the generic type arguments are *existentially-quantified type variables* on the receiver's side. For example, the lineage of a SiloRef may contain instances of a type Mapped. This generic type has four type parameters. The receiver of a freshly unpickled Mapped instance typically uses a pattern match:

```
case mapped: Mapped[u, t, v, s] =>
```

The type arguments `u`, `t`, `v`, and `s` are *type variables*. While unknown, the static type of `mapped` is still useful for type-safety:

```
val newSilo = new LocalSilo[v, s](mapped.fun(value))
```

However, it is impossible to generate type-specific code to unpickle a type like `Mapped[u, t, v, s]`. As a solution to this problem we propose what we call "self-describing" pickles. Basically, the

[5] https://github.com/scala/pickling

idea is to augment the serialized representation with additional information about how to unpickle. The key is to capture the type-specific pickler and unpickler when the fully-concrete type of a Mapped instance is known:

```
def doPickle[T](msg: T)
  (implicit pickler: Pickler[T],
              unpickler: Unpickler[T]): Array[Byte] = ...
```

Essentially, this means when doPickle is called with a concrete type T, say:[6]

```
doPickle[Mapped[Int, List[Int], String, List[String]]](mapped)
```

not only a type-specific implicit pickler (a type class instance) is looked up, but also a type-specific implicit unpickler. The doPickle method can then build a self-describing pickle as follows. First, the actual message is pickled using the pickler, yielding a byte array. Then, an instance of the following simple record-like class is created:

```
case class SelfDescribing(blob: Array[Byte],
                          unpicklerClassName: String)
```

Besides the just produced byte array, it contains the class name of the type- specific unpickler. This enables, using this fully type-specific unpickler, even when the message type to be unpickled is only partially known. All that is required is an unpickler for type SelfDescribing. First, it reads the byte array and class name from the pickle. Second, it instantiates the type-specific unpickler reflectively using the class name. (Note that this is possible on both the JVM as well as on JavaScript runtimes using Scala's current JavaScript backend.) Finally, the unpickler is used to unpickle the byte array. In conclusion, this approach ensures (a) that a type that is pickleable using a type-specific pickler is guaranteed to be unpickleable by the receiver of the pickled SelfDescribing instance, and (b) that unpickling is as efficient as pickling, thanks to using type-specific unpicklers.

6.5.2 Type-based optimization of serialization

We have used our implementation to measure the impact of type-specific, compile-time-generated serializers (see above) on end-to-end application performance. In our benchmark application, a group of 4 silos is distributed across 4 different nodes/JVMs. Each silo is populated with a collection of "person" records. The application first transforms each silo using *map*, and then using *groupBy* and *join*. For the benchmark we measure the running time for a varying number of records.

[6] Note that the type arguments are inferred by the Scala compiler; they are only shown for clarity.

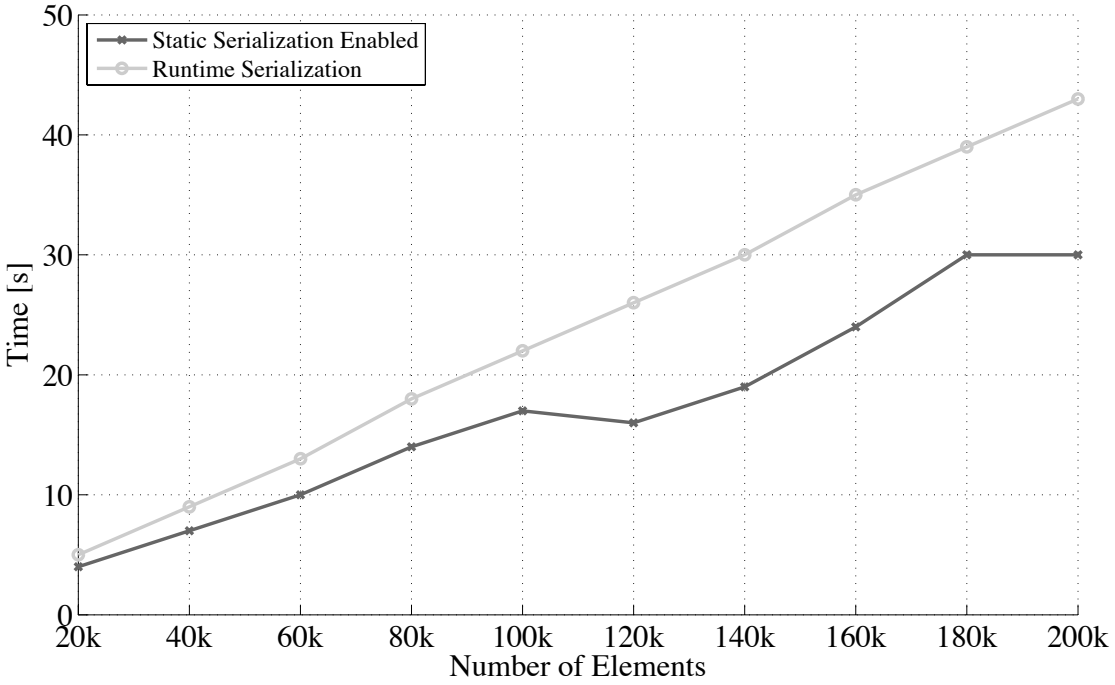


Figure 6.10 – Impact of Static Types on Performance, End-to-End Application (`groupBy` + `join`).

We ran our experiments on a 2.3 GHz Intel Core i7 with 16 GB RAM under Mac OS X 10.9.5 using Java HotSpot Server 1.8.0-b132. For each input size we report the median of 7 runs. Figure 6.10 shows the results. Interestingly, for an input size of 100,000 records, the use of type-specific serializers resulted in an overall speedup of about 48% with respect to the same system using runtime-based serializers.

6.6 Related Work

Alice ML [Rossberg et al., 2004] is an extension of Standard ML which adds a number of important features for distributed programming such as futures and proxies. The design leading up to F-P has incorporated many similar ideas, such as type-safe, generic and platform-independent pickling. In Alice, functions intend to be mobile. Only those functions which capture (either directly or indirectly) local resources remain stationary. In the case of functions that must remain stationary, it is possible to send proxies, mobile wrappers for functions. Sending a proxy will not transfer the wrapped function; instead, when a proxy function is applied, the call is forwarded by the system to the original site as a remote invocation (pickling arguments and result appropriately). In F-P, however, functions are not wrapped in proxies but sent directly. Thus, calling a received function will not lead to remote invocations.

Cloud Haskell [Epstein et al., 2011] leverages guaranteed-serializable, static closures for a message-passing communication model inspired by Erlang. In contrast, in our model spores

are sent between passive, persistent silos. Moreover, the coordination of concurrent activity is based on futures, instead of message passing. Closures and continuations in Termite Scheme [Germain, 2006] are always serializable; references to non- serializable objects (like open files) are automatically wrapped in processes that are serialized as their process ID. Similar to Cloud Haskell, Termite is inspired by Erlang. In contrast to Termite, F-P is statically typed, enabling advanced type-based optimizations. In non-process-oriented models, parallel closures [Matsakis, 2012] and RiverTrail [Herhut et al., 2013] address important safety issues of closures in a concurrent setting. However, RiverTrail currently does not support capturing variables in closures, which is critical for the flatMap combinator in F-P. In contrast to parallel closures, spores do not require a type system extension in Scala.

Acute ML [Sewell et al., 2005] is a dialect of ML which proposes numerous primitives for distributed programming, such as type-safe serialization, dynamic linking and rebinding, and versioning. F-P, in contrast, is based on spores, which ship with their serialized environment or they fail to compile, obviating the need for dynamic rebinding. HashCaml [Billings et al., 2006] is a practical evolution of Acute ML's ideas in the form of an extension to the OCaml bytecode compiler, which focuses on type-safe serialization and providing globally meaningful type names. In contrast, F-P merely a programming model, which does not require extensions to the Scala compiler.

ML5 [Murphy VII et al., 2007] provides mobile closures verified not to use resources not present on machines where they are applied. This property is enforced transitively (for all values reachable from captured values), which is stronger than what plain spores provide. However, type constraints allow spores to require properties not limited to mobility. Transitive properties are supported either using type constraints based on type classes which enforce a transitive property or by integrating with type systems that enforce transitive properties. Unlike ML5, spores do not require a type system extension. Further, the F-P model sits on top of these primitives to provide a full programming model for distribution, which also integrates spores and type-safe pickling.

Systems like Spark [Zaharia et al., 2012], MapReduce [Dean and Ghemawat, 2008], and Dryad [Isard et al., 2007] are distributed systems. Rather than being a system itself, F-P is meant to act as more of a substrate upon which to build systems like Spark, MapReduce, or Dryad. F-P aims to facilitate the design and implementation of such systems, and as a result provides much finer-grained control over details such as fault handling and network topology (*i.e.,* peer-to-peer vs master/worker).

The Clojure programming language proposes agents [Hickey, 2008]–stationary mutable data containers that users apply functions to in order to update an agent's state. F-P, in contrast, proposes that data in stationary containers be immutable, and that transformations by function application form a persistent data structure. Further, Clojure's agents are designed to manage state in a shared memory scenario, whereas F-P is designed with remote references for a distributed scenario.

The F-P model is also related to the actor model of concurrency [Agha, 1985], which features multiple implementations in Scala [Haller and Odersky, 2009, He et al., 2014, Typesafe, 2009]. Actors can serve as in-memory data containers in a distributed system, like our silos. Unlike silos, actors encapsulate behavior in addition to immutable or mutable values. While only some actor implementations support mobile actors (none in Scala), mobile behavior in the form of serializable closures is central to the F-P model.

6.7 Conclusion

We have presented F-P, a new programming model and principled substrate for building data-centric distributed systems. Built atop a foundation consisting of performant and type-safe serialization, and safe, serializable closures, we believe that it's possible to build elegant fault-tolerant functional systems. One insight of our model is that lineage-based fault recovery mechanisms, used in widespread frameworks for distribution, can be modeled elegantly in a functional way using persistent data structures. Our operational semantics shows that this approach makes it even amenable to formal treatment. We have also shown that F-P is able to express rich patterns of computation while maintaining fault-tolerance–such computation patterns include decentralized peer-to-peer patterns of communication. Finally, we have implemented our approach in and for Scala, and have discovered new ways to reconcile type-specific serializers with patterns of static typing common in distributed systems.

Conclusion

This thesis presented a number of extensions and libraries in and for Scala aimed at providing a more reliable foundation upon which to build distributed systems. Throughout, we have been concerned with two essential aspects of distribution: communication and concurrency.

First, we presented a new approach to communicate both objects and functions between distributed nodes safely and efficiently.

We began with objects; we saw scala/pickling, an approach for functionally composing serialization logic. Generation and composition of functionally-inspired object oriented picklers could be effectively generated and composed at compile time. This had the benefit of shifting the burden of serialization to compile time, allowing users to statically catch serialization errors while gaining performance through the static generation of performant serialization code. Scala/pickling has since become a popular open source library, and the go-to library for serialization in Scala; it has more than 630 stars and about 70 watchers on GitHub[7], and has been taken up by flagship Scala projects such as sbt, Scala's universal build tool.

We then moved on to functions; functions were made able to be communicated over the network through the introduction of spores, an abstraction that when combined with scala/pickling can provide extra static checking in order to ensure that closure is able to be reliably serialized. We also saw ways in which the accompanying spore type system was able to control specific hazards from being captured.

Second, we saw a novel lock-free concurrency abstraction suitable for building large-scale distributed systems. We covered FlowPools, an abstraction and backing data structure for non-blocking, fully asynchronous programming. We saw that FlowPools were provably deterministic, lock-free, and linearizable, in addition to having concrete performance benefits over comparable concurrent collections in Java's standard library.

Finally, we brought together our two approaches to communicate both objects and functions between distributed nodes safely and efficiently, pickling and spores, in the context of a new distributed programming model. Designed from the ground up using our new primitives for distribution, the model generalizes existing widely-used programming systems for data-intensive computing.

[7] Project repositories may be starred or watched. Starred indicates interest (akin to "liking" on a social network like Facebook) and users who "watch" subscribe to notifications of all project updates

A FlowPools, Proofs

A.1 Introduction

Implementing correct and deterministic parallel programs is challenging. Even though concurrency constructs exist in popular programming languages to facilitate the task of deterministic parallel programming, they are often too low level, or do not compose well due to underlying blocking mechanisms. In this appendix, we present the detailed proofs of the lock-freedom, and determinism properties of FlowPools, a deterministic concurrent dataflow abstraction presented in [Prokopec et al., 2012a]. The detailed proofs for linearizability and determinism can be found in the companion tech report [Prokopec et al., 2012b].

We first provide a summary of the lemmas and theorems introduced in the associated paper, *FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction* [Prokopec et al., 2012a]. We then cover definitions and invariants before moving on to our proof of lock-freedom.

We define the notion of an abstract pool $\mathbb{A} = (elems, callbacks, seal)$ of elements in the pool, callbacks and the seal size. Given an abstract pool, abstract pool operations produce a new abstract pool. The key to showing correctness is to show that an abstract pool operation corresponds to a FlowPool operation– that is, it produces a new abstract pool corresponding to the state of the FlowPool after the FlowPool operation has been completed.

Lemma A.1.1 Given a FlowPool consistent with some abstract pool, CAS instructions in lines 156, 198 and 201 do not change the corresponding abstract pool.

Lemma A.1.2 Given a FlowPool consistent with an abstract pool $(elems, cbs, seal)$, a successful CAS in line 157 changes it to the state consistent with an abstract pool $(\{elem\} \cup elems, cbs, seal)$. There exists a time $t_1 \geq t_0$ at which every callback $f \in cbs$ has been called on $elem$.

Lemma A.1.3 Given a FlowPool consistent with an abstract pool $(elems, cbs, seal)$, a successful CAS in line 259 changes it to the state consistent with an abstract pool $(elems, (f, \emptyset) \cup$

```
136  def create()
137    new FlowPool {
138      start = createBlock(0)
139      current = start
140    }
141
142  def createBlock(bidx: Int)
143    new Block {
144      array = new Array(BLOCKSIZE)
145      index = 0
146      blockindex = bidx
147      next = null
148    }
149
150  def append(elem: Elem)
151    b = READ(current)
152    idx = READ(b.index)
153    nexto = READ(b.array(idx + 1))
154    curo = READ(b.array(idx))
155    if check(b, idx, curo) {
156      if CAS(b.array(idx + 1), nexto, curo) {
157        if CAS(b.array(idx), curo, elem) {
158          WRITE(b.index, idx + 1)
159          invokeCallbacks(elem, curo)
160        } else append(elem)
161      } else append(elem)
162    } else {
163      advance()
164      append(elem)
165    }
166
167  def check(b: Block, idx: Int, curo: Object)
168    if idx > LASTELEMPOS return false
169    else curo match {
170      elem: Elem =>
171        return false
172      term: Terminal =>
173        if term.sealed = NOSEAL return true
174        else {
175          if totalElems(b, idx) < term.sealed
176            return true
177          else error("sealed")
178        }
179      null =>
180        error("unreachable")
181    }
182
183  def advance()
184    b = READ(current)
185    idx = READ(b.index)
186    if idx > LASTELEMPOS
187      expand(b, b.array(idx))
188    else {
189      obj = READ(b.array(idx))
190      if obj is Elem WRITE(b.index, idx + 1)
191    }
192
193  def expand(b: Block, t: Terminal)
194    nb = READ(b.next)
195    if nb is null {
196      nb = createBlock(b.blockindex + 1)
197      nb.array(0) = t
198      if CAS(b.next, null, nb)
199        expand(b, t)
200    } else {
201      CAS(current, b, nb)
202    }
203  def totalElems(b: Block, idx: Int)
204    return b.blockindex * (BLOCKSIZE - 1) + idx
205
206  def invokeCallbacks(e: Elem, term: Terminal)
207    for (f <- term.callbacks) future {
208      f(e)
209    }
210
211  def seal(size: Int)
212    b = READ(current)
213    idx = READ(b.index)
214    if idx <= LASTELEMPOS {
215      curo = READ(b.array(idx))
216      curo match {
217        term: Terminal =>
218          if ¬tryWriteSeal(term, b, idx, size)
219            seal(size)
220        elem: Elem =>
221          WRITE(b.index, idx + 1)
222          seal(size)
223        null =>
224          error("unreachable")
225      }
226    } else {
227      expand(b, b.array(idx))
228      seal(size)
229    }
230
231  def tryWriteSeal(term: Terminal, b: Block,
232    idx: Int, size: Int)
233    val total = totalElems(b, idx)
234    if total > size error("too many elements")
235    if term.sealed = NOSEAL {
236      nterm = new Terminal {
237        sealed = size
238        callbacks = term.callbacks
239      }
240      return CAS(b.array(idx), term, nterm)
241    } else if term.sealed ≠ size {
242      error("already sealed with different size")
243    } else return true
244
245  def foreach(f: Elem => Unit)
246    future {
247      asyncFor(f, start, 0)
248    }
249
250  def asyncFor(f: Elem => Unit, b: Block, idx: Int)
251    if idx <= LASTELEMPOS {
252      obj = READ(b.array(idx))
253      obj match {
254        term: Terminal =>
255          nterm = new Terminal {
256            sealed = term.sealed
257            callbacks = f ∪ term.callbacks
258          }
259          if ¬CAS(b.array(idx), term, nterm)
260            asyncFor(f, b, idx)
261        elem: Elem =>
262          f(elem)
263          asyncFor(f, b, idx + 1)
264        null =>
265          error("unreachable")
266      }
267    } else {
268      expand(b, b.array(idx))
269      asyncFor(f, b.next, 0)
270    }
```


Figure A.1 – FlowPool operations pseudocode

$t ::=$		terms	$p \in \{(vs, \sigma, cbs) \mid vs \subseteq Elem, \sigma \in \{-1\} \cup \mathbb{N},$
	`create` p	pool creation	$cbs \subset Elem \Rightarrow Unit\}$
	$p << v$	append	$v \in Elem$
	p `foreach` f	foreach	$f \in Elem \Rightarrow Unit$
	p `seal` n	seal	$n \in \mathbb{N}$
	t_1 ; t_2	sequence	

Figure A.2 – Syntax

$cbs, seal$) There exists a time $t_1 \geq t_0$ at which f has been called for every element in $elems$.

Lemma A.1.4 Given a FlowPool consistent with an abstract pool $(elems, cbs, seal)$, a successful CAS in line 240 changes it to the state consistent with an abstract pool $(elems, cbs, s)$, where either $seal = -1 \wedge s \in \mathbb{N}_0$ or $seal \in \mathbb{N}_0 \wedge s = seal$.

Theorem A.1.5 [Safety] FlowPool operations `append`, `foreach` and `seal` are consistent with the abstract pool semantics.

Theorem A.1.6 [Linearizable operations] FlowPool operations `append` and `seal` are linearizable.

Lemma A.1.7 After invoking a FlowPool operation `append`, `seal` or `foreach`, if a non-consistency changing CAS instruction in lines 156, 198, or 201 fails, they must have already been completed by another thread since the FlowPool operation began.

Lemma A.1.8 After invoking a FlowPool operation `append`, `seal` or `foreach`, if a consistency-changing CAS instruction in lines 157, 240, or 259 fails, then some thread has successfully completed a consistency changing CAS after some finite number of steps.

Lemma A.1.9 After invoking a FlowPool operation `append`, `seal` or `foreach`, a consistency changing instruction will be completed after a finite number of steps.

Theorem A.1.10 [Lock-freedom] FlowPool operations `append`, `foreach` and `seal` are lock-free.

A.2 Proof of Correctness

Definition A.2.1 [Data types] A **Block** b is an object which contains an array $b.array$, which itself can contain elements, $e \in Elem$, where **Elem** represents the type of e and can be any

countable set. A given block b additionally contains an index $b.index$ which represents an index location in $b.array$, a unique index identifying the array $b.blockIndex$, and $b.next$, a reference to a successor block c where $c.blockIndex = b.blockIndex + 1$. A **Terminal** $term$ is a sentinel object, which contains an integer $term.sealed \in \{-1\} \cup \mathbb{N}_0$, and $term.callbacks$, a set of functions $f \in Elem \Rightarrow Unit$.

We define the following functions:

$$following(b : Block) = \begin{cases} \emptyset & \text{if b.next = null,} \\ b.next \cup following(b.next) & \text{otherwise} \end{cases}$$

$$reachable(b : Block) = \{b\} \cup following(b)$$

$$last(b : Block) = b' : b' \in reachable(b) \wedge b'.next = null$$

$$size(b : Block) = |\{x : x \in b.array \wedge x \in Elem\}|$$

Based on them we define the following relation:

$$reachable(b, c) \Leftrightarrow c \in reachable(b)$$

Definition A.2.2 [FlowPool] A **FlowPool** $pool$ is an object that has a reference $pool.start$, to the first block b_0 (with $b_0.blockIndex = 0$), as well as a reference $pool.current$. We sometimes refer to these just as $start$ and $current$, respectively.

A **scheduled callback invocation** is a pair (f, e) of a function $f \in Elem => Unit$ and an element $e \in Elem$. The programming construct that adds such a pair to the set of $futures$ is `future { f(e) }`.

The **FlowPool state** is defined as a pair of the directed graph of objects transitively reachable from the reference $start$ and the set of scheduled callback invocations called $futures$.

A **state changing** or **destructive** instruction is any atomic write or CAS instruction that changes the FlowPool state.

We say that the FlowPool **has an element** e at some time t_0 if and only if the relation $hasElem(start, e)$ holds.

$$hasElem(start, e) \Leftrightarrow \exists b \in reachable(start), e \in b.array$$

We say that the FlowPool **has a callback** f at some time t_0 if and only if the relation $hasCallback(start, f)$ holds.

$$hasCallback(start, f) \Leftrightarrow \quad \forall b = last(start), b.array = x^P \cdot t \cdot y^N, x \in Elem, \\ t = Terminal(seal, callbacks), f \in callbacks$$

We say that a callback f in a FlowPool **will be called** for the element e at some time t_0 if and only if the relation $willBeCalled(start, e, f)$ holds.

$$willBeCalled(start, e, f) \Leftrightarrow \exists t_1, \forall t > t_1, (f, e) \in futures$$

We say that the FlowPool is **sealed** at the size s at some t_0 if and only if the relation $sealedAt(start, s)$ holds.

$$sealedAt(start, s) \Leftrightarrow \quad s \neq -1 \wedge \forall b = last(start), b.array = x^P \cdot t \cdot y^N, \\ x \in Elem, t = Terminal(s, callbacks)$$

FlowPool operations are `append`, `foreach` and `seal`, and are defined by pseudocodes in Figure A.1.

Definition A.2.3 [Invariants] We define the following invariants for the **FlowPool**:

INV1 $start = b : Block, b \neq null, current \in reachable(start)$

INV2 $\forall b \in reachable(start), b \notin following(b)$

INV3 $\forall b \in reachable(start), b \neq last(start) \Rightarrow size(b) = LASTELEMPOS \wedge b.array(BLOCKSIZE - 1) \in Terminal$

INV4 $\forall b = last(start), b.array = p \cdot c \cdot n$, where:

$p = X^P, c = c_1 \cdot c_2, n = null^N$

$x \in Elem, c_1 \in Terminal, c_2 \in \{null\} \cup Terminal$

$P + N + 2 = BLOCKSIZE$

INV5 $\forall b \in reachable(start), b.index > 0 \Rightarrow b.array(b.index - 1) \in Elem$

Definition A.2.4 [Validity] A FlowPool state $\mathbb{S}$ is **valid** if and only if the invariants [INV1-5] hold for that state.

Definition A.2.5 [Abstract pool] An **abstract pool** $\mathbb{P}$ is a function from time t to a tuple $(elems, callbacks, seal)$ such that:

$seal \in \{-1\} \cup \mathbb{N}_0$

$callbacks \subset \{(f : Elem => Unit, called)\}$

$called \subseteq elems \subseteq Elem$

We say that an abstract pool $\mathbb{P}$ **is in state** $\mathbb{A} = (elems, callbacks, seal)$ at time t if and only if $\mathbb{P}(t) = (elems, callbacks, seal)$.

Definition A.2.6 [Abstract pool operations] We say that an **abstract pool operation** op that is applied to some abstract pool $\mathbb{P}$ in abstract state $\mathbb{A}_0 = (elems_0, callbacks_0, seal_0)$ at some time t **changes** the abstract state of the abstract pool to $\mathbb{A} = (elems, callbacks, seal)$ if $\exists t_0, \forall \tau, t_0 < \tau < t, \mathbb{P}(\tau) = \mathbb{A}_0$ and $\mathbb{P}(t) = \mathbb{A}$. We denote this as $\mathbb{A} = op(\mathbb{A}_0)$.

Abstract pool operation $foreach(f)$ changes the abstract state at t_0 from $(elems, callbacks, seal)$ to $(elems, (f, \emptyset) \cup callbacks, seal)$. Furthermore:

$$\exists t_1 \geq t_0, \quad \forall t_2 > t_1, \mathbb{P}(t_2) = (elems_2, callbacks_2, seal_2)$$
$$\wedge \forall (f, called_2) \in callbacks_2, elems \subseteq called_2 \subseteq elems_2$$

Abstract pool operation $append(e)$ changes the abstract state at t_0 from $(elems, callbacks, seal)$ to $(\{e\} \cup elems, callbacks, seal)$. Furthermore:

$$\exists t_1 \geq t_0, \quad \forall t_2 > t_1, \mathbb{P}(t_2) = (elems_2, callbacks_2, seal_2)$$
$$\wedge \forall (f, called_2) \in callbacks_2, (f, called) \in callbacks \Rightarrow e \in called_2$$

Abstract pool operation $seal(s)$ changes the abstract state of the FlowPool at t_0 from $(elems, callbacks, seal)$ to $(elems, callbacks, s)$, assuming that $seal \in \{-1\} \cup \{s\}$ and $s \in \mathbb{N}_0$, and $|elems| \leq s$.

Definition A.2.7 [Consistency] A FlowPool state $\mathbb{S}$ is **consistent** with an abstract pool $\mathbb{P} = (elems, callbacks, seal)$ at t_0 if and only if $\mathbb{S}$ is a valid state and:

$\forall e \in Elem, hasElem(start, e) \Leftrightarrow e \in elems$

$\forall f \in Elem => Unit, hasCallback(start, f) \Leftrightarrow f \in callbacks$

$\forall f \in Elem => Unit, \forall e \in Elem, willBeCalled(start, e, f) \Leftrightarrow \exists t_1 \geq t_0, \forall t_2 > t_1, \mathbb{P}(t_2) = (elems_2, (f, called_2) \cup callbacks_2, seal_2), elems \subseteq called_2$

$\forall s \in \mathbb{N}_0, sealedAt(start, s) \Leftrightarrow s = seal$

A FlowPool operation op is **consistent** with the corresponding abstract state operation op' if and only if $\mathbb{S}' = op(\mathbb{S})$ is consistent with an abstract state $\mathbb{A}' = op'(\mathbb{A})$.

A **consistency change** is a change from state $\mathbb{S}$ to state $\mathbb{S}'$ such that $\mathbb{S}$ is consistent with an abstract state $\mathbb{A}$ and $\mathbb{S}'$ is consistent with an abstract set $\mathbb{A}'$, where $\mathbb{A} \neq \mathbb{A}'$.

Proposition A.2.8 Every valid state is consistent with some abstract pool.

Definition A.2.9 [Lock-freedom] In a scenario where some finite number of threads are executing a concurrent operation, that concurrent operation is *lock-free* if and only if that concurrent operation is completed after a finite number of steps by some thread.

Theorem A.2.10 [Lock-freedom] FlowPool operations `append`, `seal`, and `foreach` are lock-free.

We begin by first proving that there are a finite number of execution steps before a consistency change occurs.

By Lemma A.2.15, after invoking `append`, a consistency change occurs after a finite number of steps. Likewise, by Lemma A.2.18, after invoking `seal`, a consistency change occurs after a finite number of steps. And finally, by Lemma A.2.19, after invoking `foreach`, a consistency change likewise occurs after a finite number of steps.

By Lemma A.2.20, this means a concurrent operation `append`, `seal`, or `foreach` will successfully complete. Therefore, by Definition A.2.9, these operations are lock-free.

Note. For the sake of clarity in this section of the correctness proof, we assign the following aliases to the following CAS and WRITE instructions:

- $CAS_{append-out}$ corresponds to the outer CAS in `append`, on line 156.
- $CAS_{append-inn}$ corresponds to the inner CAS in `append`, on line 157.
- $CAS_{expand-nxt}$ corresponds to the CAS on $next$ in `expand`, line 198.
- $CAS_{expand-curr}$ corresponds to the CAS on $current$ in `expand`, line 201.
- CAS_{seal} corresponds to the CAS on the $Terminal$ in `tryWriteSeal`, line 240.
- $CAS_{foreach}$ corresponds to the CAS on the $Terminal$ in `asyncFor`, line 259.
- $WRITE_{app}$ corresponds to the WRITE on the new $index$ in `append`, line 158.
- $WRITE_{adv}$ corresponds to the WRITE on the new $index$ in `advance`, line 190.
- $WRITE_{seal}$ corresponds to the WRITE on the new $index$ in `seal`, line 221.

Lemma A.2.11 After invoking an operation op, if non-consistency changing CAS operations $CAS_{append-out}$, $CAS_{expand-nxt}$, or $CAS_{expand-curr}$, in the pseudocode fail, they must have already been successfully completed by another thread since op began.

Proof: Trivial inspection of the pseudocode reveals that since $CAS_{append-out}$ makes up a check that precedes $CAS_{append-inn}$, and since $CAS_{append-inn}$ is the only operation besides $CAS_{append-out}$ which can change the expected value of $CAS_{append-out}$, in the case of a failure of $CAS_{append-out}$, $CAS_{append-inn}$ (and thus $CAS_{append-out}$) must have already successfully completed or $CAS_{append-out}$ must have already successfully completed by a different thread since op began executing.

Likewise, by trivial inspection $CAS_{expand-nxt}$ is the only CAS which can update the $b.next$ reference, therefore in the case of a failure, some other thread must have already successfully completed $CAS_{expand-nxt}$ since the beginning of op.

Like above, $CAS_{expand-curr}$ is the only CAS which can change the $current$ reference, therefore in the case of a failure, some other thread must have already successfully completed $CAS_{expand-curr}$ since op began. □

Lemma A.2.12 [Expand] Invoking the $expand$ operation will execute a non- consistency changing instruction after a finite number of steps. Moreover, it is guaranteed that the $current$ reference is updated to point to a subsequent block after a finite number of steps. Finally, $expand$ will return after a finite number of steps

Proof:

From inspection of the pseudocode, it is clear that the only point at which $expand(b)$ can be invoked is under the condition that for some block b, $b.index > LASTELEMPOS$, where $LASTELEMPOS$ is the maximum size set aside for elements of type $Elem$ in any block. Given this, we will proceed by showing that a new block will be created with all related references $b.next$ and $current$ correctly set.

There are two conditions under which a non-consistency changing CAS instruction will be carried out.

- **Case 1:** if $b.next = null$, a new block nb will be created and $CAS_{expand-nxt}$ will be executed. From Lemma A.2.11, we know that $CAS_{expand-nxt}$ must complete successfully on some thread. Afterwards recursively calling $expand$ on the original block b.
- **Case 2:** if $b.next \neq null$, $CAS_{expand-curr}$ will be executed. Lemma A.2.11 guarantees that $CAS_{expand-curr}$ will update $current$ to refer to $b.next$, which we will show can only be a new block. Likewise, Lemma A.2.11 has shown that $CAS_{expand-nxt}$ is the only state changing instruction that can initiate a state change at location $b.next$, therefore,

since $CAS_{expand-nxt}$ takes place within Case 1, Case 2 can only be reachable after Case 1 has been executed successfully. Given that Case 1 always creates a new block, therefore, $b.next$ in this case, must always refer to a new block.

Therefore, since from Lemma A.2.11 we know that both $CAS_{expand-nxt}$ and $CAS_{expand-curr}$ can only fail if already completed guaranteeing their finite completion, and since $CAS_{expand-nxt}$ and $CAS_{expand-curr}$ are the only state changing operations invoked through $expand$, the $expand$ operation must complete in a finite number of steps.

Finally, since we saw in Case 2 that a new block is always created and related references are always correctly set, that is both $b.next$ and $current$ are correctly updated to refer to the new block, it follows that $numBlocks$ strictly increases after some finite number of steps. □

Lemma A.2.13 [$CAS_{append-inn}$] After invoking `append(elem)`, if $CAS_{append-inn}$ fails, then some thread has successfully completed $CAS_{append-inn}$ or CAS_{seal} (or likewise, $CAS_{foreach}$) after some finite number of steps.

Proof: First, we show that a thread attempting to complete $CAS_{append-inn}$ can't fail due to a different thread completing $CAS_{append-out}$ so long as `seal` has not been invoked after completing the read of $currobj$. We address this exception later on.

Since after $check$, the only condition under which $CAS_{append-out}$, and by extension, $CAS_{append-inn}$ can be executed is the situation where the current object $currobj$ with index location idx is the $Terminal$ object, it follows that $CAS_{append-out}$ can only ever serve to duplicate this $Terminal$ object at location $idx+1$, leaving at most two $Terminal$s in block refered to by $current$ momentarily until $CAS_{append-inn}$ can be executed. By Lemma A.2.11, since $CAS_{append-out}$ is a non-consistency changing instruction, it follows that any thread holding any element $elem'$ can execute this instruction without changing the expected value of $currobj$ in $CAS_{append-inn}$, as no new object is ever created and placed in location idx. Therefore, $CAS_{append-inn}$ cannot fail due to $CAS_{append-out}$, so long as `seal` has not been invoked by some other thread after the read of $currobj$.

This leaves only two scenarios in which consistency changing $CAS_{append-inn}$ can fail:

- **Case 1:** Another thread has already completed $CAS_{append-inn}$ with a different element $elem'$.
- **Case 2:** Another thread completes an invocation to the `seal` operation after the current thread completes the read of $currobj$. In this case, $CAS_{append-inn}$ can fail because CAS_{seal} (or, likewise $CAS_{foreach}$) might have completed before, in which case, it inserts a new $Terminal$ object $term$ into location idx (in the case of a `seal` invocation, $term.sealed \in \mathbb{N}_0$, or in the case of a `foreach` invocation, $term.callbacks \in \{Elem \Rightarrow Unit\}$).

We omit the proof and detailed discussion of $CAS_{foreach}$ because it can be proven using the same steps as were taken for CAS_{seal}. □

Lemma A.2.14 [Finite Steps Before State Change] All operations with the exception of append, `seal`, and `foreach` execute only a finite number of steps between each state changing instruction.

Proof: The advance, check, `totalElems`, `invokeCallbacks`, and `tryWriteSeal` operations have a finite number of execution steps, as they contain no recursive calls, loops, or other possibility to restart.

While the expand operation contains a recursive call following a CAS instruction, it was shown in Lemma A.2.12 that an invocation of expand is guaranteed to execute a state changing instruction after a finite number of steps.

□

Lemma A.2.15 [Append] After invoking `append(elem)`, a consistency changing instruction will be completed after a finite number of steps.

Proof: The append operation can be restarted in three cases. We show that in each case, it's guaranteed to either complete in a finite number of steps, or leads to a state changing instruction:

- **Case 1:** The call to check, a finite operation by Lemma A.2.14, returns $false$, causing a call to advance, also a finite operation by Lemma A.2.14, followed by a recursive call to append with the same element $elem$ which in turn once again calls check.

  We show that after a finite number of steps, the check will evaluate to $true$, or some other thread will have completed a consistency changing operation since the initial invocation of append. In the case where check evaluates to $true$, Lemma A.2.13 applies, as it guarantees that a consistency changing CAS is completed after a finite number of steps.

  When the call to the finite operation check returns $false$, if the subsequent advance finds that a $Terminal$ object is at the current block index idx, then the next invocation of append will evaluate check to $true$. Otherwise, it must be the case that another thread has moved the Terminal to a subsequent index since the initial invocation of append, which is only possible using a consistency changing instruction.

  Finally, if advance finds that the element at idx is an $Elem$, $b.index$ will be incremented after a finite number of steps. By $INV1$, this can only happen a finite number of times until a $Terminal$ is found. In the case that expand is meanwhile invoked through

advance, by Lemma A.2.12 it's guaranteed to complete state changing instructions $CAS_{expand-nxt}$ or $CAS_{expand-curr}$ in a finite number of steps. Otherwise, some other thread has moved the $Terminal$ to a subsequent index. However, this latter case is only possible by successfully completing $CAS_{append-inn}$, a consistency changing instruction, after the initial invocation of append.

- **Case 2:** $CAS_{append-out}$ fails, which we know from Lemma A.2.11means that it must've already been completed by another thread, guaranteeing that $CAS_{append-inn}$ will be attempted. If $CAS_{append-inn}$ fails, after a finite number of steps, a consistency changing instruction will be completed. If $CAS_{append-inn}$ succeeds, as a consistency changing instruction, consistency will have clearly been changed.
- **Case 3:** $CAS_{append-inn}$ fails, which, by Lemma A.2.13, indicates that either some other thread has already completed $CAS_{append-inn}$ with another element, or another consistency changing instruction, CAS_{seal} or $CAS_{foreach}$ has successfully completed.

Therefore, append itself as well as all other operations reachable via an invocation of append are guaranteed to have a finite number of steps between *consistency* changing instructions. □

Lemma A.2.16 [CAS_{seal}] After invoking `seal(size)`, if CAS_{seal} fails, then some thread has successfully completed CAS_{seal} or $CAS_{append-inn}$ after some finite number of steps.

Proof: Since by Lemma A.2.13, we know that $CAS_{append-out}$ only duplicates an existing $Terminal$, it can not be the cause for a failing CAS_{seal}. This leaves only two cases in which CAS_{seal} can fail:

- **Case 1:** Another thread has already completed CAS_{seal}.
- **Case 2:** Another thread completes an invocation to the $append(elem)$ operation after the current thread completes the read of $currobj$. In this case, CAS_{seal} can fail because $CAS_{append-inn}$ might have completed before, in which case, it inserts a new $Elem$ object $elem$ into location idx. □

Lemma A.2.17 [$WRITE_{adv}$ and $WRITE_{seal}$] After updating $b.index$ using $WRITE_{adv}$ or $WRITE_{seal}$, $b.index$ is guaranteed to be incremented after a finite number of steps.

Proof: For some index, idx, both calls to $WRITE_{adv}$ and $WRITE_{seal}$ attempt to write $idx+1$ to $b.index$. In both cases, it's possible that another thread could complete either $WRITE_{adv}$ or $WRITE_{seal}$, once again writing idx to $b.index$ after the current thread has completed, in effect overwriting the current thread's write with $idx+1$. By inspection of the pseudocode, both $WRITE_{adv}$ and $WRITE_{seal}$ will be repeated if $b.index$ has not been incremented. However,

since the number of threads operating on the FlowPool is finite, p, we are guaranteed that in the worst case, this scenario can repeat at most p times, before a write correctly updates $b.index$ with $idx+1$. □

Lemma A.2.18 [Finite Steps Before Consistency Change] After invoking `seal(size)`, a consistency changing instruction will be completed after a finite number of steps, or the initial invocation of `seal(size)` completes.

Proof: The `seal` operation can be restarted in two scenarios.

- **Case 1:** The check $idx \leq LASTELEMPOS$ succeeds, indicating that we are at a valid location in the current block b, but the object at the current index location idx is of type $Elem$, not $Terminal$, causing a recursive call to `seal` with the same size $size$.

  In this case, we begin by showing that the atomic write of $idx+1$ to $b.index$, required to iterate through the block b for the recursive call to `seal`, will be correctly incremented after a finite number of steps.

  Therefore, by both the guarantee that, in a finite number of steps, $b.index$ will eventually be correctly incremented as we saw in Lemma A.2.17, as well as by $INV1$ we know that the original invocation of `seal` will correctly iterate through b until a $Terminal$ is found. Thus, we know that the call to `tryWriteSeal` will be invoked, and by both Lemma A.2.14 and Lemma A.2.15, we know that either `tryWriteSeal`, will successfully complete in a finite number of steps, in turn successfully completing `seal(size)`, or $CAS_{append-inn}$, another consistency changing operation will successfully complete.

- **Case 2:** The check $idx \leq LASTELEMPOS$ fails, indicating that we must move on to the next block, causing first a call to `expand` followed by a recursive call to `seal` with the same size $size$.

  We proceed by showing that after a finite number of steps, we must end up in Case 1, which we have just showed itself completes in a finite number of steps, or that a consistency change must've already occurred.

  By Lemma A.2.12, we know that an invocation of `expand` returns after a finite number of steps, and $pool.current$ is updated to point to a subsequent block.

  If we are in the recursive call to `seal`, and the $idx \leq LASTELEMPOS$ condition is $false$, trivally, a consistency changing operation must have occurred, as, the only way for the condition to evaluate to $true$ is through a consistency changing operation, in the case that a block has been created during an invocation to `append`, for example.

  Otherwise, if we are in the recursive call to `seal`, and the $idx \leq LASTELEMPOS$ condition evaluates to $true$, we enter Case 1, which we just showed will successfully complete in a finite number of steps.

□

Lemma A.2.19 [Foreach] After invoking `foreach(fun)`, a consistency changing instruction will be completed after a finite number of steps.

We omit the proof for `foreach` since it proceeds in the exactly the same way as does the proof for `seal` in Lemma A.2.18.

Lemma A.2.20 Assume some concurrent operation is started. If some thread completes a consistency changing CAS instruction, then some concurrent operation is guaranteed to be completed.

Proof:

By trival inspection of the pseudocode, if $CAS_{append-inn}$ successfully completes on some thread, then that thread is guaranteed to complete the corresponding invocation of append in a finite number of steps.

Likewise by trivial inspection, if CAS_{seal} successfully completes on some thread, then by Lemma A.2.14, `tryWriteSeal` is guaranteed to complete in a finite number of steps, and therefore, that thread is guaranteed to complete the corresponding invocation of `seal` in a finite number of steps.

The case for $CAS_{foreach}$ is omitted since it follows the same steps as for the case of CAS_{seal} □

B Spores, Formally

B.1 Overview

Spores are designed to avoid problems of closures. This is done using two mechanisms: the spore shape and context bounds for the spore's environment.

A spore is a closure with a specific shape that dictates how the environment of a spore is declared. In general, a spore has the following shape:

```
spore {
  val y1: S1 = <expr1>
  ...
  val yn: Sn = <exprn>
  (x: T) => {
    // ...
  }
}
```

A spore consists of two parts: the header and the body. The list of value definitions at the beginning is called the spore header. The header is followed by a regular closure, the spore's body. The characteristic property of a spore is that the body of its closure is only allowed to access its parameter, values in the spore header, as well as top-level singleton objects (public, global state). In particular, the spore closure is not allowed to capture variables in the environment. Only an expression on the right-hand side of a value definition in the spore header is allowed to capture variables.

By enforcing this shape, the environment of a spore is always declared explicitly in the spore header which avoids accidentally capturing problematic references. Moreover, and that's important for OO languages, it's no longer possible to accidentally capture the "this" reference.

Note that the evaluation semantics of a spore is equivalent to a closure obtained by leaving

out the "spore" marker:

```
{
  val y1: S1 = <expr1>
  ...
  val yn: Sn = <exprn>
  (x: T) => {
    // ...
  }
}
```

In Scala, the above block first initializes all value definitions in order and then evaluates to a closure that captures the introduced local variables y1, ..., yn. The corresponding spore has the exact same evaluation semantics. What's interesting is that this closure shape is already used in production systems such as Spark to avoid problems with accidentally captured "this" references. However, in these systems the above shape is not enforced, whereas with spores it is.

The result type of the "spore" constructor is not a regular function type, but a subtype of one of Scala's function types. This is possible, because in Scala functions are instances of classes that mix in one of the function traits. For example, the trait for functions of arity one looks like this:[1]

```
trait Function1[-A, +B] {
  def apply(x: A):  B
}
```

The `apply` method is abstract; a concrete implementation applies the body of the function that's being defined to the argument `x`. Functions are contravariant in their argument type A, indicated using the "-" symbol, and covariant in their result type B, indicated using the "+" symbol.

The type of a spore of arity one is a subtype of `Function1`:

```
trait Spore[-A, +B] extends Function1[A, B]
```

Using the `Spore` trait methods can require argument closures to be spores:

```
def sendOverWire(s: Spore[Int, Int]): Unit = ...
```

[1] For simplicity we omit definitions of the 'andThen' and 'compose' methods in the definition of 'Function1'.

This way, libraries and frameworks can enforce the use of spores instead of plain closures, thereby reducing the risk for common programming errors.

B.1.1 Context bounds

The fact that for spores a certain shape is enforced is very useful. However, in some situations this is not enough. For example, using closures in a concurrent setting is very error-prone, because of the fact that it's possible to capture mutable objects which leads to race conditions. Thus, closures should only capture immutable objects to avoid interference. However, such constraints cannot be enforced using the spore shape alone (captured objects are stored in constant values in the spore header, but such a constant might still refer to a mutable object).

In this section we introduce a form of type-based constraints called "context bounds" that can be attached to a spore which enforce certain type-based properties for all captured variables of a spore.

Taking another example, it might be necessary for a spore to require the availability of instances of a certain type class for the types of all its captured variables. A typical example for such a type class is Pickler: types with an instance of the Pickler type class can be pickled using a new pickling framework for Scala. To be able to pickle a spore, it's necessary that all its captured types have an instance of Pickler.[2]

Spores allow expressing such a requirement using implicit properties. The idea is that if there is an implicit of type `Property[Pickler]` in scope at the point where a spore is created, then it is enforced that all captured types in the spore header have an instance of the Pickler type class:

```
import spores.withPickler

spore {
  val name: String = <expr1>
  val age: Int = <expr2>
  (x: String) => {
    // ...
  }
}
```

While an imported property does not have an impact on how a spore is constructed (besides the property import), it has an impact on the result type of the spore macro. In the above example, the result type would be a refinement of the Spore type:

[2] A spore can be pickled by pickling its environment and the fully-qualified class name of its corresponding function class.

```
Spore[String, Int] {
  type Captured = (String, Int)
  val captured: Captured
  implicit val p$1 = implicitly[Pickler[(String, Int)]]
  (x: String) => {
    // ...
  }
}
```

The refinement type contains a type member `Captured` which is defined to be a tuple of all the captured types. The values of the actual captured variables are accessible using the captured value member. What's more, the refinement type contains for each type class that's required an implicit value with a type class instance for type `Captured`.

Such implicit values allow retrieving a type class instance for the captured types of a given spore using Scala's implicitly function as follows:

```
val s = spore { ... }

implicitly[Pickler[s.Captured]]
```

Note that `s.Captured` is defined to be the type of the environment of spore `s`: a tuple with all types of captured variables.

B.2 Formalization

$t ::= x$	variable
$\mid (x : T) \Rightarrow t$	abstraction
$\mid t\ t$	application
$\mid \texttt{let}\ x = t\ \texttt{in}\ t$	let binding
$\mid \{\overline{l = t}\}$	record construction
$\mid t.l$	selection
$\mid \texttt{spore}\ \{\ \overline{x : T = t}\ ; \overline{pn}; (x : T) \Rightarrow t\ \}$	spore
$\mid \texttt{import}\ pn\ \texttt{in}\ t$	property import
$\mid t\ \texttt{compose}\ t$	spore composition
$v ::= (x : T) \Rightarrow t$	abstraction
$\mid \{\overline{l = v}\}$	record value
$\mid \texttt{spore}\ \{\ \overline{x : T = v}\ ; \overline{pn}; (x : T) \Rightarrow t\ \}$	spore value
$T ::= T \Rightarrow T$	function type
$\mid \{\overline{l : T}\}$	record type
$\mid \mathscr{S}$	
$\mathscr{S} ::= T \Rightarrow T\ \{\ \texttt{type}\ \mathscr{C} = \overline{T}\ ;\ \overline{pn}\ \}$	spore type
$\mid T \Rightarrow T\ \{\ \texttt{type}\ \mathscr{C}\ ;\ \overline{pn}\ \}$	abstract spore type
$P \in pn \rightarrow \mathscr{T}$	property map
$\mathscr{T} \in \mathscr{P}(T)$	type family
$\Gamma ::= \overline{x : T}$	type environment
$\Delta ::= \overline{pn}$	property environment

Figure B.1 – Core language syntax

We formalize spores in the context of a standard, typed lambda calculus with records. Apart from novel language and type-systematic features, our formal development follows a well-known methodology [Pierce, 2002]. Figure B.1 shows the syntax of our core language. Terms are standard except for the `spore`, `import`, and `compose` terms. A `spore` term creates a new spore. It contains a list of variable definitions (the spore header), a list of property names, and the spore's closure. A property name refers to a type family (a set of types) that all captured types must belong to.

An illustrative example of a property name and its associated type family, but in the context of Scala, is a type class: a spore satisfies such a property if there is a type class instance for all its captured types.

An `import` term imports a property name into the property environment within a lexical scope (a term); the property environment contains properties that are registered as requirements whenever a spore is created. This is explained in more detail in Section B.2.2. A `compose` term is used to compose two spores. The core language provides spore composition as a built-in feature, because type checking spore composition is markedly different from type checking

regular function composition (see Section B.2.2).

The grammar of values is standard except for spore values; in a spore value each term on the right-hand side of a definition in the spore header is a value.

The grammar of types is standard except for spore types. Spore types are refinements of function types. They additionally contain a (possibly-empty) sequence of captured types, which can be left abstract, and a sequence of property names.

B.2.1 Subtyping

Figure B.2 shows the subtyping rules. Record (S-REC) and function (S-FUN) subtyping are standard.

The subtyping rule for spores (S-SPORE) is analogous to the subtyping rule for functions with respect to the argument and result types. Additionally, for two spore types to be in a subtyping relationship either their captured types have to be the same ($M_1 = M_2$) or the supertype must be an abstract spore type ($M_2 = \texttt{type}\ \mathscr{C}$). The subtype must guarantee at least the properties of its supertype, or a superset thereof. Taken together, this rule expresses the fact that a spore type whose type member $\mathscr{C}$ is not abstract is compatible with an abstract spore type as long as it has a superset of the supertype's properties. This is important for spores used as first-class values: functions operating on spores with arbitrary environments can simply demand an abstract spore type. The way both the captured types and the properties are modeled corresponds to (but simplifies) the subtyping rule for refinement types in Scala (see Section 5.2.4).

Rule S-SPOREFUN expresses the fact that spore types are refinements of their corresponding function types, giving rise to a subtyping relationship.

$$\text{S-REC}\ \frac{\overline{l'} \subseteq \overline{l} \qquad l_i = l'_i \rightarrow T_i <: T'_i \wedge T'_i <: T_i}{\{\overline{l : T}\} <: \{\overline{l' : T'}\}} \qquad \text{S-FUN}\ \frac{T_2 <: T_1 \qquad R_1 <: R_2}{T_1 \Rightarrow R_1 <: T_2 \Rightarrow R_2}$$

$$\text{S-SPORE}\ \frac{T_2 <: T_1 \qquad R_1 <: R_2 \qquad \overline{pn'} \subseteq \overline{pn} \qquad M_1 = M_2 \vee M_2 = \texttt{type}\ \mathscr{C}}{T_1 \Rightarrow R_1\ \{\ M_1\ ;\ \overline{pn}\ \} <: T_2 \Rightarrow R_2\ \{\ M_2\ ;\ \overline{pn'}\ \}}$$

$$\text{S-SPOREFUN}\ \frac{}{T_1 \Rightarrow R_1\ \{\ M\ ;\ \overline{pn}\ \} <: T_1 \Rightarrow R_1}$$

Figure B.2 – Subtyping

$$\text{T-VAR}\ \frac{x : T \in \Gamma}{\Gamma;\Delta \vdash x : T} \qquad \text{T-SUB}\ \frac{\Gamma;\Delta \vdash t : T' \quad T' <: T}{\Gamma;\Delta \vdash t : T} \qquad \text{T-ABS}\ \frac{\Gamma, x : T_1;\Delta \vdash t : T_2}{\Gamma;\Delta \vdash (x : T_1) \Rightarrow t : T_1 \Rightarrow T_2}$$

$$\text{T-APP}\ \frac{\Gamma;\Delta \vdash t_1 : T_1 \Rightarrow T_2 \quad \Gamma;\Delta \vdash t_2 : T_1}{\Gamma;\Delta \vdash (t_1\ t_2) : T_2} \qquad \text{T-LET}\ \frac{\Gamma;\Delta \vdash t_1 : T_1 \quad \Gamma, x : T_1;\Delta \vdash t_2 : T_2}{\Gamma;\Delta \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : T_2}$$

$$\text{T-REC}\ \frac{\Gamma;\Delta \vdash \overline{t : T}}{\Gamma;\Delta \vdash \{\overline{l = t}\} : \{\overline{l : T}\}} \qquad \text{T-SEL}\ \frac{\Gamma;\Delta \vdash t : \{\overline{l : T}\}}{\Gamma;\Delta \vdash t.l_i : T_i} \qquad \text{T-IMP}\ \frac{\Gamma;\Delta, pn \vdash t : T}{\Gamma;\Delta \vdash \texttt{import}\ pn\ \texttt{in}\ t : T}$$

$$\text{T-SPORE}\ \frac{\forall s_i \in \overline{s}.\ \Gamma;\Delta \vdash s_i : S_i \qquad \overline{y : S}, x : T_1;\Delta \vdash t_2 : T_2 \qquad \forall pn \in \Delta, \Delta'.\ \overline{S} \subseteq P(pn)}{\Gamma;\Delta \vdash \texttt{spore}\ \{\ \overline{y : S = s}\ ;\Delta';(x : T_1) \Rightarrow t_2\ \} : \qquad T_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}\ ;\ \Delta, \Delta'\ \}}$$

$$\text{T-COMP}\ \frac{\begin{array}{c}\Gamma;\Delta \vdash t_1 : T_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}\ ;\ \Delta_1\ \} \\ \Gamma;\Delta \vdash t_2 : U_1 \Rightarrow T_1\ \{\ \texttt{type}\ \mathscr{C} = \overline{R}\ ;\ \Delta_2\ \} \qquad \Delta' = \{pn \in \Delta_1 \cup \Delta_2 \mid \overline{S} \subseteq P(pn) \wedge \overline{R} \subseteq P(pn)\}\end{array}}{\Gamma;\Delta \vdash t_1\ \texttt{compose}\ t_2 : U_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}, \overline{R}\ ;\ \Delta'\ \}}$$

Figure B.3 – Typing rules

B.2.2 Typing rules

Typing derivations use a judgement of the form $\Gamma;\Delta \vdash t : T$. Besides the standard variable environment Γ we use a property environment Δ which is a sequence of property names that are "active" while deriving the type T of term t. The property environment is reminiscent of the implicit parameter context used in the original work on implicit parameters [Lewis et al., 2000]; it is an environment for names whose definition sites "just happen to be far removed from their usages."

In the typing rules we assume the existence of a global property mapping P from property names pn to type families $\mathscr{T}$. This technique is reminiscent of the way some object-oriented core languages provide a global class table for type-checking. The main difference is that our core language does not include constructs to extend the global property map; such constructs are left out of the core language for simplicity, since the creation of properties is not essential to our model.

The typing rules are standard except for rules T-IMP, T-SPORE, and T-COMP, which are new. Only these three type rules inspect or modify the property environment Δ. Note that there is no rule for spore application, since there is a subtyping relationship between spores and functions (see Section B.2). Using the subsumption rule T-SUB spore application is expressed using the standard rule for function application (T-APP).

Rule T-IMP imports a property pn into the property environment within the scope defined by term t.

Rule T-SPORE derives a type for a spore term. In the spore, all terms on right-hand sides of variable definitions in the spore header must be well-typed in the same environment $\Gamma;\Delta$ according to their declared type. The body of the spore's closure, t_2, must be well-typed in an environment containing only the variables in the spore header and the closure's parameter, one of the central properties of spores. The last premise requires all captured types to satisfy both the properties in the current property environment, Δ, as well as the properties listes in the spore term, Δ'. Finally, the resulting spore type contains the argument and result types of the spore's closure, the sequence of captured types according to the spore header, and the concatenation of properties Δ and Δ'. The intuition here is that properties in the environment have been explicitly imported by the user, thus indicating that all spores in the scope of the corresponding import should satisfy them.

Rule T-COMP derives a result type for the composition of two spores. It inspects the captured types of both spores ($\overline{S}$ and $\overline{R}$) to ensure that the properties of the resulting spore, Δ, are satisfied by the captured variables of both spores. Otherwise, the argument and result types are analogous to regular function composition. Note that it's always possible to weaken the properties of a spore through spore subtyping and subsumption (T-SUB).

B.2.3 Operational semantics

Figure B.4 shows the evaluation rules of a small-step operational semantics for our core language. The only non-standard rules are E-APPSPORE, E-SPORE, E-IMP, and E-COMP3. Rule E-APPSPORE applies a spore literal to an argument value. The differences to regular function application (E-APPABS) are (a) that the types in the spore header must satisfy the properties of the spore dynamically, and (b) that the variables in the spore header must be replaced by their values in the body of the spore's closure. Rule E-SPORE is a simple congruence rule. Rule E-IMP is a computation rule that is always enabled. It adds property name pn to all spore terms within the body t. The $insert$ helper function is defined in Figure B.5 (we omit rules for `compose` and `let`, since they are analogous to rules H-INSAPP and H-INSSEL).

Rule E-COMP3 is the computation rule for spore composition. Besides computing the composition in a way analogous to regular function composition, it defines the spore header of the result spore, as well as its properties. The properties of the result spore are restricted to those that are satisfied by the captured variables of both argument spores.

B.2.4 Soundness

This section presents a soundness proof of the spore type system. The proof is based on a pair of progress and preservation theorems [Wright and Felleisen, 1994]. In addition to standard lemmas, such as Lemma B.2.3 and Lemma B.2.4, we also prove a lemma specific

E-Let1
$$\frac{t_1 \to t_1'}{\texttt{let } x = t_1 \texttt{ in } t_2 \to \texttt{let } x = t_1' \texttt{ in } t_2}$$

E-Let2
$$\texttt{let } x = v_1 \texttt{ in } t_2 \to [x \mapsto v_1]\, t_2$$

E-Rec
$$\frac{t_k \to t_k'}{\{\overline{l = v}, l_k = t_k, \overline{l' = t'}\} \to \{\overline{l = v}, l_k = t_k', \overline{l' = t'}\}}$$

E-Sel1
$$\frac{t \to t'}{t.l \to t'.l}$$

E-Sel2
$$\{\overline{l = v}\}.l_i \to v_i$$

E-App1
$$\frac{t_1 \to t_1'}{t_1\, t_2 \to t_1'\, t_2}$$

E-App2
$$\frac{t_2 \to t_2'}{v_1\, t_2 \to v_1\, t_2'}$$

E-AppAbs
$$((x : T) \Rightarrow t)\, v \to [x \mapsto v]\, t$$

E-AppSpore
$$\frac{\forall pn \in \overline{pn}.\ \overline{T} \subseteq P(pn)}{\texttt{spore } \{\ \overline{x : T = v}; \overline{pn}; (x' : T) \Rightarrow t\ \}\, v' \to \overline{[x \mapsto v]}[x' \mapsto v']\, t}$$

E-Spore
$$\frac{t_k \to t_k'}{\begin{array}{c}\texttt{spore } \{\ \overline{x : T = v}, x_k : T_k = t_k, \overline{x' : T' = t'}\ ; (x : T) \Rightarrow t\ \} \to \\ \texttt{spore } \{\ \overline{x : T = v}, x_k : T_k = t_k', \overline{x' : T' = t'}\ ; (x : T) \Rightarrow t\ \}\end{array}}$$

E-Imp
$$\texttt{import } pn \texttt{ in } t \to insert(pn, t)$$

E-Comp1
$$\frac{t_1 \to t_1'}{t_1 \texttt{ compose } t_2 \to t_1' \texttt{ compose } t_2}$$

E-Comp2
$$\frac{t_2 \to t_2'}{v_1 \texttt{ compose } t_2 \to v_1 \texttt{ compose } t_2'}$$

E-Comp3
$$\frac{\Delta = \{p \mid p \in \overline{pn}, \overline{qn}.\ \overline{T} \subseteq P(p) \wedge \overline{S} \subseteq P(p)\}}{\begin{array}{c}\texttt{spore } \{\ \overline{x : T = v}; \overline{pn}; (x' : T') \Rightarrow t\ \} \texttt{ compose spore } \{\ \overline{y : S = w}; \overline{qn}; (y' : S') \Rightarrow t'\ \} \to \\ \texttt{spore } \{\ \overline{x : T = v}, \overline{y : S = w}; \Delta; (y' : S') \Rightarrow \texttt{let } z' = t' \texttt{ in } [x' \mapsto z']\, t\}\end{array}}$$

Figure B.4 – Operational Semantics[3]

H-InsSpore1
$$\frac{\forall t_i \in \overline{t}.\ insert(pn, t_i) = t_i' \qquad insert(pn, t) = t'}{insert(pn, \texttt{spore } \{\ \overline{x : T = t}; \overline{pn}; (x' : T) \Rightarrow t\ \}) = \quad \texttt{spore } \{\ \overline{x : T = t'}; \overline{pn}, pn; (x' : T) \Rightarrow t'\ \}}$$

H-InsSpore2
$$insert(pn, \texttt{spore } \{\ \overline{x : T = v}; \overline{pn}; (x' : T) \Rightarrow t\ \}) = \quad \texttt{spore } \{\ \overline{x : T = v}; \overline{pn}, pn; (x' : T) \Rightarrow t\ \}$$

H-InsApp
$$insert(pn, t_1\ t_2) = insert(pn, t_1)\ insert(pn, t_2)$$

H-InsSel
$$insert(pn, t.l) = insert(pn, t).l$$

Figure B.5 – Helper function *insert*

to our type system, namely Lemma B.2.2, which ensures types are preserved under property import. Soundness of the type system follows from Theorem B.2.1 and Theorem B.2.2.

Lemma B.2.1. (Canonical forms)

1. *If v is a value of type $\{\overline{l : T}\}$, then v is $\{\overline{l = v}\}$ where $\overline{v}$ is a sequence of values.*
2. *If v is a value of type $T \Rightarrow R$, then v is either $(x : T_1) \Rightarrow t$ or* $\texttt{spore}\ \{\ \overline{y : S = v}\ ; \overline{pn}; (x : T_1) \Rightarrow t\ \}$ *where $T <: T_1$ and $\overline{v}$ is a sequence of values.*
3. *If v is a value of type $T \Rightarrow R\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}\ ;\ \overline{pn}\ \}$, then v is* $\texttt{spore}\ \{\ \overline{y : S = v}\ ; \overline{pn}; (x : T_1) \Rightarrow t\ \}$ *where $T <: T_1$ and $\overline{v}$ is a sequence of values.*

Proof. According to the grammar in Figure B.1, values in the core language can have three forms: $(x : T) \Rightarrow t$, $\{\overline{l = v}\}$, and $\texttt{spore}\ \{\ \overline{x : T = v}\ ; \overline{pn}; (x : T) \Rightarrow t\ \}$ where $\overline{v}$ is a sequence of values.

For the first part, according to (T-Rec) and the subtyping rules, v is $\{\overline{l = v}\}$ where $\overline{v}$ is a sequence of values of types $\overline{T}$.

For the second part, according to the subtyping rules v can have either type $T_1 \Rightarrow R_1$, $T_1 \Rightarrow R_1\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}\ ;\ \overline{pn}\ \}$, or $T_1 \Rightarrow R_1\ \{\ \texttt{type}\ \mathscr{C}\ ;\ \overline{pn}\ \}$ where $T <: T_1$ and $R_1 <: R$. If v has type $T_1 \Rightarrow R_1$, then according to the grammar and (T-Abs) v must be $(x : T) \Rightarrow t$. If v has either type $T_1 \Rightarrow R_1\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}\ ;\ \overline{pn}\ \}$ or type $T_1 \Rightarrow R_1\ \{\ \texttt{type}\ \mathscr{C}\ ;\ \overline{pn}\ \}$, then according to the grammar and (T-Spore) v must be $\texttt{spore}\ \{\ \overline{x : T = v}\ ; \overline{pn}; (x : T_1) \Rightarrow t\ \}$ where $\overline{v}$ is a sequence of values.

Part three is similar. □

Theorem B.2.1. (Progress) *Suppose t is a closed, well-typed term (that is, $\vdash t : T$ for some T). Then either t is a value or else there is some t' with $t \to t'$.*

Proof. By induction on a derivation of $t : T$. The only three interesting cases are the ones for spore creation, application (where we might apply a spore to some argument), and spore composition.

Case T-Spore: $t = \texttt{spore}\ \{\ \overline{x : S = t}\ ; \Delta'; (x : T_1) \Rightarrow t_2\ \}$, $\forall\, t_i \in \overline{t}.\ \vdash t_i : S_i$, and $\overline{x : S}, x : T_1 \vdash t_2 : T_2$. By the induction hypothesis, either all $\overline{t}$ are values, in which case t is a value; or there is a term t_i such that $t_i \to t_i'$ (since $\vdash t_i : S_i$). Thus, by (E-Spore), $t \to t'$ for some term t'.

Case T-App: $t = t_1\ t_2$ and $\vdash t_1 : T_1 \Rightarrow T_2$ and $\vdash t_2 : T_1$. By the induction hypothesis, either t_1 is a value v_1, or $t_1 \to t_1'$. In the latter case it follows from (E-App1) that $t \to t'$ for some t'. In the former case, by the induction hypothesis t_2 is either a value v_2 or $t_2 \to t_2'$. In the former case by the canonical forms lemma we have that v_2 is either $(x : T_1) \Rightarrow t$ or $\texttt{spore}\ \{\ \overline{x : T = v}\ ; \overline{pn}; (x : T_1) \Rightarrow t\ \}$ where $T <: T_1$ and $\overline{v}$ is a sequence of values; thus, either (E-AppAbs) or (E-AppSpore) apply. In the latter case, the result follows from (E-App2).

Case T-COMP: $t = t_1$ `compose` t_2 and $\vdash t_1 : T_1 \Rightarrow T_2$ { `type` $\mathscr{C} = \overline{S}$; Δ_1 } and $\vdash t_2 : U_1 \Rightarrow T_1$ { `type` $\mathscr{C} = \overline{R}$; Δ_2 }. If either t_1 or t_2 is not a value, the result follows from the induction hypothesis and (E-COMP1) or (E-COMP2). If t_1 is a value v_1 and t_2 is a value v_2, then by the canonical forms lemma, $v_1 =$ `spore` { $\overline{y : S = v}$; Δ_1 ; $(x : T_1) \Rightarrow s_1$ } and $v_2 =$ `spore` { $\overline{z : R = w}$; Δ_2 ; $(u : U_1) \Rightarrow s_2$ }. Thus, by (E-COMP3), $t \to t'$ for some t'.

□

Lemma B.2.2. (Preservation of types under import) *If* $\Gamma; \Delta, pn \vdash t : T$ *then* $\Gamma; \Delta \vdash insert(pn, t) : T$

Proof. By induction on a derivation of $t : T$. The only three interesting cases are the ones for spore creation, application (where we might apply a spore to some argument), and spore composition.

Case T-SPORE: $t =$ `spore` { $\overline{x : S = t}$; Δ' ; $(x : T_1) \Rightarrow t_2$ }, $\forall t_i \in \overline{t}$. $\vdash t_i : S_i$, and $\overline{x : S}, x : T_1 \vdash t_2 : T_2$. By the induction hypothesis, either all $\overline{t}$ are values, in which case t is a value; or there is a term t_i such that $t_i \to t_i'$ (since $\vdash t_i : S_i$). Thus, by (E-SPORE), $t \to t'$ for some term t'.

Case T-APP: $t = t_1\ t_2$ and $\vdash t_1 : T_1 \Rightarrow T_2$ and $\vdash t_2 : T_1$. By the induction hypothesis, either t_1 is a value v_1, or $t_1 \to t_1'$. In the latter case it follows from (E-APP1) that $t \to t'$ for some t'. In the former case, by the induction hypothesis t_2 is either a value v_2 or $t_2 \to t_2'$. In the former case by the canonical forms lemma we have that v_2 is either $(x : T_1) \Rightarrow t$ or `spore` { $\overline{x : T = v}$; $\overline{pn}$; $(x : T_1) \Rightarrow t$ } where $T <: T_1$ and $\overline{v}$ is a sequence of values; thus, either (E-APPABS) or (E-APPSPORE) apply. In the latter case, the result follows from (E-APP2).

Case T-COMP: $t = t_1$ `compose` t_2 and $\vdash t_1 : T_1 \Rightarrow T_2$ { `type` $\mathscr{C} = \overline{S}$; Δ_1 } and $\vdash t_2 : U_1 \Rightarrow T_1$ { `type` $\mathscr{C} = \overline{R}$; Δ_2 }. If either t_1 or t_2 is not a value, the result follows from the induction hypothesis and (E-COMP1) or (E-COMP2). If t_1 is a value v_1 and t_2 is a value v_2, then by the canonical forms lemma, $v_1 =$ `spore` { $\overline{y : S = v}$; Δ_1 ; $(x : T_1) \Rightarrow s_1$ } and $v_2 =$ `spore` { $\overline{z : R = w}$; Δ_2 ; $(u : U_1) \Rightarrow s_2$ }. Thus, by (E-COMP3), $t \to t'$ for some t'.

□

Lemma B.2.3. (Preservation of types under substitution) *If* $\Gamma, x : S; \Delta \vdash t : T$ *and* $\Gamma; \Delta \vdash s : S$, *then* $\Gamma; \Delta \vdash [x \mapsto s]t : T$

Proof. By induction on a derivation of $\Gamma, x : S; \Delta \vdash t : T$. □

Lemma B.2.4. (Weakening) *If* $\Gamma; \Delta \vdash t : T$ *and* $x \notin dom(\Gamma)$, *then* $\Gamma, x : S; \Delta \vdash t : T$.

Proof. By induction on a derivation of $\Gamma; \Delta \vdash t : T$. □

Theorem B.2.2. (Preservation) *If* $\Gamma; \Delta \vdash t : T$ *and* $t \to t'$, *then* $\Gamma; \Delta \vdash t' : T$.

Proof. By induction on a derivation of $t : T$.

- Case T-Sel: $t = s.l_i$ and $\Gamma;\Delta \vdash s : \{\overline{l : S}\}$. Since $t \to t'$ we have either by (E-Sel1) $s \to s'$ and $t' = s'.l_i$, or we have by (E-Sel2) $s = \{\overline{l = v}\}$ and $t' = v_i$. In the former case, by the induction hypothesis, $\Gamma;\Delta \vdash s' : \{\overline{l : S}\}$ and thus by (T-Sel), $\Gamma;\Delta \vdash s'.l_i : S_i$. In the latter case, by (T-Rec), $\Gamma;\Delta \vdash v_i : S_i$.

- Case T-Imp: $t = \texttt{import}\ pn\ \texttt{in}\ s$ and $\Gamma;\Delta, pn \vdash s : T$. Since $t \to t'$, we have by (E-Imp) $t' = insert(pn, s)$. By Lemma B.2.2, $\Gamma;\Delta \vdash insert(pn, s) : T$.

- Case T-App: $t = s_1\ s_2$ and $T = S_2$. By (T-App), $\Gamma;\Delta \vdash s_1 : S_1 \Rightarrow S_2$ and $\Gamma;\Delta \vdash s_2 : S_1$. Since $t \to t'$, either (E-App1), (E-App2), (E-AppAbs), or (E-AppSpore) applies. If (E-App1) applies, then $s_1 \to s_1'$ and $t' = s_1'\ s_2$. By the induction hypothesis, $\Gamma;\Delta \vdash s_1' : S_1 \Rightarrow S_2$. By (T-App), $\Gamma;\Delta \vdash t' : S_2$. The case where (E-App2) applies is similar. If (E-AppAbs) applies, then $s_1 = (x : S_1) \Rightarrow t_2$ and $s_2 = v$ and $t' = [x \mapsto v]t_2$. By (T-Abs), $\Gamma, x : S_1;\Delta \vdash t_2 : S_2$. By (T-App), $\Gamma;\Delta \vdash v : S_1$. By Lemma B.2.3, $\Gamma;\Delta \vdash [x \mapsto v]t_2 : S_2$.

  If (E-AppSpore) applies, then $s_1 = \texttt{spore}\ \{\ \overline{x : T = v}\ ; \overline{pn}; (y : S_1) \Rightarrow t_2\ \}$ and $s_2 = v'$ and $\forall pn \in \overline{pn}.\ \overline{S} \subseteq P(pn)$ and $t' = \overline{[x \mapsto v]}[y \mapsto v']t_2$. By (T-Spore), $\overline{x : T}, y : S_1;\Delta \vdash t_2 : S_2$. By (T-App), $\Gamma;\Delta \vdash v' : S_1$. By Lemma B.2.4, $\Gamma, \overline{x : T}, y : S_1;\Delta \vdash t_2 : S_2$. By Lemma B.2.4, $\Gamma, \overline{x : T};\Delta \vdash v' : S_1$. By Lemma B.2.3, $\Gamma, \overline{x : T};\Delta \vdash [y \mapsto v']t_2 : S_2$. By (T-Spore), we also have $\forall v_i \in \overline{v}.\ \Gamma;\Delta \vdash v_i : T_i$. By Lemma B.2.3, $\Gamma;\Delta \vdash \overline{[x \mapsto v]}[y \mapsto v']t_2 : S_2$.

- Case T-Spore: $t = \texttt{spore}\ \{\ \overline{y : S = s}\ ; \Delta'; (x : T_1) \Rightarrow t_2\ \}$ and $T = T_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}\ ;\ \Delta, \Delta'\ \}$. By (T-Spore), $\forall s_i \in \overline{s}.\ \Gamma;\Delta \vdash s_i : S_i$ and $\overline{y : S}, x : T_1;\Delta \vdash t_2 : T_2$ and $\forall pn \in \Delta, \Delta'.\ \overline{S} \subseteq P(pn)$. Since $t \to t'$, rule (E-Spore) must apply, and thus $s_i \to s_i'$ for some s_i. By the induction hypothesis, $\Gamma;\Delta \vdash s_i' : S_i$. Thus, by (T-Spore), $\Gamma;\Delta \vdash t' : T$.

- Case T-Comp: $t = s_1\ \texttt{compose}\ s_2$ and $T = T_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}, \overline{R}\ ;\ \Delta_3\ \}$. By (T-Comp), $\Gamma \vdash s_1 : U_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}\ ;\ \Delta_1\ \}$ and $\Gamma \vdash s_2 : T_1 \Rightarrow U_1\ \{\ \texttt{type}\ \mathscr{C} = \overline{R}\ ;\ \Delta_2\ \}$ and $\Delta_3 = \{pn \in \Delta_1 \cup \Delta_2 \mid \overline{S} \subseteq P(pn) \wedge \overline{R} \subseteq P(pn)\}$. Since $t \to t'$, either (E-Comp1), (E-Comp2), or (E-Comp3) applies.

  If (E-Comp1) applies, then $s_1 \to s_1'$, and by (T-Comp), $\Gamma;\Delta \vdash s_1 : U_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}\ ;\ \Delta_1\ \}$, and $t' = s_1'\ \texttt{compose}\ s_2$. By the induction hypothesis, $\Gamma;\Delta \vdash s_1' : U_1 \Rightarrow T_2\{\ \texttt{type}\ \mathscr{C} = \overline{S}\ ;\ \Delta_1\ \}$. By (T-Comp), we know that $\Gamma;\Delta \vdash s_2 : T_1 \Rightarrow U_1\ \{\ \texttt{type}\ \mathscr{C} = \overline{R}\ ;\ \Delta_2\ \}$ and $\Delta_3 = \{pn \in \Delta_1 \cup \Delta_2 \mid \overline{S} \subseteq P(pn) \wedge \overline{R} \subseteq P(pn)\}$. By (T-Comp), $\Gamma;\Delta \vdash t' : T$.

  If (E-Comp2) applies, then $s_2 \to s_2'$, and by (T-Comp), $\Gamma;\Delta \vdash s_2 : T_1 \Rightarrow U_1\ \{\ \texttt{type}\ \mathscr{C} = \overline{R}\ ;\ \Delta_2\ \}$, and $t' = v_1\ \texttt{compose}\ s_2'$. By the induction hypothesis, $\Gamma;\Delta \vdash s_2' : T_1 \Rightarrow U_1\ \{\ \texttt{type}\ \mathscr{C} = \overline{R}\ ;\ \Delta_2\ \}$. Since (E-Comp2) applies, $s_1 = v_1$, so by (T-Comp), we know that $\Gamma;\Delta \vdash v_1 : U_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}\ ;\ \Delta_1\ \}$ and $\Delta_3 = \{pn \in \Delta_1 \cup \Delta_2 \mid \overline{S} \subseteq P(pn) \wedge \overline{R} \subseteq P(pn)\}$. By (T-Comp), $\Gamma;\Delta \vdash t' : T$.

  If (E-Comp3) applies, then $s_1 = \texttt{spore}\ \{\overline{x : S = v}; \Delta_1; (y : U_1) \Rightarrow t_2\}$ and $s_2 = \texttt{spore}\ \{\overline{y : R = w}; \Delta_2; (z : T_1) \Rightarrow u_1\ \}$ and $\Delta_3 = \{p \mid p \in \Delta_1, \Delta_2.\ \overline{S} \subseteq P(p) \wedge \overline{R} \subseteq P(p)\}$. By (E-Comp3), $t' = \texttt{spore}\ \{\overline{x : S = v}, \overline{y : R = w}; \Delta_3; (z : T_1) \Rightarrow \texttt{let}\ x = u_1\ \texttt{in}\ [y \mapsto x]t_2\}$.

First, we show that $\forall v_i \in \overline{v}.\, \Gamma;\Delta \vdash v_i : S_i$ and $\forall w_i \in \overline{w}.\, \Gamma;\Delta \vdash w_i : R_i$. This follows from the fact that s_1 and s_2 are well-typed spores and (T-SPORE).

Second, we show that $\overline{x:S}, \overline{y:R}, z:T_1;\Delta \vdash \texttt{let}\ x = u_1\ \texttt{in}\ [y \mapsto x]\,t_2 : T_2$. By (T-LET), we need to show that $\overline{x:S}, \overline{y:R}, z:T_1;\Delta \vdash u_1 : U_1$ and $\overline{x:S}, \overline{y:R}, z:T_1, x:U_1;\Delta \vdash [y \mapsto x]\,t_2 : T_2$. The former follows from (T-SPORE) and Lemma B.2.4. To prove the latter: given that s_1 is well-typed, by (T-SPORE) we have that $\overline{x:S}, y:U_1 \vdash t_2 : T_2$. By Lemma B.2.4, $\overline{x:S}, y:U_1, x:U_1 \vdash t_2 : T_2$. By Lemma B.2.3, $\overline{x:S}, x:U_1 \vdash [y \mapsto x]\,t_2 : T_2$. By Lemma B.2.4, $\overline{x:S}, \overline{y:R}, z:T_1, x:U_1;\Delta \vdash [y \mapsto x]\,t_2 : T_2$.

Third, we show that $\forall pn \in \Delta, \Delta_3.\ \overline{S} \subseteq P(pn) \wedge \overline{R} \subseteq P(pn)$. Since s_1 is well-typed, we have $\forall pn \in \Delta, \Delta_1.\ \overline{S} \subseteq P(pn)$. Since s_2 is well-typed, we have $\forall pn \in \Delta, \Delta_2.\ \overline{R} \subseteq P(pn)$. Moreover, we have that $\Delta_3 = \{p \mid p \in \Delta_1, \Delta_2.\ \overline{S} \subseteq P(p) \wedge \overline{R} \subseteq P(p)\}$. Thus, $\forall pn \in \Delta, \Delta_3.\ \overline{S} \subseteq P(pn) \wedge \overline{R} \subseteq P(pn)$.

By (T-SPORE) it follows from the previous three subgoals that $\Gamma;\Delta \vdash t' : T$.

□

B.2.5 Relation to spores in Scala

The soundness proof (see Section B.2.4) of the formal type system guarantees several important properties for well-typed programs which closely correspond to the pragmatic model of spores in Scala:

1. Application of spores: for each property name pn, it is ensured that the dynamic types of all captured variables are contained in the type family pn maps to ($P(pn)$).
2. Dynamically, a spore only accesses its parameter and the variables in its header.
3. The properties computed for a composition of two spores is a safe approximation of the properties that are dynamically required.

B.2.6 Excluded types

This section shows how the formal model can be extended with excluded types as described above (see Section 5.2.4). Figure B.6 shows the syntax extensions: first, spore terms and values are augmented with a sequence of excluded types; second, spore types and abstract spore types get another member $\texttt{type}\ \mathscr{E} = \overline{T}$ specifying the excluded types.

Figure B.7 shows how the subtyping rules for spores have to be extended. Rule S-ESPORE requires that for each excluded type T' in the supertype, there must be an excluded type T in the subtype such that $T' <: T$. This means that by excluding type T, subtypes like T' are also prevented from being captured.

$$
\begin{array}{lll}
t ::= \ldots & & \text{terms} \\
\quad \mid \texttt{spore}\ \{\ \overline{x : T = t}\ ;\ \overline{T}; \overline{pn}; (x : T) \Rightarrow t\ \} & & \text{spore} \\
\\
v ::= \ldots & & \text{values} \\
\quad \mid \texttt{spore}\ \{\ \overline{x : T = v}\ ;\ \overline{T}; \overline{pn}; (x : T) \Rightarrow t\ \} & & \text{spore value} \\
\\
\mathscr{S} ::= T \Rightarrow T\ \{\ \texttt{type}\ \mathscr{C} = \overline{T}\ ;\ \texttt{type}\ \mathscr{E} = \overline{T}\ ;\ \overline{pn}\ \} & & \text{spore type} \\
\quad \mid T \Rightarrow T\ \{\ \texttt{type}\ \mathscr{C}\ ;\ \texttt{type}\ \mathscr{E} = \overline{T}\ ;\ \overline{pn}\ \} & & \text{abstract spore type}
\end{array}
$$

Figure B.6 – Core language syntax extensions

S-ESPORE

$$
\frac{T_2 <: T_1 \qquad R_1 <: R_2 \qquad \overline{pn'} \subseteq \overline{pn} \qquad M_1 = M_2 \vee M_2 = \texttt{type}\ \mathscr{C} \qquad \forall T' \in \overline{U'}.\ \exists T \in \overline{U}.\ T' <: T}{T_1 \Rightarrow R_1\ \{\ M_1\ ;\ \texttt{type}\ \mathscr{E} = \overline{U}\ ;\ \overline{pn}\ \} \qquad <: T_2 \Rightarrow R_2\ \{\ M_2\ ;\ \texttt{type}\ \mathscr{E} = \overline{U'}\ ;\ \overline{pn'}\ \}}
$$

S-ESPOREFUN

$$
T_1 \Rightarrow R_1\ \{\ M\ ;\ E\ ;\ \overline{pn}\ \} <: T_1 \Rightarrow R_1
$$

Figure B.7 – Subtyping extensions

Figure B.8 shows the extensions to the typing rules. Rule T-ESPORE additionally requires that none of the captured types $\overline{S}$ is a subtype of one of the types contained in the excluded types $\overline{U}$. The excluded types are recorded in the type of the spore. Rule T-ECOMP computes a new set of excluded types $\overline{V}$ based on both the excluded types and the captured types of t_1 and t_2. Given that it is possible that one of the spores captures a type that is excluded in the other spore, the type of the result spore excludes only those types that are guaranteed not be captured.

T-ESPORE

$$
\frac{\begin{array}{c}\forall s_i \in \overline{s}.\ \Gamma; \Delta \vdash s_i : S_i \qquad \overline{y : S}, x : T_1; \Delta \vdash t_2 : T_2 \\ \forall pn \in \Delta, \Delta'.\ \overline{S} \subseteq P(pn) \qquad \forall S_i \in \overline{S}.\ \forall U_j \in \overline{U}.\ \neg(S_i <: U_j)\end{array}}{\Gamma; \Delta \vdash \texttt{spore}\ \{\ \overline{y : S = s}\ ; \overline{U}; \Delta'; (x : T_1) \Rightarrow t_2\ \} : \qquad T_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}\ ;\ \texttt{type}\ \mathscr{E} = \overline{U}\ ;\ \Delta, \Delta'\ \}}
$$

T-ECOMP

$$
\frac{\begin{array}{c}\Gamma; \Delta \vdash t_1 : T_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}\ ;\ \texttt{type}\ \mathscr{E} = \overline{U}\ ;\ \Delta_1\ \} \\ \Gamma; \Delta \vdash t_2 : U_1 \Rightarrow T_1\ \{\ \texttt{type}\ \mathscr{C} = \overline{R}\ ;\ \texttt{type}\ \mathscr{E} = \overline{U'}\ ;\ \Delta_2\ \} \\ \Delta' = \{pn \in \Delta_1 \cup \Delta_2 \mid \overline{S} \subseteq P(pn) \wedge \overline{R} \subseteq P(pn)\} \qquad \overline{V} = (\overline{U} \setminus \overline{R}) \cup (\overline{U'} \setminus \overline{S})\end{array}}{\Gamma; \Delta \vdash t_1\ \texttt{compose}\ t_2 : U_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathscr{C} = \overline{S}, \overline{R}\ ;\ \texttt{type}\ \mathscr{E} = \overline{V}\ ;\ \Delta'\ \}}
$$

Figure B.8 – Typing extensions

Figure B.9 shows the extensions to the operational semantics. Rule E-EAPPSPORE additionally requires that none of the captured types $\overline{T}$ are contained in the excluded types $\overline{U}$. Rule E-ECOMP3 computes the set of excluded types of the result spore in

the same way as in the corresponding type rule (T-ECOMP).

$$\text{E-EAPPSPORE}\quad \frac{\forall pn \in \overline{pn}.\ \overline{T} \subseteq P(pn) \qquad \forall T_i \in \overline{T}.\ T_i \notin \overline{U}}{\texttt{spore}\ \{\ \overline{x : T = v}\ ;\ \overline{U}\ ;\ \overline{pn}\ ;\ (x' : T) \Rightarrow t\ \}\ v' \rightarrow \qquad \overline{[x \mapsto v]}[x' \mapsto v']t}$$

$$\text{E-ECOMP3}\quad \frac{\Delta = \{p \mid p \in \overline{pn}, \overline{qn}.\ \overline{T} \subseteq P(p) \wedge \overline{S} \subseteq P(p)\} \qquad \overline{V} = (\overline{U} \setminus \overline{S}) \cup (\overline{U'} \setminus \overline{T})}{\begin{array}{c}\texttt{spore}\ \{\ \overline{x : T = v}\ ;\ \overline{U}\ ;\ \overline{pn}\ ;\ (x' : T') \Rightarrow t\ \}\ \texttt{compose} \\ \texttt{spore}\ \{\ \overline{y : S = w}\ ;\ \overline{U'}\ ;\ \overline{qn}\ ;\ (y' : S') \Rightarrow t'\ \} \rightarrow \qquad \texttt{spore}\ \{\ \overline{x : T = v}, \overline{y : S = w}\ ;\ \overline{V}\ ;\ \Delta\ ; \\ (y' : S') \Rightarrow \texttt{let}\ z' = t'\ \texttt{in}\ [x' \mapsto z']t\ \}\end{array}}$$

Figure B.9 – Operational semantics extensions

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Bibliography

M. D. Adams and T. M. DuBuisson. Template your boilerplate: Using Template Haskell for efficient generic programming. In *Haskell'12*, 2012.

Gul A Agha. Actors: A model of concurrent computation in distributed systems. Technical report, DTIC Document, 1985.

A. Alimarine and M. J. Plasmeijer. A generic programming extension for clean. In *IFL '02*, 2002.

A. Alimarine and S. Smetsers. Efficient generic functional programming. Technical report NIII-R0425, Nijmegen Institute for Computing and Information Sciences, University of Nijmegen, 2004.

Apache. Avro®. http://avro.apache.org, 2013. Accessed: 2013-08-11.

Apache. Hadoop. http://hadoop.apache.org/, 2015.

Andrew W. Appel and Marcelo J. R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, Computer Science Department, 1993.

Michael Armbrust, Armando Fox, David A. Patterson, Nick Lanham, Beth Trushkowsky, Jesse Trutna, and Haruki Oh. SCADS: Scale-independent storage for social computing applications. In *CIDR*, 2009.

Joe Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, September 2010. ISSN 0001-0782.

Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. Prog. Lang. and Sys.*, 11(4):598–632, October 1989.

Azavea. GeoTrellis. http://www.azavea.com/products/geotrellis/, 2010. Accessed: 2013-08-11.

John Billings, Peter Sewell, Mark Shinwell, and Rok Strniša. Type-safe distributed programming for ocaml. In *Proceedings of the 2006 workshop on ML*, pages 20–31. ACM, 2006.

Rajendra Bose and James Frew. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37(1):1–28, 2005.

G. Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, et al. Concurrent collections. *Scientific Programming*, 18(3), 2010.

Zoran Budimlic, Michael G. Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David M. Peixotto, Vivek Sarkar, Frank Schlimbach, and Sagnak Tasirlar. Concurrent collections. *Scientific Programming*, 18(3-4):203–217, 2010. doi: 10.3233/SPR-2011-0305.

Zoran Budimlic, Vincent Cavé, Raghavan Raman, Jun Shirako, Sagnak Tasirlar, Jisheng Zhao, and Vivek Sarkar. The design and implementation of the Habanero-Java parallel programming language. In *OOPSLA Companion*, pages 185–186, 2011. doi: 10.1145/2048147.2048198.

Michael G. Burke, Kathleen Knobe, Ryan Newton, and Vivek Sarkar. Concurrent collections programming model. In *Encyclopedia of Parallel Computing*, pages 364–371. 2011. doi: 10.1007/978-0-387-09766-4_238.

E. Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Scala'13*, 2013.

E. Burmako and M. Odersky. Scala macros, a technical report. In *Third International Valentin Turchin Workshop on Metacomputation*, 2012.

Luca Cardelli, James E. Donahue, Mick J. Jordan, Bill Kalsow, and Greg Nelson. The modula-3 Type system. In *POPL*, pages 202–212, 1989.

Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. Object serialization for marshalling data in a Java interface to MPI. In *Java Grande*, pages 66–71, 1999.

B. Chadwick and K. Lieberherr. Weaving generic programming and traversal performance. In *AOSD'10*, 2010.

Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: A status report. In *Proc. DAMP Workshop*, pages 10–18. ACM, 2007.

Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. *ACM SIGPLAN Notices*, 45(6):363–375, June 2010a.

Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *PLDI*, pages 363–375, 2010b.

James Cheney, Laura Chiticariu, and Wang Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *PLDI'05*, pages 85–95, 2005.

D. Clarke and A. Löh. Generic haskell, specifically. In J. Gibbons and J. Jeuring, editors, *Generic Programming*, IFIP, pages 21–47. Kluwer Academic Publishers, 2003.

Alex Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. NOVA: A functional language for data parallelism. Technical Report NVR-2013-002, NVIDIA Corporation, July 2013.

Evan Czaplicki. Elm: Concurrent frp for functional guis. *Senior thesis, Harvard University*, 2012.

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd W. Schiller. Building and using pluggable type-checkers. In *ICSE'11*, pages 681–690, 2011.

Gilles Dubochet. *Embedded Domain-Specific Languages using Libraries and Dynamic Metaprogramming*. PhD thesis, EPFL, Switzerland, 2011.

Martin Elsman. Type-specialized serialization with sharing. In *Trends in Functional Programming*, pages 47–62, 2005.

B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In *ECOOP'07*, 2007.

Jeff Epstein, Andrew P. Black, and Simon L. Peyton Jones. Towards haskell in the cloud. In *Haskell Symposium*, pages 118–129, 2011.

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: Library-based syntactic language extensibility. In *OOSPLA'11*, 2011.

Marius Eriksen. Your server as a function. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*, PLOS '13, pages 5:1–5:7, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2460-1. doi: 10.1145/2525528.2525538. URL http://doi.acm.org/10.1145/2525528.2525538.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247. ACM, 1993.

Daniel Friedman and David Wise. The impact of applicative programming on multiprocessing. In *International Conference on Parallel Processing*, 1976.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

Guillaume Germain. Concurrency oriented programming in Termite Scheme. In *Erlang Workshop*, page 20. ACM, 2006.

J. Gibbons. Design patterns as higher-order datatype-generic programs. In *WGP '06*, 2006.

Joseph Gil and Itay Maman. Whiteoak: introducing structural typing into Java. In Gail E. Harris, editor, *OOPSLA*, pages 73–90, 2008.

Brian Goetz. JSR 335: Lambda expressions for the Java programming language, 2013.

Google. Protocol Buffers. https://code.google.com/p/protobuf/, 2008. Accessed: 2013-08-11.

Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220, 2009.

Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In Theo D'Hondt, editor, *ECOOP*, pages 354–378, 2010.

Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. Futures and promises. *http://docs.scala-lang.org/overviews/core/futures.html*, 2012.

Jr. R. H. Halstead. MultiLISP: A language for concurrent symbolic computation. *ACM Trans. Prog. Lang. and Sys.*, 7(4):501–538, October 1985.

Jiansen He, Philip Wadler, and Philip Trinder. Typecasting actors: from akka to TAkka. In *Scala*, SCALA '14, pages 23–33, 2014. ISBN 978-1-4503-2868-5.

Jr. Henry C. Baker and Carl Hewitt. The incremental garbage collection of processes. In *Proc. Symp. on Art. Int. and Prog. Lang.*, 1977.

Stephan Herhut, Richard L. Hudson, Tatiana Shpeisman, and Jaswanth Sreeram. River trail: a path to parallelism in JavaScript. In *OOPSLA*, pages 729–744, 2013.

Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *PPoPP*, pages 197–206, 1990.

Maurice Herlihy and Barbara Liskov. A value transmission method for abstract data types. *ACM Trans. Program. Lang. Syst*, 4(4):527–551, 1982.

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. April 2008. ISBN 0123705916.

Rich Hickey. The clojure programming language. In *DLS*, page 1. ACM, 2008.

R. Hinze, J. Jeuring, and A. Loeh. Comparing approaches to generic programming in Haskell. In *Datatype-Generic Programming*, volume 4719. Springer Berlin/Heidelberg, 2007.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst*, 23(3):396–450, May 2001.

William N. Scherer III, Doug Lea, and Michael L. Scott. Scalable synchronous queues. *Commun. ACM*, 52(5):100–111, 2009. doi: 10.1145/1506409.1506431.

International Standard ISO/IEC 14882:2011. *Programming Languages ‚Äì C++*. International Organization for Standards, 2011.

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

P. Jansson and J. Jeuring. Polyp - a polytypic programming language extension. In *POPL '97*, 1997.

Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel Java. In *OOPSLA*, pages 97–116, 2009. doi: 10.1145/1640089.1640097.

Andrew Kennedy. Pickler combinators. *J. Funct. Program.*, 14(6):727–739, 2004.

Roland Kuhn. Akka typed – between session types and the actor model, 7 2015. URL http://www.slideshare.net/rolandkuhn/akka-typed-between-session-types-and-the-actor-model. Curry On, Prague, Czech Republic.

Lindsey Kuper and Ryan R. Newton. Lvars: Lattice-based data structures for deterministic parallelism. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 71–84, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2381-9.

Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. Freeze after writing: Quasi-deterministic parallel programming with lvars. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 257–270, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8.

R. Lämmel and K. Ostermann. Software extension and integration with type classes. In *GPCE '06*, 2006.

R. Lämmel and S. Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *TLDI'03*, 2003.

Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.

Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark Shields. Implicit parameters: Dynamic scoping with static types. In *POPL*, pages 108–118, 2000.

K. J. Lieberherr. *Adaptive Object Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing, 1996.

Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat. An efficient implementation of Java's remote method invocation. In *PPOPP*, pages 173–182, August 1999.

J. P. Magalhães, S. Holdermans, J. Jeuring, and A. Löh. Optimizing generics is easy! In *PEPM '10*, 2010.

José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for Haskell. In Jeremy Gibbons, editor, *Haskell*, pages 37–48, 2010.

Patrick Maier and Philip W. Trinder. Implementing a high-level distributed-memory parallel Haskell in Haskell. In *IFL*, volume 7257, pages 35–50. Springer, 2011. ISBN 978-3-642-34406-0.

Simon Marlow, Ryan Newton, and Simon Peyton Jones. A monad for deterministic parallelism. In *Proc. Haskell Symposium*, pages 71–82. ACM, 2011.

Nicholas D. Matsakis. Parallel closures: A new twist on an old idea. In *HotPar*, 2012.

Niko Matsakis. Fn types in Rust, take 3. http://smallcultfollowing.com/babysteps/blog/2013/10/10/fn-types-in-rust, 2013.

Michael M. McKerns, Leif Strand, Tim Sullivan, Alta Fang, and Michael A. G. Aivazis. Building a framework for predictive science. *CoRR*, abs/1202.1056, 2012.

Erik Meijer. Confessions of a used programming language salesman. In *OOPSLA*, 2007.

Christopher Meiklejohn and Peter Van Roy. Lasp: A language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, PPDP '15, pages 184–195, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3516-4.

John M. Mellor-Crummey. Concurrent queues: Practical fetch-and-Φ algorithms. 1987.

Leo A. Meyerovich and Ariel S. Rabkin. Empirical analysis of programming language adoption. In *OOPSLA*, 2013.

Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.

Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky. Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization. In *OOPSLA*, pages 183–202, 2013.

Heather Miller, Philipp Haller, and Martin Odersky. Spores: A type-based foundation for closures in the age of concurrency and distribution. In *ECOOP*, volume 8586, pages 308–333. Springer, 2014a.

Heather Miller, Philipp Haller, Lukas Rytz, and Martin Odersky. Functional programming for all! Scaling a MOOC for students and professionals alike. In *ICSE*, pages 265–263, 2014b.

Moir and Shavit. Concurrent data structures. In Mehta and Sahni, editors, *Handbook of Data Structures and Applications, Chapman & Hall/CRC*. 2005.

A. Moors, F. Piessens, and W. Joosen. An object-oriented approach to datatype-generic programming. In *WGP '06*, 2006.

J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst*, 21(3):527–568, 1999.

Tom Murphy VII, Karl Crary, and Robert Harper. Type-safe distributed programming with ML5. In *TGC*, volume 4912, pages 108–123. Springer, 2007. doi: 10.1007/978-3-540-78663-4_9.

D. R. Musser and A. A. Stepanov. Generic programming. In *ISAAC '88*, 1989.

Nathan Marz and James Xu and Jason Jackson et al. Storm. http://storm-project.net/, 2012. Accessed: 2013-08-11.

Nathan Sweet. Kryo. https://code.google.com/p/kryo/, 2013. Accessed: 2013-08-11.

Karen Ng, Matt Warren, Peter Golde, and Anders Hejlsberg. The Roslyn project: Exposing the C# and VB compiler,äôs code analysis. http://msdn.microsoft.com/en-gb/hh500769, September 2012. Accessed: 2013-08-11.

NICTA. Scoobi. https://github.com/nicta/scoobi, 2015.

Peter Norvig and Jeff Dean. Latency numbers every programmer should know. https://gist.github.com/jboner/2841832, 2012.

M. Odersky and A. Moors. Fighting bit rot with types (experience report: Scala collections). In *IARCS'09*, 2009.

Martin Odersky. The Scala language specification, 2013.

Martin Odersky and Matthias Zenger. Scalable component abstractions. In Ralph E. Johnson and Richard P. Gabriel, editors, *OOPSLA*, pages 41–57, 2005.

Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Press, Mountain View, CA, 2010.

B. C. d. S. Oliveira and J. Gibbons. Scala for generic programmers. *Journal of Functional Programming*, 20(3,4):303–352, 2010.

B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *OOPSLA'10*, 2010.

Oracle, Inc. Java Object Serialization Specification. http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html, 2011. Accessed: 2013-08-11.

D. Orleans and K. J. Lieberherr. DJ: Dynamic adaptive programming in Java. In *REFLECTION*. Springer-Verlag, 2001.

Oscar Boykin and Mike Gagnon and Sam Ritchie. Twitter Chill. https://github.com/twitter/chill, 2012. Accessed: 2013-08-11.

Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 293–307, Oakland, CA, 2015. USENIX Association. ISBN 978-1-931971-218. URL https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout.

Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA'08*, 2008.

Simon Peyton Jones. Haskell language and library specification. https://wiki.haskell.org/Language_and_library_specification, 2014.

Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *POPL*, pages 295–308, 1996.

Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and RMI for Java. *Concurrency - Practice and Experience*, 12(7):495–518, 2000.

Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

Esmond Pitt and Kathy McNiff. *Java.Rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0201700433.

Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. A generic parallel collection framework. In *Euro-Par*, volume 6853, pages 136–147. Springer, 2011.

Aleksandar Prokopec, Heather Miller, Tobias Schlatter, Philipp Haller, and Martin Odersky. Flowpools: A lock-free deterministic concurrent dataflow abstraction. In *LCPC*, pages 158–173. Springer, 2012a.

Aleksandar Prokopec, Heather Miller, Tobias Schlatter, Philipp Haller, and Martin Odersky. Flowpools: A lock-free deterministic concurrent dataflow abstraction– proofs. Technical Report EPFL-REPORT-181098, EPFL, Lausanne, June 2012b.

Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ concepts. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 295–308, 2006.

A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell'08*, 2008.

Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6), 2012.

Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *ICFP*, pages 317–328. ACM, 2009.

Andreas Rossberg. *Typed open programming: a higher-order, typed approach to dynamic modularity and distribution*. PhD thesis, Saarland University, 2007.

Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. Alice through the looking glass. *Trends in Functional Programming*, 5:79–96, 2004.

Andreas Rossberg, Guido Tack, and Leif Kornstaedt. Status report: HOT pickles, and how to serve them. In *ML*, pages 25–36, 2007.

Peter Van Roy. Announcing the mozart programming system. *SIGPLAN Notices*, 34(4):33–34, 1999.

Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004. ISBN 0-262-22069-5.

Vijay A. Saraswat, Vivek Sarkar, and Christoph von Praun. X10: concurrent programming for modern architectures. In *PPOPP*, page 271, 2007. doi: 10.1145/1229428.1229483.

Alex Schwendner. Distributed functional programming in Scheme. Master's thesis, Massachusetts Institute of Technology, 2009.

Peter Sewell, James J Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: high-level programming language design for distributed computation. In *ACM SIGPLAN Notices*, volume 40, pages 15–26. ACM, 2005.

D. Shabalin, E. Burmako, and M. Odersky. Quasiquotes for Scala. Technical Report EPFL-REPORT-185242, EPFL, Switzerland, 2013.

E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412, September 1989.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt, January 2011a. URL https://hal.inria.fr/inria-00555588.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011b. Springer-Verlag. ISBN 978-3-642-24549-7.

T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In *Haskell '02*, 2002.

Kamil Skalski. Syntax-extending and type-reflecting macros in an object-oriented language. Master's thesis, University of Warsaw, Poland, 2005.

Rok Strnisa, Peter Sewell, and Matthew J. Parkinson. The Java module system: core design and semantic definition. In *OOPSLA*, pages 499–514, 2007.

Guido Tack, Leif Kornstaedt, and Gert Smolka. Generic pickling and minimization. *Electr. Notes Theor. Comput. Sci.*, 148(2):79–103, 2006.

Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci*, 248(1-2):211–242, 2000.

Sagnak Tasirlar and Vivek Sarkar. Data-driven tasks and their implementation. In *ICPP*, pages 652–661, 2011. doi: 10.1109/ICPP.2011.87.

Twitter. Scalding. https://github.com/twitter/scalding, 2015.

Typesafe. Akka. http://akka.io/, 2009. Accessed: 2013-08-11.

Guido van Rossum. Python programming language. In *USENIX Annual Technical Conference*. USENIX, 2007.

Dimitrios Vytiniotis and Andrew J. Kennedy. Functional pearl: every bit counts. *SIGPLAN Not.*, 45(9):15–26, September 2010.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89*, 1989.

Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A note on distributed computing. In *MOS*, pages 49–64, 1996.

Stefan Wehr and Peter Thiemann. JavaGI: The interaction of type classes with interfaces and inheritance. *ACM Trans. Program. Lang. Syst*, 33(4):12, 2011.

Matt Welsh and David E. Culler. Jaguar: enabling efficient communication and I/O in java. *Concurrency - Practice and Experience*, 12(7):519–538, 2000.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput*, 115(1):38–94, November 1994.

Matei Zaharia. Next-generation languages meet next-generation big data: Leveraging scala in spark, August 2014. URL http://functional.tv/post/97699944449/scala-by-the-bay2014-matei-zaharia-next-generation-langu. Talk given at Scala By the Bay 2014, San Francisco, CA, USA.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. USENIX, 2012.

Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. Ownership and immutability in generic java. In *OOPSLA*, pages 598–617. ACM, 2010.

Faculty of Computer, Communication,
and Information Science
EPFL
Station 14
1015 Lausanne
Switzerland

Phone: +41 78 625 20 23
Fax: +41 21 693 66 60
heather.miller@epfl.ch
http://heather.miller.am

HEATHER MILLER

Citizenship

USA

Education

EPFL, *Lausanne, Switzerland* *2009 – 2015*
Ph.D. in Computer Science
Advisor: Martin Odersky *2011 – 2015*

University of Miami, *Coral Gables, FL* *2006 – 2009*
BSEE in Electrical Engineering, Audio Engineering, *with honors, May 2009*

Cooper Union for the Advancement of Science and Art, *New York, NY* *2004 – 2006*

Professional Experience

Research Intern, ***Databricks***, *Berkeley, CA, USA* *8/2014 – 11/2014*
Supervisor: Matei Zaharia
Integrated Scala Pickling, our framework for fast, boilerplate-free, extensible serialization focused on distributed programming (OOPSLA'13) into Spark. Developed new function-passing programming model and framework, can be thought of as a generalization of Spark/MapReduce programming model.

Teaching Experience

Lecturer, Co-Designer, *Reactive Programming & Parallelism* *2015*
EPFL Undergraduate course on parallel, distributed, and asynchronous programming (~90 students)

Lecturer, Co-Designer, *Parallel Programming & Data Analysis* *2015*
Upcoming Coursera MOOC on parallel, distributed, and asynchronous programming.

Lead, *Functional Programming Principles in Scala* *2012 – 2014*
Popular Coursera MOOC on functional programming in Scala, with >200,000 participants to date & largest completion rate for a course its size (~19%)

- Lead teaching staff organizing a team of graduate students, managing content production, designed course exercises with cloud-hosted grading, production of lecture videos, etc
- Created extensive course analysis with interactive visualizations; led to a publication at ICSE'14

(Lead) Teaching Assistant, *Programming Principles* *2011-2014*
Required EPFL undergraduate course on functional & logic programming (~160 students)

Instructor, *Scala as a Research Tool* *2013*
ECOOP Tutorial

Research Interests

Concurrent, distributed, data-centric, and data-intensive (big data) programming, from the perspective of programming languages. I work on both theoretical ideas & implementations for the Scala programming language which seek to make it easier to build distributed systems.

Publications

Distributed Programming via Safe Closure Passing *PLACES 2015*
Philipp Haller, Heather Miller
Programming Language Approaches to Communication and Concurrency Centric Systems

Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution *ECOOP 2014*
Heather Miller, Philipp Haller, Martin Odersky
European Conference on Object Oriented Programming

Functional Programming For All! Scaling a MOOC for Students And Professionals Alike *ICSE 2014*
Heather Miller, Philipp Haller, Lukas Rytz, Martin Odersky
ACM SIGSOFT International Conference on Software Engineering

Instant Pickles: Generating Object-Oriented Pickler Combinators for Fast and Extensible Serialization *OOPSLA 2013*
Heather Miller, Philipp Haller, Eugene Burmako, Martin Odersky
ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications

RAY: Integrating Rx and Async for Direct-Style Reactive Streams *REM 2013*
Philipp Haller, Heather Miller
ACM SPLASH Workshop on Reactivity, Events and Modularity

FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction *LCPC 2012*
Aleksandar Prokopec, Heather Miller, Tobias Schlatter, Philipp Haller, Martin Odersky
International Workshop on Languages and Compilers for Parallel Computing
Invited to Revised Selected Papers on the 25th International Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, Vol. 7760, 2013

Tools and Frameworks for Big Learning in Scala: Leveraging the Language for High Productivity and Performance *BigLearn 2011*
Heather Miller, Philipp Haller, Martin Odersky
NIPS Workshop on Parallel and Large-Scale Machine Learning

Parallelizing Machine Learning – Functionally: A Framework and Abstractions for Parallel Graph Processing *Scala 2011*
Philipp Haller, Heather Miller
Scala Workshop

Submitted/In Preparation

Function Passing: A Model for Typed, Distributed Functional Programming
Heather Miller, Philipp Haller

Self-Assembly: Lightweight Language Extension and Datatype Generic Programming, All-in-One!
Heather Miller, Philipp Haller, Bruno C. d. S. Oliveira

Improving Human-Compiler Interaction Through Customizable Type Feedback
Hubert Plociniczak, Heather Miller, Martin Odersky

Selected Tech Reports

Spores, Formally
Heather Miller, Philipp Haller
December 2013

FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction – Proofs
Aleksandar Prokopec, Heather Miller, Philipp Haller
June 2012

Open Source

Scala Programming Language, *member of the Scala team* *2011 –*

- **Scala Spores** (Scala Improvement Proposal SIP-21), ***project lead***
  novel type-based abstraction for using closures safely
  in concurrent and distributed environments
- **Scala Pickling**, ***project lead***
  novel framework for fast, boilerplate-free, extensible serialization.
  Adopted by sbt, the most widely-used build tool for Scala. Popular
  open-source project on GitHub with >480 stars & dozens of contributors
- **Scala Futures & Promises** (Scala Improvement Proposal SIP-14), ***team member***
  unified non-blocking concurrency substrate for
  Scala, Akka, Play, and others
- **Scala Documentation**, ***creator, writer, lead maintainer***
  a central website for community-driven documentation for
  the Scala programming language and core libraries
- **Scaladoc**, ***co-maintainer***
  documentation tool for Scala's official API documentation

Honors

US National Science Foundation Graduate Research Fellowship	*2011 – 2014*
EPFL Outstanding Teaching Award	*2012*
EPFL Computer Science Fellowship	*2009 – 2010*
Most Outstanding Audio Engineering Student, University of Miami	*2009*
Most Outstanding Eta Kappa Nu Student, University of Miami	*2009*
Information Technology Scholarship, University of Miami	*2006 – 2009*
John Farina Family Scholarship, University of Miami	*2006 – 2009*
Eta Kappa Nu	*2008*
Tau Beta Pi	*2008*
SMART US Department of Defense Scholarship Alternate	*2007*
Cooper Union Full Tuition Scholarship	*2004 – 2006*

Selected Talks

Function Passing Style: Typed, Distributed Functional Programming — *Strange Loop 2014*
St. Louis, MO, USA. September 19, 2014

Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution — *ECOOP 2014*
Uppsala, Sweden. August 1, 2014

Functional Programming For All! Scaling a MOOC for Students and Professionals Alike — *ICSE 2014*
Hyderabad, India. June 4, 2014

Academese to English: Scala's Type System, Dependent Types and What It Means To You — *NEScala 2014*
New York, NY, USA. March 1, 2014

Instant Pickles: Generating Object-Oriented Pickler Combinators for Fast and Extensible Serialization — *OOPSLA 2013*
Indianapolis, IN, USA. October 30, 2013

PL Abstractions for Distributed Programming: Pickle Your Spores! — *Indiana University **(invited)***
Bloomington, IN, USA. October 25, 2013

Spores: Distributable Functions in Scala — *Strange Loop 2013*
St. Louis, MO, USA. September 19, 2013

Open Issues in Dataflow Programming — *LaME 2013 **(invited)***
Montpellier, France. July 1, 2013

Scala as a Research Tool — *ECOOP 2013 Tutorial*
Montpellier, France. July 1, 2013

On Pickles & Spores: Improving Scala's Support for Distributed Programming — *ScalaDays 2013*
New York, NY, USA. June 12, 2013

Futures & Promises in Scala 2.10 — *PhillyETE 2013 **(invited)***
Philadelphia, PA, USA. April 2, 2013

I am also a frequent speaker in industry, at industrial conferences, developer "meet-ups", and everything in between. Some such events include:

f(by) (11/2014, Minsk, Belarus), SF Scala (11/2014, SF, USA), Scalapeño (9/2014, Tel Aviv, Israel), SoundCloud TechTalks (7/2014, Berlin, Germany), **Scala Days** (6/2014, Berlin, Germany), NEScala (3/2014, NYC, USA), amongst others.

External Activities

Scalawags Monthly Podcast, co-host — *2014 –*

External Service

Curry On 2015, organizer (co-chair)	*7/2015*
ECOOP 2015, organizing committee member (sponsorship)	*7/2015*
PLE 2015, program committee member	*7/2015*
DSLDI 2015, program committee member	*7/2015*
Scala Symposium 2015, organizer (co-chair)	*6/2015*
POPL 2015, artifact evaluation committee member	*1/2015*
Scala Workshop 2014, organizer (co-chair)	*7/2014*
Scala Workshop 2013, organizer (co-chair)	*7/2013*

External Reviewer for: ECOOP 2013, Scala 2013
Editor of proceedings for: Scala 2015, Scala 2014, Scala 2013

Students Supervised[1]

Louis Bliss, *Incremental Picklers for Scala Pickling* — *9/2013 – 1/2014*
M.Sc. level, co-supervision with Philipp Haller

Thaddée Yann Tyl, *Learning Scala Style* — *2/2013 – 6/2013*
M.Sc. thesis

Tobias Schlatter, *FlowSeqs: Barrier-Free ParSeqs* — *9/2012 – 1/2013*
M.Sc. level, co-supervision w/ Philipp Haller & Aleksandar Prokopec

Tobias Schlatter, *Multi-Lane FlowPools* — *2/2012 – 6/2012*
M.Sc. level, co-supervision w/ Philipp Haller & Aleksandar Prokopec

Pierre Grydbeck, *Parallel Machine Learning: An Expectation Maximization Algorithm for Gaussian Mixture Models* — *2/2012 – 6/2012*
M.Sc. level, co-supervision with Philipp Haller

Bruno Studer, *Parallel Machine Learning: Collaborative Filtering via Alternating Least Squares* — *2/2012 – 6/2012*
B.Sc. level, co-supervision with Philipp Haller

Stanislav Peshterliev, *Parallel Natural Language Processing Algorithms in Scala* — *9/2011 – 1/2012*
M.Sc. level, co-supervision with Philipp Haller

Olivier Blanvillain & Louis Bliss, *Parallelization of a Collaborative Filtering Algorithm with Menthor* — *9/2011 – 1/2012*
B.Sc. level, co-supervision with Philipp Haller

Florian Gysin, *Improving Parallel Graph Processing Through the Introduction of Parallel Collections* — *9/2011 – 1/2012*
M.Sc. level, co-supervision with Philipp Haller

Georges Discry, *Extending the Menthor Framework for Parallel Graph Processing to Distributed Computing* — *2/2011 – 6/2011*
M.Sc. level, co-supervision with Philipp Haller

[1] At EPFL, research groups offer substantial projects for B.Sc./M.Sc. students to complete for credit. EPFL PhD students design and supervise these projects, as well as M.Sc. thesis projects.

References

Martin Odersky
Faculty of Computer, Communication, and Information Science
École Polytechnique Fédérale de Lausanne
☎ +41 21 693 68 63
✉ martin.odersky@epfl.ch

Philipp Haller
School of Computer Science and Communication
KTH Royal Institute of Technology
☎ +41 76 205 39 32
✉ phaller@kth.se

Matei Zaharia
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
☎ +1-510-610-0001
✉ matei@mit.edu

Marius Eriksen
Twitter
✉ marius@twitter.com