A Type-Based Foundation for Closure-Passing in the Age of Concurrency and Distribution

Heather Miller, Philipp Haller

EPFL, Switzerland
KTH Royal Institute of Technology, Sweden

Abstract: Functional programming (FP) is regularly touted as the way forward for bringing parallel, concurrent, and distributed programming to the mainstream. However, despite this established viewpoint, reliably distributing function closures over a network, or using them in concurrent environments nonetheless remains a challenge across FP and OO languages. Our work on Spores takes a step towards more principled distributed and concurrent programming by introducing a new closure-like abstraction and type system that can guarantee closures to be serializable, thread-safe, or have custom user-defined properties. In ongoing work we explore the combination of Spores and Scala Pickling to provide a common substrate for type-safe, performant data-intensive applications.

1 Motivation

With the growing trend towards decentralized computing–cloud computing, mobile applications, and big data–distributed programming has entered the mainstream. Meanwhile, functional programming (FP) has been gaining traction, as is evidenced by the ongoing trend of imperative OO languages being extended with FP features, such as lambdas in Java 8, C++ 11, and VB 9, the perceived importance of FP in empirical studies on software developers, and the popularity of FP massive online open courses (MOOCs).¹

One reason for the rise in popularity of FP features within object-oriented communities is the observation that a functional style simplifies reasoning about data in parallel, concurrent, and distributed code. Distributed data-parallel frameworks like MapReduce and Spark are designed around FP 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, for both OO and FP languages, there still exist numerous hurdles for even these most basic functional building blocks to overcome in order to be reliable and easy to reason about in a concurrent or distributed setting. Closure-related hazards related to concurrency and distribution include: (1) accidental capture of non-serializable variables; (2) transitive references that inadvertently hold on to large object graphs, creating memory leaks; (3)

¹In the interest of space, references are omitted from this extended abstract, and are found in the paper: 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, LNCS, pages 308–333. Springer, 2014.
capturing references to mutable objects, leading to race conditions in a concurrent setting; (4) unknowingly accessing object members that are not constant, leading to semantic inconsistencies when closures are distributed.

2 Spores
As a step towards more principled function-passing style we introduce a type-based foundation for closures, 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. 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, by principle of a type-based approach, spores can potentially benefit from optimization, further safety checks via type system extensions, and verification opportunities.

3 Formalization and Type System
In our accompanying publication and technical report, we present a formalization of spores with type constraints in the context of a standard typed lambda calculus with records, and prove soundness of the type system.

4 Empirical Studies
We evaluate the practicality and the benefits of using spores as an alternative to normal closures in Scala. First, we measure the impact of introducing spores in existing programs. Second, 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.

Using spores instead of closures We analyzed a number of real Scala programs: (1) general, closure-heavy code, taken from the popular MOOC on FP Principles in Scala, (2) parallel programs using parallel collections, and (3) distributed programs based on Spark. Our results show that of all closures, 90% could be converted without extra effort. For those that had to be manually converted, on average only 1.4 LOC had to be changed. This suggests that programs using closures in non-trivial ways can typically be converted to using spores with little effort.

Spores and Apache Spark To evaluate both benefit and overhead of using spores in real distributed programs, we studied the codebases of 7 large open-source applications using Spark. Our results show that of all closures passed to Spark’s RDD.map method, about 67.2% do not capture any variables; these closures could be automatically converted to spores. The remaining 32.8% of closures capture 1.39 variables on average. This indicates that unchecked patterns for serializable closures are widespread in real programs, and that benefiting from the static guarantees of spores would require only little syntactic overhead.