Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution

Heather Miller, Philipp Haller[1], and Martin Odersky

EPFL and Typesafe, Inc.[1]
{heather.miller, martin.odersky}@epfl.ch and philipp.haller@typesafe.com[1]

Abstract. Functional programming (FP) is regularly touted as the way forward for bringing parallel, concurrent, and distributed programming to the mainstream. The popularity of the rationale behind this viewpoint has even led to a number of object-oriented (OO) programming languages outside the Smalltalk tradition adopting functional features such as lambdas and thereby function closures. However, despite this established viewpoint of FP as an enabler, reliably distributing function closures over a network, or using them in concurrent environments nonetheless remains a challenge across FP and OO languages. This paper takes a step towards more principled distributed and concurrent programming by introducing 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.

Keywords: closures, functions, distributed programming, concurrent programming, type systems

1 Introduction

With the growing trend towards cloud computing, mobile applications, and big data, 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.

Meanwhile, at the same time, functional programming has been undeniably gaining traction in recent years, 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 [7], C++11 [9], and Visual Basic 9 [16], the perceived importance of functional programming in general empirical studies on software developers [17], and the popularity of functional programming massively online open courses (MOOCs) [20].

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. Examples across functional and object-oriented paradigms include Java 8's monadic-style optionally parallel collections [7], Scala's parallel [25] and concurrent dataflow [26] collections, Data Parallel Haskell [2], CnC [1], Nova [3], and Haskell's `Par` monad [12] to name a few.

In the context of distributed programming, data-parallel frameworks like MapReduce [4] and Spark [31] 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:

```
1    sendFunc :: SendPort (Int -> Int) -> Int -> ProcessM ()
2    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):

```
1  class MyCoolRddApp {
2    val log = new Log(...)
3    def shift(p: Int): Int = ...
4    ...
5    def work(rdd: RDD[Int]) {
6      rdd.map(x => x + shift(x)).reduce(...)
7    }
8  }
```

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 paper 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.

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 [19].

- **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.

1.1 Selected Related Work

Cloud Haskell [5] 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 [9] 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 [13] 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 [7] 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.

We discuss other related work in Section 7.

1.2 Contributions

This paper makes 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.

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, 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 2.2 we describe the `Spore` type. In Section 2.4 we informally describe the type system, and how to add user-defined constraints to customize what types a spore can capture.

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

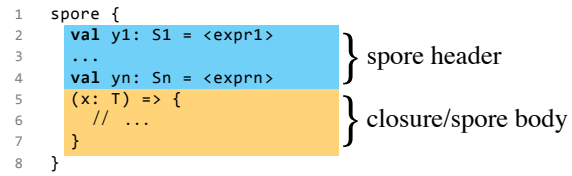

Fig. 1: The syntactic shape of a spore.

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 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.

```
{
  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.

Fig. 2: The evaluation semantics of a spore is equivalent to that of a closure, obtained by simply leaving out the spore marker.

Evaluation Semantics The evaluation semantics of a spore is equivalent to a closure obtained by leaving out the `spore` marker, as shown in Figure 2. In Scala, the block shown in Figure 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 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, such as `this`. However, in systems like Spark, the above shape is merely a convention that is not enforced.

2.2 The `Spore` Type

Figure 3 shows Scala's arity-1 function type and the arity-1 spore type.[2] Functions are

[2] For simplicity, we omit `Function1`'s definitions of the `andThen` and `compose` methods.

```
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.

Fig. 3: The `Spore` type.

```
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.

Fig. 4: An example of the `Captured` type member.
Note: we omit the `Excluded` type member for simplicity; we detail it later in Section 2.4.

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 2.4 introduces the `Excluded` type member.

2.3 Basic Usage

Definition A spore can be defined as shown in Figure 4a, with its corresponding type shown in Figure 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 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[3] macro[4] 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.[5]

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)`.[6]

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

```
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 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.

---

[3] 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.

[4] 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.

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

[6] 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 [23], Section 3.1).

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 paper, 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 [29] 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 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.[7]

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 [19]. To be able to pickle a spore, it's necessary that all its captured types have an instance of `Pickler`.[8]

Spores allow expressing such a requirement using a notion of implicit *properties*. The idea is that if there is an implicit value[9] 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:[10]

[7] 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.

[8] A spore can be pickled by pickling its environment and the fully-qualified class name of its corresponding function class.

[9] An implicit value is a value in *implicit scope* that is statically selected based on its type.

[10] In the code example, `implicitly[T]` returns the uniquely-defined implicit value of `T` which is in scope at the invocation site.

```
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:

```
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]
}
```

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 [19].

Transitive properties expressed using known techniques, *e.g.,* generics (Zibin etal's OIGJ system [32] 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[O, 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).

3 Formalization

$t ::= x$	variable
$\mid (x : T) \Rightarrow t$	abstraction
$\mid t\ t$	application
$\mid$ `let` $x = t$ `in` t	let binding
$\mid \{\overline{l = t}\}$	record construction
$\mid t.l$	selection
$\mid$ `spore` $\{\ \overline{x : T = t}\ ; \overline{pn}; (x : T) \Rightarrow t\ \}$	spore
$\mid$ `import` pn `in` t	property import
$\mid t$ `compose` t	spore composition
$v ::= (x : T) \Rightarrow t$	abstraction
$\mid \{\overline{l = v}\}$	record value
$\mid$ `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
$\mid \mathcal{S}$	
$\mathcal{S} ::= T \Rightarrow T\ \{$ `type` $\mathcal{C} = \overline{T}\ ;\ \overline{pn}\ \}$	spore type
$\mid T \Rightarrow T\ \{$ `type` $\mathcal{C}\ ;\ \overline{pn}\ \}$	abstract spore type
$P \in pn \to \mathcal{T}$	property map
$\mathcal{T} \in \mathcal{P}(T)$	type family
$\Gamma ::= \overline{x : T}$	type environment
$\Delta ::= \overline{pn}$	property environment

Fig. 5: 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 [24]. Figure 5 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 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 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.

3.1 Subtyping

Figure 6 shows the subtyping rules; rules S-Rec and S-Fun are standard [24].

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 ($M_2 = \texttt{type}\ \mathcal{C}$). 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 $\mathcal{C}$ 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 2.4).

Rule S-SporeFun expresses the fact that spore types are refinements of their corresponding function types, giving rise to a subtyping relationship.

$$\text{S-Rec}\quad \frac{\overline{l'} \subseteq \overline{l} \qquad l_i = l'_i \to T_i <: T'_i \wedge T'_i <: T_i}{\{\overline{l : T}\} <: \{\overline{l' : T'}\}} \qquad \text{S-Fun}\quad \frac{T_2 <: T_1 \qquad R_1 <: R_2}{T_1 \Rightarrow R_1 <: T_2 \Rightarrow R_2}$$

$$\text{S-Spore}\quad \frac{T_2 <: T_1 \qquad R_1 <: R_2 \qquad \overline{pn'} \subseteq \overline{pn} \qquad M_1 = M_2 \vee M_2 = \texttt{type}\ \mathcal{C}}{T_1 \Rightarrow R_1\ \{\ M_1\ ;\ \overline{pn}\ \} <: T_2 \Rightarrow R_2\ \{\ M_2\ ;\ \overline{pn'}\ \}}$$

$$\text{S-SporeFun}\quad T_1 \Rightarrow R_1\ \{\ M\ ;\ \overline{pn}\ \} <: T_1 \Rightarrow R_1$$

Fig. 6: Subtyping

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 [10]; 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 $\mathcal{T}$. 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 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

$$\textsc{T-Var}\ \frac{x : T \in \Gamma}{\Gamma; \Delta \vdash x : T} \qquad \textsc{T-Sub}\ \frac{\Gamma; \Delta \vdash t : T' \quad T' <: T}{\Gamma; \Delta \vdash t : T} \qquad \textsc{T-Abs}\ \frac{\Gamma, x : T_1; \Delta \vdash t : T_2}{\Gamma; \Delta \vdash (x : T_1) \Rightarrow t : T_1 \Rightarrow T_2}$$

$$\textsc{T-App}\ \frac{\Gamma; \Delta \vdash t_1 : T_1 \Rightarrow T_2 \quad \Gamma; \Delta \vdash t_2 : T_1}{\Gamma; \Delta \vdash (t_1\ t_2) : T_2} \qquad \textsc{T-Let}\ \frac{\Gamma; \Delta \vdash t_1 : T_1 \quad \Gamma, x : T_1; \Delta \vdash t_2 : T_2}{\Gamma; \Delta \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : T_2}$$

$$\textsc{T-Rec}\ \frac{\Gamma; \Delta \vdash \overline{t : T}}{\Gamma; \Delta \vdash \{\overline{l = t}\} : \{\overline{l : T}\}} \qquad \textsc{T-Sel}\ \frac{\Gamma; \Delta \vdash t : \{\overline{l : T}\}}{\Gamma; \Delta \vdash t.l_i : T_i} \qquad \textsc{T-Imp}\ \frac{\Gamma; \Delta, pn \vdash t : T}{\Gamma; \Delta \vdash \texttt{import}\ pn\ \texttt{in}\ t : T}$$

$$\textsc{T-Spore}\ \frac{\forall s_i \in \overline{s}.\ \Gamma; \Delta \vdash s_i : S_i \qquad \overline{y : S}, x : T_1; \Delta \vdash t_2 : T_2 \qquad \forall pn \in \Delta, \Delta'.\ \overline{S} \subseteq P(pn)}{\Gamma; \Delta \vdash \texttt{spore}\ \{\ \overline{y : S = s}\ ; \Delta'; (x : T_1) \Rightarrow t_2\ \} : \qquad T_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathcal{C} = \overline{S}\ ;\ \Delta, \Delta'\ \}}$$

$$\textsc{T-Comp}\ \frac{\Gamma; \Delta \vdash t_1 : T_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathcal{C} = \overline{S}\ ;\ \Delta_1\ \} \qquad \Gamma; \Delta \vdash t_2 : U_1 \Rightarrow T_1\ \{\ \texttt{type}\ \mathcal{C} = \overline{R}\ ;\ \Delta_2\ \} \qquad \Delta' = \{pn \in \Delta_1 \cup \Delta_2 \mid \overline{S} \subseteq P(pn) \wedge \overline{R} \subseteq P(pn)\}}{\Gamma; \Delta \vdash t_1\ \texttt{compose}\ t_2 : U_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathcal{C} = \overline{S}, \overline{R}\ ;\ \Delta'\ \}}$$

Fig. 7: Typing rules

that there is no rule for spore application, since there is a subtyping relationship between spores and functions (see Section 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).

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
$$\frac{\Delta = \{p \mid p \in \overline{pn}, \overline{qn}.\ \overline{T} \subseteq P(p) \wedge \overline{S} \subseteq P(p)\}}{\begin{array}{c}\texttt{spore}\ \{\ \overline{x : T = v}; \overline{pn}; (x' : T') \Rightarrow t\ \}\ \texttt{compose spore}\ \{\ \overline{y : S = w}; \overline{qn}; (y' : S') \Rightarrow t'\ \} \rightarrow \\ \texttt{spore}\ \{\ \overline{x : T = v}, \overline{y : S = w}; \Delta; (y' : S') \Rightarrow \texttt{let}\ z' = t'\ \texttt{in}\ [x' \mapsto z']t\}\end{array}}$$

Fig. 8: Operational Semantics[11]

H-InsSpore1
$$\frac{\forall t_i \in \overline{t}.\ insert(pn, t_i) = t'_i \qquad insert(pn, t) = t'}{\begin{array}{c}insert(pn, \texttt{spore}\ \{\ \overline{x : T = t}; \overline{pn}; (x' : T) \Rightarrow t\ \}) = \\ \texttt{spore}\ \{\ \overline{x : T = t'}; \overline{pn}, pn; (x' : T) \Rightarrow t'\ \}\end{array}}$$

H-InsSpore2
$$\frac{insert(pn, t) = t'}{\begin{array}{c}insert(pn, \texttt{spore}\ \{\ \overline{x : T = v}; \overline{pn}; (x' : T) \Rightarrow t\ \}) = \\ \texttt{spore}\ \{\ \overline{x : T = v}; \overline{pn}, pn; (x' : T) \Rightarrow t'\ \}\end{array}}$$

H-InsApp
$$insert(pn, t_1\ t_2) = insert(pn, t_1)\ insert(pn, t_2)$$

H-InsSel
$$insert(pn, t.l) = insert(pn, t).l$$

Fig. 9: Helper function $insert$

3.3 Operational semantics

Figure 8 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. 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 9 (we omit rules for `compose` and `let`; they are analogous to rules H-InsApp and H-InsSel).

[11] For the sake of brevity, here we omit the standard evaluation rules. The complete set of evaluation rules can be found in the accompanying technical report [18]

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.

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 [30]. A complete proof of soundness appears in the companion technical report [18]. In addition to standard lemmas, we also prove a lemma specific to our type system, Lemma 1, which ensures types are preserved under property import. Soundness of the type system follows from Theorem 1 and Theorem 2.

Theorem 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 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 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 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 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 $\Gamma; \Delta \vdash t : T$.

3.5 Relation to spores in Scala

The type soundness proof (see Section 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 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.

$$
\begin{array}{lll}
t ::= \ldots & & \text{terms} \\
\quad \mid \texttt{spore}\ \{\ \overline{x : T = t}\ ;\ \overline{T}; \overline{pn}; (x : T) \Rightarrow t\ \} & & \text{spore} \\
\\
v ::= \ldots & & \text{values} \\
\quad \mid \texttt{spore}\ \{\ \overline{x : T = v}\ ;\ \overline{T}; \overline{pn}; (x : T) \Rightarrow t\ \} & & \text{spore value} \\
\\
\mathcal{S} ::= T \Rightarrow T\ \{\ \texttt{type}\ \mathcal{C} = \overline{T}\ ;\ \texttt{type}\ \mathcal{E} = \overline{T}\ ;\ \overline{pn}\ \} & & \text{spore type} \\
\quad \mid T \Rightarrow T\ \{\ \texttt{type}\ \mathcal{C}\ ;\ \texttt{type}\ \mathcal{E} = \overline{T}\ ;\ \overline{pn}\ \} & & \text{abstract spore type}
\end{array}
$$

Fig. 10: Core language syntax extensions

S-ESpore

$$
\frac{\overline{pn'} \subseteq \overline{pn} \qquad M_1 = M_2 \vee M_2 = \texttt{type}\ \mathcal{C} \qquad T_2 <: T_1 \qquad R_1 <: R_2 \qquad \forall T' \in \overline{U'}.\ \exists T \in \overline{U}.\ T' <: T}{T_1 \Rightarrow R_1\ \{\ M_1\ ;\ \texttt{type}\ \mathcal{E} = \overline{U}\ ;\ \overline{pn}\ \} \quad <: T_2 \Rightarrow R_2\ \{\ M_2\ ;\ \texttt{type}\ \mathcal{E} = \overline{U'}\ ;\ \overline{pn'}\ \}}
$$

S-ESporeFun

$$
T_1 \Rightarrow R_1\ \{\ M\ ;\ E\ ;\ \overline{pn}\ \} <: T_1 \Rightarrow R_1
$$

Fig. 11: Subtyping extensions

E-EAppSpore

$$
\frac{\forall pn \in \overline{pn}.\ \overline{T} \subseteq P(pn) \qquad \forall T_i \in \overline{T}.\ T_i \notin \overline{U}}{\texttt{spore}\ \{\ \overline{x : T = v}\ ;\ \overline{U}\ ;\ \overline{pn}\ ;\ (x' : T) \Rightarrow t\ \}\ v' \rightarrow \quad \overline{[x \mapsto v]}[x' \mapsto v']t}
$$

E-EComp3

$$
\frac{\Delta = \{p \mid p \in \overline{pn}, \overline{qn}.\ \overline{T} \subseteq P(p) \wedge \overline{S} \subseteq P(p)\} \qquad \overline{V} = (\overline{U} \setminus \overline{S}) \cup (\overline{U'} \setminus \overline{T})}{\begin{array}{c}\texttt{spore}\ \{\ \overline{x : T = v}\ ;\ \overline{U}\ ;\ \overline{pn}\ ;\ (x' : T') \Rightarrow t\ \}\ \texttt{compose} \\ \texttt{spore}\ \{\ \overline{y : S = w}\ ;\ \overline{U'}\ ;\ \overline{qn}\ ;\ (y' : S') \Rightarrow t'\ \} \rightarrow \quad \texttt{spore}\ \{\ \overline{x : T = v}, \overline{y : S = w}\ ;\ \overline{V}\ ;\ \Delta\ ; \\ (y' : S') \Rightarrow \texttt{let}\ z' = t'\ \texttt{in}\ [x' \mapsto z']t\ \}\end{array}}
$$

Fig. 12: Operational semantics extensions

3.6 Excluded types

This section shows how the formal model can be extended with excluded types as described above (see Section 2.4). Figure 10 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 $\texttt{type}\ \mathcal{E} = \overline{T}$ specifying the excluded types.

Figure 11 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 12 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 13 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-

T-ESpore

$$\frac{\forall s_i \in \overline{s}.\, \Gamma; \Delta \vdash s_i : S_i \qquad \overline{y : S}, x : T_1; \Delta \vdash t_2 : T_2 \qquad \forall pn \in \Delta, \Delta'.\, \overline{S} \subseteq P(pn) \qquad \forall S_i \in \overline{S}.\, \forall U_j \in \overline{U}.\, \neg(S_i <: U_j)}{\Gamma; \Delta \vdash \texttt{spore}\ \{\ \overline{y : S = s}\ ; \overline{U}; \Delta'; (x : T_1) \Rightarrow t_2\ \} : T_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathcal{C} = \overline{S}\ ;\ \texttt{type}\ \mathcal{E} = \overline{U}\ ;\ \Delta, \Delta'\ \}}$$

T-EComp

$$\frac{\Gamma; \Delta \vdash t_1 : T_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathcal{C} = \overline{S}\ ;\ \texttt{type}\ \mathcal{E} = \overline{U}\ ;\ \Delta_1\ \} \qquad \Gamma; \Delta \vdash t_2 : U_1 \Rightarrow T_1\ \{\ \texttt{type}\ \mathcal{C} = \overline{R}\ ;\ \texttt{type}\ \mathcal{E} = \overline{U'}\ ;\ \Delta_2\ \} \qquad \Delta' = \{pn \in \Delta_1 \cup \Delta_2 \mid \overline{S} \subseteq P(pn) \wedge \overline{R} \subseteq P(pn)\} \qquad \overline{V} = (\overline{U} \setminus \overline{R}) \cup (\overline{U'} \setminus \overline{S})}{\Gamma; \Delta \vdash t_1\ \texttt{compose}\ t_2 : U_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathcal{C} = \overline{S}, \overline{R}\ ;\ \texttt{type}\ \mathcal{E} = \overline{V}\ ;\ \Delta'\ \}}$$

Fig. 13: Typing extensions

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.

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 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

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	

Fig. 14: Evaluating the practicality of using spores in place of normal closures

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 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 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.

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.

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-learning ★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%

Fig. 15: 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.

Results The results are shown in Figure 14. 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 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.

5.2 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 15 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 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.

5.3 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.

6 Case Study

Frameworks like MapReduce [4] and Apache Spark [31] 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 following typical pattern was extracted from a code base used in production:

```
1  class GenericOp(sc: SparkContext, mapping: Map[String, String]) {
2    private var cachedSessions: spark.RDD[Session] = ...
3
```

```
4     def doOp(keyList: List[...], ...): Result = {
5       val localMapping = mapping
6
7       val mapFun: Session => (List[String], GenericOpAggregator) = { s =>
8         (keyList, new GenericOpAggregator(s, localMapping))
9       }
10
11      val reduceFun: (GenericOpAggregator, GenericOpAggregator) =>
12        GenericOpAggregator = { (a, b) => a.merge(b) }
13
14      cachedSessions.map(mapFun).reduceByKey(reduceFun).collectAsMap
15    }
16  }
```

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:

```
1  val mapFun: Spore[Session, (List[String], GenericOpAggregator)] =
2    spore { val localMapping = mapping
3      (s: Session) => (keyList, new GenericOpAggregator(s, localMapping)) }
4  val reduceFun: Spore[(GenericOpAggregator, GenericOpAggregator),
5                       GenericOpAggregator] =
6    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.

7 Other Related Work

Parallel closures [14] 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 [8] provides a concurrency model for JavaScript, similar to parallel closures; however, capturing variables in closures is currently not supported.

ML5 [22] 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 [21]. 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 [11] 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 Section 1.1 also applies to HdpH. Delimited continuations [27] 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 [6] 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 [28]. Python's standard serialization module, `pickle`, does not support serializing closures. Dill [15] extends Python's `pickle` module, adding support for functions and closures, but without constraints.

8 Conclusion

We've 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.

Acknowledgements

We would like to thank the anonymous ECOOP 2014 referees for their thorough reviews and helpful suggestions which greatly improved the quality of the paper. Heather Miller was supported by a US National Science Foundation Graduate Research Fellowship.

References

1. Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, et al. Concurrent collections. *Scientific Programming*, 18(3), 2010.
2. M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A status report. In *Proc. DAMP Workshop*, pages 10–18. ACM, 2007.
3. A. Collins, D. Grewe, V. Grover, S. Lee, and A. Susnea. NOVA: A functional language for data parallelism. Technical Report NVR-2013-002, NVIDIA Corporation, July 2013.
4. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
5. J. Epstein, A. P. Black, and S. Peyton-Jones. Towards Haskell in the cloud. In *Proc. Haskell Symposium*, pages 118–129. ACM, 2011.

6. G. Germain. Concurrency oriented programming in Termite Scheme. In *Erlang Workshop*, page 20. ACM, 2006.
7. B. Goetz. JSR 335: Lambda expressions for the Java programming language. `https://jcp.org/en/jsr/detail?id=335`, 2013.
8. S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. River trail: a path to parallelism in JavaScript. In *OOPSLA*, pages 729–744, 2013.
9. International Standard ISO/IEC 14882:2011. *Programming Languages – C++*. International Organization for Standards, 2011.
10. J. R. Lewis, J. Launchbury, E. Meijer, and M. Shields. Implicit parameters: Dynamic scoping with static types. In *POPL*, pages 108–118, 2000.
11. P. Maier and P. W. Trinder. Implementing a high-level distributed-memory parallel Haskell in Haskell. In *IFL*, volume 7257, pages 35–50. Springer, 2011.
12. S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. In *Proc. Haskell Symposium*, pages 71–82. ACM, 2011.
13. N. Matsakis. Fn types in Rust, take 3. `http://smallcultfollowing.com/babysteps/blog/2013/10/10/fn-types-in-rust`, 2013.
14. N. D. Matsakis. Parallel closures: a new twist on an old idea. In *HotPar*. USENIX, 2012.
15. M. M. McKerns, L. Strand, T. Sullivan, A. Fang, and M. A. Aivazis. Building a framework for predictive science. In *Proc. of the 10th Python in Science Conf.*, 2011.
16. E. Meijer. Confessions of a used programming language salesman. In *OOPSLA*, 2007.
17. L. A. Meyerovich and A. S. Rabkin. Empirical analysis of programming language adoption. In *OOPSLA*, 2013.
18. H. Miller and P. Haller. Spores, formally. Technical Report EPFL-REPORT-191240, Department of Computer Science, EPFL, Lausanne, Switzerland, December 2013.
19. H. Miller, P. Haller, E. Burmako, and M. Odersky. Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization. In *OOPSLA*, pages 183–202, 2013.
20. H. Miller, P. Haller, L. Rytz, and M. Odersky. Functional programming for all! Scaling a MOOC for students and professionals alike. In *ICSE*, pages 265–263, 2014.
21. J. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst*, 21(3):527–568, 1999.
22. T. Murphy VII, K. Crary, and R. Harper. Type-safe distributed programming with ML5. In *TGC*, volume 4912, pages 108–123. Springer, 2007.
23. M. Odersky. The Scala language specification, 2013.
24. B. C. Pierce. *Types and programming languages*. MIT Press, 2002.
25. A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky. A generic parallel collection framework. In *Euro-Par*, volume 6853, pages 136–147. Springer, 2011.
26. A. Prokopec, H. Miller, T. Schlatter, P. Haller, and M. Odersky. Flowpools: A lock-free deterministic concurrent dataflow abstraction. In *LCPC*, pages 158–173. Springer, 2012.
27. T. Rompf, I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *ICFP*, pages 317–328. ACM, 2009.
28. A. Schwendner. Distributed functional programming in Scheme. Master's thesis, Massachusetts Institute of Technology, 2009.
29. Typesafe. Akka. `http://akka.io/`, 2009.
30. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput*, 115(1):38–94, Nov. 1994.
31. M. Zaharia, M. Chowdhury, T. Das, A. Dave, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. USENIX, 2012.
32. Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst. Ownership and immutability in generic java. In *OOPSLA*, pages 598–617. ACM, 2010.