Spores, Formally

Heather Miller and Philipp Haller

December 2013

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 = ...
```

This way, libraries and frameworks can enforce the use of spores instead of plain closures, thereby reducing the risk for common programming errors.

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

[1] For simplicity we omit definitions of the 'andThen' and 'compose' methods in the definition of 'Function1'.

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:

```
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.

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

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
$\mid \mathcal{S}$	
$\mathcal{S} ::= T \Rightarrow T\ \{\ \texttt{type}\ \mathcal{C} = \overline{T}\ ;\ \overline{pn}\ \}$	spore type
$\mid T \Rightarrow T\ \{\ \texttt{type}\ \mathcal{C}\ ;\ \overline{pn}\ \}$	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

Figure 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 **?**. Figure 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 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 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.

2.1 Subtyping

Figure 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 ($M_2 = \texttt{type}\ \mathcal{C}$). 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 **??**).

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

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

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

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

Figure 2: Subtyping

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 **?**; 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

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

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

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

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

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

Figure 3: Typing rules

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 2.1). 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

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
$$\frac{t_k \to t_k'}{\begin{array}{l}\texttt{spore } \{\ \overline{x : T = v}, x_k : T_k = t_k, \overline{x' : T' = t'}\ ; (x : T) \Rightarrow t\ \} \to \\ \quad \texttt{spore } \{\ \overline{x : T = v}, x_k : T_k = t_k', \overline{x' : T' = t'}\ ; (x : T) \Rightarrow t\ \}\end{array}}$$

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

Figure 4: Operational Semantics[3]

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).

2.3 Operational semantics

Figure 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 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.

H-InsSpore1

$$\frac{\forall t_i \in \overline{t}.\ insert(pn, t_i) = t_i' \qquad insert(pn, t) = t'}{insert(pn, \texttt{spore}\ \{\ \overline{x : T = t}; \overline{pn}; (x' : T) \Rightarrow t\ \}) = \texttt{spore}\ \{\ \overline{x : T = t'}; \overline{pn}, pn; (x' : T) \Rightarrow t'\ \}}$$

H-InsSpore2

$$insert(pn, \texttt{spore}\ \{\ \overline{x : T = v}; \overline{pn}; (x' : T) \Rightarrow t\ \}) = \texttt{spore}\ \{\ \overline{x : T = v}; \overline{pn}, pn; (x' : T) \Rightarrow t\ \}$$

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: Helper function *insert*

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 **?**. A complete proof of soundness appears in the companion technical report **?**. In addition to standard lemmas, such as Lemma 2.4 and Lemma 2.5, we also prove a lemma specific to our type system, namely Lemma 2.3, which ensures types are preserved under property import. Soundness of the type system follows from Theorem 2.2 and Theorem 2.6.

Lemma 2.1. (Canonical forms)

[3]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 **?**

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* `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 $T \Rightarrow R$* $\{$ `type` $\mathcal{C} = \overline{S}\ ;\ \overline{pn}\ \}$*, then v is* `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 1, values in the core language can have three forms: $(x : T) \Rightarrow t$, $\{\overline{l = v}\}$, and `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 $T_1 \Rightarrow R_1$, $T_1 \Rightarrow R_1$ $\{$ `type` $\mathcal{C} = \overline{S}\ ;\ \overline{pn}\ \}$, or $T_1 \Rightarrow R_1$ $\{$ `type` $\mathcal{C}\ ;\ \overline{pn}\ \}$ 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 $T_1 \Rightarrow R_1$ $\{$ `type` $\mathcal{C} = \overline{S}\ ;\ \overline{pn}\ \}$ or type $T_1 \Rightarrow R_1$ $\{$ `type` $\mathcal{C}\ ;\ \overline{pn}\ \}$, then according to the grammar and (T-Spore) v must be `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 2.2. (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: $t =$ `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 `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` $\mathcal{C} = \overline{S}\ ;\ \Delta_1\ \}$ and $\vdash t_2 : U_1 \Rightarrow T_1$ $\{$ `type` $\mathcal{C} = \overline{R}\ ;\ \Delta_2\ \}$. 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` $\{ \overline{y : S = v}\ ; \Delta_1; (x : T_1) \Rightarrow s_1 \}$ and $v_2 =$ `spore` $\{ \overline{z : R = w}\ ; \Delta_2; (u : U_1) \Rightarrow s_2 \}$. Thus, by (E-Comp3), $t \to t'$ for some t'.

□

Lemma 2.3. (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: $t = \texttt{spore}\ \{\ \overline{x : S = t}\ ; \Delta'; (x : T_1) \Rightarrow t_2\ \}$, $\forall t_i \in \overline{t}.\ \vdash t_i : S_i$, 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: $t = t_1\ \texttt{compose}\ t_2$ and $\vdash t_1 : T_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathcal{C} = \overline{S}\ ;\ \Delta_1\ \}$ and $\vdash t_2 : U_1 \Rightarrow T_1\ \{\ \texttt{type}\ \mathcal{C} = \overline{R}\ ;\ \Delta_2\ \}$. 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, $v_1 = \texttt{spore}\ \{\ \overline{y : S = v}\ ; \Delta_1; (x : T_1) \Rightarrow s_1\ \}$ and $v_2 = \texttt{spore}\ \{\ \overline{z : R = w}\ ; \Delta_2; (u : U_1) \Rightarrow s_2\ \}$. Thus, by (E-Comp3), $t \to t'$ for some t'.

□

Lemma 2.4. (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 2.5. (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 2.6. (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 2.3, $\Gamma; \Delta \vdash insert(pn, s) : T$.

- 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 2.4, $\Gamma; \Delta \vdash [x \mapsto v]t_2 : S_2$.

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 2.5, $\Gamma, \overline{x : T}, y : S_1; \Delta \vdash t_2 : S_2$. By Lemma 2.5, $\Gamma, \overline{x : T}; \Delta \vdash v' : S_1$. By Lemma 2.4, $\Gamma, \overline{x : T}; \Delta \vdash [y \mapsto v']t_2 : S_2$. By (T-Spore), we also have $\forall v_i \in \overline{v}.\ \Gamma; \Delta \vdash v_i : T_i$. By Lemma 2.4, $\Gamma; \Delta \vdash \overline{[x \mapsto v]}[y \mapsto v']t_2 : S_2$.

- Case T-Spore: $t = \texttt{spore}\ \{\ \overline{y : S = s}\ ; \Delta'; (x : T_1) \Rightarrow t_2\ \}$ and $T = T_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathcal{C} = \overline{S}\ ;\ \Delta, \Delta'\ \}$. 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 $T = T_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathcal{C} = \overline{S}, \overline{R}\ ;\ \Delta_3\ \}$. By (T-Comp), $\Gamma \vdash s_1 : U_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathcal{C} = \overline{S}\ ;\ \Delta_1\ \}$ and $\Gamma \vdash s_2 : T_1 \Rightarrow U_1\ \{\ \texttt{type}\ \mathcal{C} = \overline{R}\ ;\ \Delta_2\ \}$ 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), $\Gamma; \Delta \vdash s_1 : U_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathcal{C} = \overline{S}\ ;\ \Delta_1\ \}$, and $t' = s_1'\ \texttt{compose}\ s_2$. By the induction hypothesis, $\Gamma; \Delta \vdash s_1' : U_1 \Rightarrow T_2\{\ \texttt{type}\ \mathcal{C} = \overline{S}\ ;\ \Delta_1\ \}$. By (T-Comp), we know that $\Gamma; \Delta \vdash s_2 : T_1 \Rightarrow U_1\ \{\ \texttt{type}\ \mathcal{C} = \overline{R}\ ;\ \Delta_2\ \}$ 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), $\Gamma; \Delta \vdash s_2 : T_1 \Rightarrow U_1\ \{\ \texttt{type}\ \mathcal{C} = \overline{R}\ ;\ \Delta_2\ \}$, and $t' = v_1\ \texttt{compose}\ s_2'$. By the induction hypothesis, $\Gamma; \Delta \vdash s_2' : T_1 \Rightarrow U_1\ \{\ \texttt{type}\ \mathcal{C} = \overline{R}\ ;\ \Delta_2\ \}$. Since (E-Comp2) applies, $s_1 = v_1$, so by (T-Comp), we know that $\Gamma; \Delta \vdash v_1 : U_1 \Rightarrow T_2\ \{\ \texttt{type}\ \mathcal{C} = \overline{S}\ ;\ \Delta_1\ \}$ 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 2.5. 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 2.5, $\overline{x : S}, y : U_1, x : U_1 \vdash t_2 : T_2$. By Lemma 2.4, $\overline{x : S}, x : U_1 \vdash [y \mapsto x]t_2 : T_2$. By Lemma 2.5, $\overline{x : S}, \overline{y : R}, z : T_1, x : U_1; \Delta \vdash [y \mapsto x]t_2 : T_2$.

  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 $\Gamma; \Delta \vdash t' : T$.

□

2.5 Relation to spores in Scala

The soundness proof (see Section 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.

2.6 Excluded types

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

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

Figure 6: Core language syntax extensions

Figure 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.

Figure 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 $\overline{V}$

S-ESpore

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

S-ESporeFun

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

Figure 7: Subtyping extensions

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

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

T-EComp

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

Figure 8: Typing extensions

Figure 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).

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

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

Figure 9: Operational semantics extensions