A Flow-Insensitive, Modular Effect System for Purity

Lukas Rytz Nada Amin Martin Odersky

École Polytechnique Fédérale de Lausanne, Switzerland

{first.last}@epfl.ch

ABSTRACT

This article presents a modular, flow-insensitive type-and-effect system for purity with lightweight annotations. It does not enforce a global programming discipline and allows arbitrary effects to occur in impure parts of the program. The system is designed to support higher-order languages that mix functional and imperative code like Scala or C#. We show that it can express purity of non-local programming patterns which involve mutable state such as those used in the Scala collections library. We formalize the type system using a functional language with mutable records and define type and effect soundness.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program analysis; F.3.3 [**Studies of Program Constructs**]: Type structure

General Terms

Languages, Theory

Keywords

purity, type-and-effect systems

1. INTRODUCTION

In this article we present a type-and-effect system for verifying purity in higher-order languages. Our system does not enforce any global programming discipline: it can ensure purity in certain parts of a program while allowing arbitrary aliasing and side-effects to occur elsewhere.

Such a type system has many potential applications. Besides providing valuable verified documentation, it can enforce implicit purity constraints in parallel code or enable safe parallel execution as shown in [2]. Purity information can also be exploited in various ways to improve performance. For instance in Scala, when inlining a method defined in a singleton object, the compiler still has to insert an access to the object. The reason is that the object constructor, which is executed on the first access, might have side-effects.

The type system we present builds on the ideas introduced by Pearce in JPure [15], a purity system for Java. Like in JPure effects are computed using a modular, intraprocedural analysis based on effect annotations. We use the same concept of *locality* to denote private state which is part of the representation of an object. Our contributions are as follows:

- Our effect system is flow-insensitive which makes it suitable for higher-order languages such as Scala or C#. Flow-insensitivity also enables the effect system to be integrated with existing frameworks for effect checking ([10], [17]).
- We generalize the notion of *freshness* and allow annotating the precise locality a function's return value. This enables our system to correctly express the behavior of getter methods which are ubiquitous in Scala.
- Our system allows effect annotations of nested methods to refer to parameters and variables from the enclosing scope, which is important for expressing effects in higher-order code.
- We provide a formalization of the type system with static and dynamic semantics and we discuss soundness theorems for type safety and purity.

The next section gives an overview of the effect system and shows that it can express purity of important non-local programming patterns which involve mutable state.

2. OVERVIEW

2.1 Purity and Freshness

Our type-and-effect system is based on the observation that a method which modifies only freshly allocated objects has no observable effect for its clients and therefore can be considered pure. However, a simple system which annotates methods as pure or impure is unable to express some important side-effect free programming patterns used in practice, for instance the use of an iterator:

```
def contains(i: Int, l: List): Boolean = {
  val it = l.iterator()
  while (it.hasNext())
    if (it.next() == i) return true
  false
}
```


FTfJP '13, Montpellier, France

We notice that the method `contains` does not have an observable side-effect: `l.iterator()` returns a new object, `hasNext()` is pure and `next()` only modifies fields of the fresh object `it`. These updates are not observable for a caller of `contains`.

However, method `next` does have the side-effect of modifying the iterator and therefore has to be annotated as impure. Since `contains` invokes an impure method it also needs to be be annotated as impure.

To overcome this limitation we introduce more precise effect annotations which specify the parameters a method can modify:

```
class Iterator[T] { def next(): T @mod(this) = { .. } }
```

In the body of `contains`, `it` is known to be a fresh, no-aliased object. Therefore effects on `it` can be masked and `contains` is considered pure.

2.2 Ownership and Locality

The system outlined so far fails to express the purity of another common programming pattern which uses only locally scoped effects. If an object holds internal state which is never shared with other objects or instances, modifications of that state can be encapsulated as modifications of the object itself.

Such an example is the `Builder` class used in the Scala collections library [13] which supports appending elements and retrieving the resulting collection.

```
class Builder[T, Res] {
  @local private var a: Array[T] = ..
  def append(e: T): Unit @mod(this) = { .. }
  def result: Res @mod() = { .. }
}
```

The builder implementation above uses an internal array to store the appended elements. Like in the iterator example, if a fresh builder is used within a method to add elements and eventually obtain a collection, the effects on that builder and its array are not observable from the outside.

In order to support objects with internal data structures the system is extended with a simple notion of ownership. By marking a field of an object as `@local`, the programmer defines the internal state of the object accessible through that field to be owned by the outer object. A method annotated with effect `@mod(o)` is not only allowed to modify the fields of `o`, but also the fields of objects owned by `o`.

For every object its *locality* is defined as the transitive closure of all objects reachable through fields annotated `@local`. The type system ensures that an object type checks as being fresh only if all objects in its locality are also fresh.

2.3 Tracking Localities

The effect annotation `@mod(o)` on a method ensures that *if* the value passed for `o` in a specific invocation is known to be fresh, *then* that invocation only modifies fresh state.

In our first example, purity of the method `contains` depends on the fact that the iterator `it` is known to be fresh, so modifying its fields does not modify any state that existed before. But how does the type system know that an object is fresh? The answer is a third kind of annotation, the locality annotation `@loc`, which specifies the locality of the object that a method returns.

If a method always returns a freshly allocated object whose locality cannot be accessed through any previously existing state, then that method is called *fresh* and is annotated with the empty locality `@loc()`. The most prominent examples of such methods are factory methods, but there are other important fresh methods such as `List.iterator()` which always creates a new iterator.

There are also methods which *sometimes* return a fresh object, depending on the parameters. The most common case is getters of local fields, such as `getCounter` in the following example:

```
class HasCounter {
  @local var counter = new Counter()
  def getCounter: Counter @loc(this) = counter
}
```

Remember that if an object type checks as fresh, all objects in its locality are also fresh. Therefore a getter with locality `@loc(this)` returns a fresh object *if* the receiver object is known to be fresh:

```
def test: Int @mod() = {
  val hc = new HasCounter  // hc is fresh
  val c  = hc.getCounter   // c is also fresh
  c.inc()                  // modifies only fresh state
}
```

Note that at each call site, effects on parameters are translated by the type system according to the localities of the corresponding arguments. In the following example, the effect `@mod(this)` of method `inc` in class `HasCounter` is translated to `@mod(hc)`:

```
def incHc(hc: HasCounter): Int @mod(hc) = {
  val c = hc.getCounter   // c has locality hc
  c.inc()                 // modifies the locality of c
}
```

Getters are common in many languages, but even more so in Scala where every field access is performed through an accessor method [12]. When applying the type system in practice, the ability to precisely express the return locality of methods is therefore of fundamental importance. The effect of a setter method is expressed using a `@mod` annotation:

```
def setC(hc: HasCounter, c: Counter): Unit @mod(hc, c) =
  { hc.counter = c }
```

Note that the effect annotation includes not only the modified object `hc`, but also the stored object `c`. The reason is that the counter `c` is *captured* in the locality of the other object `hc`. The effect annotation states that an invocation of the setter can be considered pure only if both involved objects are fresh. This restriction is important to maintain the freshness invariant: otherwise we could store some non-fresh counter `c` inside the locality of a fresh object `hc`.

Our final example is a factory method that accepts an initial value for the local field of the constructed object. It returns a fresh object, but the argument object is stored in the result:

```
def mkHC(c: Counter): HasCounter @mod(c) @loc() = {
  val hc = new HasCounter()
  hc.counter = c
  hc
}
```

The assignment has effect `@mod(hc, c)` and the result locality of the method body is `@loc(hc)`. Once the local variable `hc` gets out of scope, references to it are replaced by its initial locality and we obtain the annotations in the code.

The freshness annotation `@loc()` seems surprising at first:

the returned object can only be typed as fresh if the parameter `c` is also fresh, otherwise it has non-fresh state in its locality.

However, in combination with the `@mod(c)` annotation the freshness annotation is safe. Intuitively, any term which does have a side-effect can create aliases between the modified object and previously fresh state and hereby invalidate freshness. Therefore objects are only known to be fresh in pure contexts. In the example of `mkHC`, this means that the resulting object can only be considered fresh if the argument for `c` is also fresh, otherwise the effect on the argument might have invalidated freshness.

Note that annotating the method `mkHC` with the result locality `@loc(c)` is equivalent and does not introduce any imprecision. One could say that the `@loc()` annotation of a method implicitly contains all localities from the method's `@mod` effect.

2.4 Higher-order Code

Since effect annotations use variable and parameter names as abstract locations, our effect system applies naturally to nested methods and higher-order functions. For instance, a nested method can update the state of a local variable in the enclosing scope:

```
def t(): Int @mod() = {
  val c = new Counter()
  def up(): Unit @mod(c) { c.inc() }
  up()
}
```

In order to support functions that act on their environment, localities of variables are flow-insensitive. This restriction is not limiting in practice because we use effects to record if a variable gets stored in some other locality, as shown in the `mkHC` example in the previous section. The advantage of a flow-insensitive system is that in a sequence of statements and subroutine invocations, effects can simply be joined, there is no ordering between effects. A flow-sensitive system would also require more complex effect annotations that can express the sequence of effects on abstract locations.

Similar to nested methods, also higher-order functions can have effects on their environment. Together with a language that supports effect-polymorphism we obtain a powerful system that can verify purity of higher-order code which uses local state.

We implemented the purity type system in our framework for effect checking in Scala[1] which is based on the effect-polymorphic type system described in [17]. In the following example, the method `foreach` of class `List` is effect-polymorphic: this means that the effect of invoking `foreach` depends on the effect of its argument function.

```
def length(l: List): Int @mod() = {
  val c = new Counter()
  l.foreach(e => c.inc())
  c.get
}
```

The function literal `e => c.inc()` has effect `@mod(c)`, therefore the call to `foreach` also has effect `@mod(c)`. Since the object `c` is known to be freshly allocated within `length`, the modification effect can be masked and the method body is type checked as pure.

[1]Available on `https://github.com/lrytz/efftp`

t	$::=$	let $x = p$ in t	let binding
		$\mid x.l := y$	assignment
		$\mid x$	variable
p	$::=$	$(x : T) \to t$	abstraction
		$\mid x\ y$	application
		$\mid \{\overline{[\text{local}]\ l = x}\}$	record construction
		$\mid x.l$	selection
		$\mid t$	term
T	$::=$	$(x : T) \xrightarrow{e}_{loc} T$	function type
		$\mid \{\overline{[\text{local}]\ l : T}\}$	record type
loc	$::=$	$\overline{x} \mid$ any	locality annotation
e	$::=$	$\overline{x} \mid$ any	effect annotation
Γ	$::=$	$\overline{x : T \circ loc}$	variable typing
Σ	$::=$	$\overline{\mathrm{r} : T}$	store typing
v	$::=$	r	reference
		$\mid [(x : T) \to t; V]$	closure
H	$::=$	$\emptyset \mid H, r \mapsto \{\overline{[\text{local}]\ l = v}\}$	heap
V	$::=$	$\emptyset \mid V, x \mapsto v$	stack
K	$::=$	halt $\mid [x; V; K; p]$	continuation

Figure 1: Core language syntax

We have not formally studied the interaction between the purity effect system and the framework for polymorphic effect checking, but the empirical results we gained with the implementation look promising.

3. FORMALIZATION

We formalize the type-and-effect system for purity in the context of a lambda calculus with mutable records, presented in Figure 1. The language is in A-normal form [6] so that all intermediate terms are named. It does not feature arbitrary mutable references (cf. [16], Chapter 13) but limits assignments to the fields of records.

To define the locality of an object as presented in Section 2, the syntax for record literals allows fields to be optionally annotated `local`. Likewise, record types register which fields of an object are `local`.

Function types consist of a parameter name and type, a latent effect e describing the localities the function might modify, a locality annotation loc which designates the locality of the returned value and a return type.

The effect and locality annotations are either a list of variables $\overline{x}$ or the unknown locality "any". We use $\emptyset$ to denote the empty list: methods with an empty effect annotation are pure, an empty locality annotation describes methods that return fresh objects. A method with the "any" effect might modify any existing object and create arbitrary aliases in the heap. A locality annotation "any" says that the locality of the returned object is unknown.

The following example creates a counter, increments it and returns its value (it assumes integers to be part of the language):

$$
\begin{array}{lll}
\text{let } c = \{x = 1\} & \text{in} & \\
\text{let } inc = (\_ : \{\}) \to & & \\
\quad \text{let } v = c.x & \text{in} & \\
\quad c.x := v + 1 & \text{in} & \\
\text{let } \_ = inc\ \{\} & \text{in} & \\
\text{let } r = c.x & \text{in} & r
\end{array}
$$

$\boxed{T' <: T}$. Reflexivity and transitivity are omitted.

$$\frac{\overline{l} \subseteq \overline{l'} \qquad \forall i. l'_i = l_i \Rightarrow T'_i <: T_i \wedge T_i <: T'_i \wedge \text{local } l'_i = \text{local } l_i}{\{\overline{[\text{local}]\ l' : T'}\} <: \{\overline{[\text{local}]\ l : T}\}} \quad \text{(S-Rec)}$$

$$\frac{T_1 <: T'_1 \qquad [x/x']T'_2 <: T_2 \qquad [x/x']e' \sqsubseteq e \qquad [x/x']loc' \leq loc}{(x' : T'_1) \xrightarrow{e'}_{loc'} T'_2 <: (x : T_1) \xrightarrow{e}_{loc} T_2} \quad \text{(S-Fun)}$$

$\boxed{[loc/x]T}$

$$\frac{}{[loc_x/x]\{\overline{[\text{local}]\ l : T}\} = \{\overline{[\text{local}]\ l : [loc_x/x]T}\}}$$

$$\frac{T = (y : T_1) \xrightarrow{e}_{loc} T_2 \qquad y \neq x}{[loc_x/x]T = (y : [loc_x/x]T_1) \xrightarrow{[loc_x/x]e}_{[loc_x/x]loc} [loc_x/x]T_2}$$

$\boxed{loc' \leq loc}$

$$\frac{}{loc' \leq \text{any}} \qquad \frac{\overline{x'} \subseteq \overline{x}}{\overline{x'} \leq \overline{x}}$$

$\boxed{[loc_x/x]loc}$

$$\frac{loc = \text{any} \vee x \notin loc}{[loc_x/x]loc = loc}$$

$$\frac{x \in \overline{x}}{[\text{any}/x]\overline{x} = \text{any}} \qquad \frac{x \in \overline{x}}{[\overline{x'}/x]\overline{x} = (\overline{x} \setminus x), \overline{x'}}$$

$\boxed{e' \sqsubseteq e}$, $\boxed{[x/x']e}$ similar to $loc' \leq loc$, $[x/x']loc$

Figure 2: Subtyping

Function *inc* has type $(\_ : \{\}) \xrightarrow{c}_c \{x : \text{Int}\}$ with latent effect c and also return locality c (assignments evaluate to the assignee). The invocation of *inc* has therefore effect c which is masked once c gets out of scope, making the overall expression pure. The details of typing are explained in Section 3.2.

3.1 Subtyping

The subtyping rules are presented in Figure 2. For function types, the subtype needs to have a smaller effect and a more precise locality. Using a pure function in places where an effectful function is expected is safe, and similarly, a function returning fresh objects can be safely used when a function returning arbitrary objects is expected.

Effects and localities form a lattice with "any" as the top element.

Since the fields of records are mutable, record subtyping needs to be invariant (cf. [16], Chapter 15-5). Note that the types need to agree on the "local" annotations on their fields.

3.2 Typing Rules

The typing statement in Figure 3 assigns a type T, a locality loc and an effect e to an expression p. An effect $\overline{x}$ can be understood as a requirement for the expression to be pure: if all variables in $\overline{x}$ hold fresh objects, then the expression can only modify fresh state and does not have an observable effect. Similarly, a locality $\overline{x}$ says that the expression evaluates to a fresh value if all variables in $\overline{x}$ are fresh.

We now explore the typing rules using an example.

$$\begin{array}{l} newHC = (\_ : \{\}) \to \quad // \ (\_ : \{\}) \xrightarrow{\emptyset}_{\emptyset} \{\text{local } c : \{x : \text{Int}\}\} \\ \quad \text{let } k = \{x = 0\} \quad \text{in} \quad \text{let } r = \{\text{local } c = k\} \quad \text{in} \quad r \end{array}$$

Function *newHC* returns a fresh object containing a counter. For the literal $\{\text{local } c = k\}$, rule T-Rec assigns locality k. T-Let of let binding k substitutes $[\emptyset/k]$ in that locality, therefore the body of *newHC* type checks with locality $\emptyset$.

For brevity we introduce two type aliases in the next example:

$$\begin{array}{l} \text{type } K = \{x : \text{Int}\} \qquad \text{type } H = \{\text{local } c : K\} \\ setC = (hc : H) \to \text{let } g = (k : K) \to hc.c := k \text{ in } g \end{array}$$

The setter *setC* has type $(hc : H) \xrightarrow{\emptyset}_{\text{any}} (k : K) \xrightarrow{hc,k}_{hc} H$. We analyze a program which creates a counter, changes its value and finally resets it:

$$\begin{array}{ll} \text{let } h = newHC\ \{\} & \text{in} \\ \text{let } z = h.c \quad \text{in} \quad \text{let } \_ = z.x := 2 & \text{in} \\ \text{let } s = setC\ h \quad \text{in} \quad \text{let } \_ = s\ \{x = 0\} & \text{in} \quad h \end{array}$$

The effect of the assignment $z.x := 2$ is z. In the last line, when applying the curried function *setC* to a first argument, the parameter symbol is substituted as $[h/hc]$ in the result type. Therefore s has type $(k : K) \xrightarrow{h,k}_h H$. Applying s to a fresh record now has effect h, so the body of the let binding for z has a total effect z, h. The T-Let rule will first substitute $[\emptyset/z](z, h)$ for the let binding of z, and finally $[\emptyset/h]h$ for the let binding of h. Therefore the example type checks as pure.

Care must be taken for functions that capture local variables from their environment:

$$\text{let } f = \text{let } c = \{x = 0\} \quad \text{in} \quad \text{let } g = (\_ : \{\}) \to c \quad \text{in} \quad g$$

The function f returns a reference to the same object on every invocation, so it cannot be considered fresh. This problem is addressed by the meta-function "elim" in typing rule T-Let which substitutes references to the local variable in effect and localities by "any" ($\emptyset$ in contravariant positions). So f has type $(\_ : \{\}) \xrightarrow{\emptyset}_{\text{any}} \{x : \text{Int}\}$, it has no effect and returns an object of unknown locality.

3.3 Dynamic Semantics

We define the semantics of our language using a CESK machine ([5], [11]) in Figure 4. A machine state $\langle H; V; K; p \rangle$ consists of a heap H, a stack V, a continuation K and an expression p. The relation $\longrightarrow$ defines a small-step operational semantics as transitions between machine states.

As described in Figure 1, a stack maps variables to values where values are either heap references or closures. A heap maps references to record literals where each label in the record points to a value.

The function $\mathcal{V}$ maps expressions to variables using environment V, and records changes to the heap H.

Each machine state holds a stack of continuations rooted in the "halt" continuation which terminates execution. A non-terminating continuation consist of a variable x, a stack V_K and an expression p_K. Continuations are invoked in rule E-Expr: once the current term p evaluates to a value v, variable x is bound to v and execution resumes using V_K and p_K. Evaluating a "let" term creates a new continuation for the let body and continues by evaluating the variable initializer.

The advantage of basing the semantics on environment passing and continuations instead of substitution is that it simplifies reasoning about localities. In a traditional semantics a record literal would evaluate to a reference, which might get replicated when substituting a function parameter. This

$\boxed{\Gamma \vdash p : T \circ loc \,!\, e}$

$$\frac{x : T \circ loc \in \Gamma}{\Gamma \vdash x : T \circ loc \,!\, \emptyset}\ \text{(T-Param)} \qquad \frac{\Gamma \vdash p : T_1 \circ loc_1 \,!\, e_1 \qquad \Gamma, x : T_1 \circ x \vdash t : T_2 \circ loc_2 \,!\, e_2}{\Gamma \vdash \text{let } x = p \text{ in } t : \text{elim}(x, T_2) \circ [loc_1/x]loc_2 \,!\, e_1 \cup [loc_1/x]e_2}\ \text{(T-Let)}$$

$$\frac{\Gamma, x : T_1 \circ x \vdash t : T \circ loc \,!\, e}{\Gamma \vdash (x : T_1) \to t : (x : T_1) \xrightarrow{e}_{loc} T \circ \text{any} \,!\, \emptyset}\ \text{(T-Abs)}$$

$$\frac{\Gamma \vdash p : T' \circ loc' \,!\, e' \quad T' <: T \quad loc' \le loc \quad e' \sqsubseteq e}{\Gamma \vdash p : T \circ loc \,!\, e}\ \text{(T-Sub)} \qquad \frac{\overline{\Gamma \vdash x : T \circ loc \,!\, \emptyset} \qquad loc_r = \bigcup_i \begin{cases} loc_i & \text{if local } l_i \\ \emptyset & \text{otherwise} \end{cases}}{\Gamma \vdash \{\overline{[\text{local}]\ l = x}\} : \{\overline{[\text{local}]\ l : T}\} \circ loc_r \,!\, \emptyset}\ \text{(T-Rec)}$$

$$\frac{\Gamma \vdash f : (x : T_1) \xrightarrow{e}_{loc} T_2 \circ \text{any} \,!\, \emptyset \quad \Gamma \vdash a : T_1 \circ loc_a \,!\, \emptyset}{\Gamma \vdash f\ a : [loc_a/x]T_2 \circ [loc_a/x]loc \,!\, [loc_a/x]e}\ \text{(T-App)}$$

$$\frac{\Gamma \vdash x : \{\overline{[\text{local}]\ l : T}\} \circ loc_x \,!\, \emptyset \qquad loc = \begin{cases} loc_x & \text{if local } l_i \\ \text{any} & \text{otherwise} \end{cases}}{\Gamma \vdash x.l_i : T_i \circ loc \,!\, \emptyset}\ \text{(T-Select)}$$

$$\frac{\Gamma \vdash x : \{\overline{[\text{local}]\ l : T}\} \circ loc_x \,!\, \emptyset \quad \Gamma \vdash y : T_i \circ loc_y \,!\, \emptyset \qquad e = \begin{cases} loc_x \cup loc_y & \text{if local } l_i \\ loc_x & \text{otherwise} \end{cases}}{\Gamma \vdash x.l_i \coloneqq y : \{\overline{[\text{local}]\ l : T}\} \circ loc_x \,!\, e}\ \text{(T-Assign)}$$

$\boxed{\text{elim}(x, T)} = \text{elim(x, T, any)}$

$$\frac{T = \{\overline{[\text{local}]\ l : T'}\}}{\text{elim}(x, T, loc) = \{\overline{[\text{local}]\ l : \text{elim}(x, T', loc)}\}} \qquad \frac{T = (y : T_1) \xrightarrow{e}_{loc} T_2 \qquad y \neq x \qquad loc'_x = \text{if } (loc_x = \text{any})\ \emptyset, \text{ else any}}{\text{elim}(x, T, loc_x) = (y : \text{elim}(x, T_1, loc'_x)) \xrightarrow{[loc_x/x]e}_{[loc_x/x]loc} \text{elim}(x, T_2, loc_x)}$$

Figure 3: Typing rules

$\boxed{\mathcal{V}(p, V, H) = v; H'}$

$$\frac{}{\mathcal{V}(y, V, H) = V(y); H} \qquad \frac{}{\mathcal{V}(\{\overline{[\text{local}]\ l = y}\}, V, H) = r; H, r \mapsto \{\overline{[\text{local}]\ l = V(y)}\}}$$

$$\frac{}{\mathcal{V}((x : T) \to t, V, H) = [(x : T) \to t; V]; H} \qquad \frac{V(y) = r \qquad H(r) = \{\overline{[\text{local}]\ l = v}\}}{\mathcal{V}(y.l_i, V, H) = v_i; H}$$

$\boxed{\langle H; V; K; p\rangle \longrightarrow \langle H'; V'; K'; p'\rangle}$

$$\frac{\mathcal{V}(p, V, H) = v; H'}{\langle H; V; [x; V_K; K; p_K]; p\rangle \longrightarrow \langle H'; V_K, x \mapsto v; K; p_K\rangle}\ \text{(E-Expr)}$$

$$\frac{V(f) = [(y : T) \to t_1; V_f] \qquad V(a) = v_a}{\langle H; V; K; f\ a\rangle \longrightarrow \langle H; V_f, y \mapsto v_a; K; t_1\rangle}\ \text{(E-App)}$$

$$\frac{}{\langle H; V; K; \text{let } x = p \text{ in } t\rangle \longrightarrow \langle H; V; [x; V; K; t]; p\rangle}\ \text{(E-Let)}$$

$$\frac{V(x) = r \qquad H(r) = \{\overline{[\text{local}]\ l = v}\}}{\langle H; V; K; x.l_i \coloneqq y\rangle \longrightarrow \left\langle [r \mapsto [V(y)/v_i]\{\overline{[\text{local}]\ l = v}\}]H; V; K; x\right\rangle}\ \text{(E-Assign)}$$

Figure 4: Dynamic Semantics

renders the preservation proof difficult because the typing rules would need to keep track of the localities of references.

4. TOWARDS SOUNDNESS

We define soundness for our type-and-effect system using a standard type safety theorem and an additional purity theorem. In short, type safety states that a well-formed machine state is either final, or it transitions into another well-formed state of the same type. The purity theorem ensures that if a term type checks as pure, then its evaluation cannot modify any existing state. Due to limited space we omit most formal definitions.

The type safety theorem ensures that a well-formed machine state steps to a well-formed machine state. A machine is well-formed if it satisfies the judgment $\Gamma; \Sigma \vdash \langle H; V; K; p\rangle : T \circ loc \,!\, e$. This judgment ensures a valid typing of expression p and various well-formedness conditions for the configuration, including the freshness invariant *freshSeparate* which will be discussed below.

The typing for expression p has the form $\Gamma \vdash p : T_p \circ loc_p \,!\, e_p$. However, p is only an intermediate expression of the entire program, the rest of the program is represented by the continuation K. Since the program has type T, the continuation is required to map a value of type T_p to the final type T, which is ensured by well-formedness of the continuation stack: $\Sigma \vdash K : T_p \circ loc_p \,!\, e_p \Rightarrow T \circ loc \,!\, e$.

Theorem 1. *Type Safety: if* $\Gamma; \Sigma \vdash \langle H; V; K; p\rangle : T \circ loc \,!\, e$ *then either the state is final, or* $\langle H; V; K; p\rangle \longrightarrow \langle H'; V'; K'; p'\rangle$ *and* $\Gamma'; \Sigma' \vdash \langle H'; V'; K'; p'\rangle : T \circ loc \,!\, e$ *for some* Γ' *and* $\Sigma' \supseteq \Sigma$.

Note that the environment Γ' which describes the resulting stack V' is unconstrained: in the case of E-App and E-Expr, V' origins in the invoked closure or continuation and is therefore not directly related to V.

To motivate the purity theorem we look at an example:

$$\text{let } c = \{x = 1\} \quad \text{in} \quad c.x \coloneqq 2$$

This program transitions through the following states:

$$\begin{array}{l} \langle \emptyset; \emptyset; \text{halt}; \text{let } c = \ldots\rangle \longrightarrow \\ \langle \emptyset; \emptyset; [c; \emptyset; \text{halt}; c.x \coloneqq 2]; \{x = 1\}\rangle \longrightarrow \\ \langle r_c \mapsto \{x = 1\}; c \mapsto r_c; \text{halt}; c.x \coloneqq 2\rangle \longrightarrow \\ \langle r_c \mapsto \{x = 2\}; c \mapsto r_c; \text{halt}; c\rangle \end{array}$$

Even though the initial program type checks as pure, the last transition clearly modifies the heap. However, type safety requires that each of the machine states type checks with the initial type, locality and effect. Therefore we need a way to type check the expression $c.x := 2$ as pure, which is only possible if the variable x is fresh in the typing environment, i.e. $(x : \{c : \text{Int}\} \circ \emptyset) \in \Gamma$.

This raises the question of how purity is defined in the presence of expressions which *do* modify the heap. The idea is that a pure expression is allowed to modify the localities of variables which are fresh in the typing environment Γ. In order to delimit the scope of these effects and to ensure that only fresh state can be modified, we define the freshness invariant *freshSeparate* which every well-formed machine state has to satisfy.

Definition 1. $\mathit{freshSeparate}(\Gamma, H, V)$ holds if the localities of fresh variables in Γ are not reachable through other variables in the environment. The formal definition is omitted.

Definition 2. The *locality* of a reference r in heap H includes r and the localities of all *local* fields in the record $H(r)$.

$$\frac{H(r) = \{\overline{[\text{local}]\ l = v}\} \qquad l_c = \bigcup_{\{i \mid \text{local } l_i = r_i\}} \mathit{locality}(r_i, H)}{\mathit{locality}(r, H) = r \cup l_c}$$

In our example, the only fresh variable is c with locality r_c. Since there are no paths to r_c starting at non-fresh variables, the separation invariant holds.

The purity theorem states that if the expression of a well-formed state type checks as pure, then a transition can only modify localities of fresh variables.

THEOREM 2. *Purity: if* $\Gamma; \Sigma \vdash \langle H; V; K; p\rangle : T \circ \mathit{loc}\ !\ e$ *and* $\Gamma \vdash p : T_p \circ \mathit{loc}_p\ !\ \emptyset$ *and* $\langle H; V; K; p\rangle \longrightarrow \langle H'; V'; K'; p'\rangle$, *then all references* r *such that* $H'(r) \neq H(r)$ *belong to the locality of some fresh variable in* Γ.

Note that the pure term p appears within a larger, possibly impure program: the continuation K has an unknown effect e. This shows that the effect system can express purity of parts of a program while allowing arbitrary effects to occur elsewhere.

In order to show that a subterm p does not modify any existing state, we need to show that it can be typed in an environment a Γ with *no* fresh variables. If we then construct a machine state $\langle H; V; \text{halt}; p\rangle$ by just inserting the "halt" continuation, this state trivially type checks as pure. Therefore, by type safety, also its successor states have to be pure and the evaluation of p can not modify any existing state.

There is however one major issue which we have not resolved yet: we do not have a way to ensure consistency of typing environments across context switches. As explained before, the resulting Γ' in the type safety theorem is unconstrained. Assume an initial state s_0 types as pure using a Γ_0 in which x is not fresh. After a number of transitions, by type safety we know that the state s_i also types as pure with respect to some environment Γ_i. Since there are no constraints on the environments, x might be fresh in Γ_i. In this case the next transition would be allowed to modify the locality of x, which contradicts our intention given the initial typing.

We are still investigating how the necessary consistency requirements can be integrated in the well-formedness properties, in order to relate typing environments across context switches.

5. RELATED WORK

The type system presented in this article is strongly influenced by JPure, a purity system for Java by Pearce [15]. Our notions of *locality* and *freshness* are equivalent to theirs.

In contrast to our system, JPure tracks the freshness of local variables in a flow-sensitive manner. While being more precise in some situation, a flow-sensitive system is difficult to integrate into a language with nested methods and higher-order code. Our observations in Scala let us believe that the additional precision is not essential in practice.

Another difference which is essential for integration in higher-order languages is that our system allows effect and locality annotations to refer to parameters and variables from the enclosing scope. Effect annotations in JPure can only refer to the method's parameters.

A method annotated `@Fresh` in JPure will always return a freshly allocated object, while all other methods return objects with an unknown locality (annotated `@loc(any)` in our system). Our type system allows defining the locality of the returned object more precisely, which enables us for instance to correctly express the behavior of getters for local fields. If an object is known to be fresh, then the values stored in its local fields are guaranteed to be fresh as well. Therefore the getter of a local field returns a fresh object *if* the owner object is fresh, which is expressed as `@loc(this)` in our system. In JPure the getter is annotated non-fresh, therefore modifying a local field of a fresh object has the unknown effect if the field is accessed through its getter.

In addition to defining the purity type system, Pearce implemented an inference system and shows purity of a significant portion of the Java library.

Ownership and universe type systems ([3], [4]) delimit the scope of side effects by controlling aliasing. They enforce disjointness of effects using a global programming discipline: state modifications are restricted to certain access patterns, programs which violate these patterns cannot be expressed. Our type system on the other hand can only identify purity of certain functions, but it allows programs with arbitrary effects by type checking them with the unknown effect.

Sălcianu and Rinard use a pointer and escape analysis to track side-effects [18]. The system performs a whole-program analysis, like most systems based on points-to information. The focus of our work is on modularity and providing lightweight effect annotations.

Huang el al. present a purity type system based on reference immutability [7]. Their inference system scales to large programs and is able to discover a significant amount of pure methods in existing Java libraries. Integrating their type system into a higher-order language would require further research and is not immediately obvious.

State monads [8] can encapsulate local state within the implementation of an externally pure algorithm, effects are represented as value types. The Koka language [9] supports inference for usages of local state and masks them by automatically inserting `runST` commands.

DPJ [2] can ensure non-interference of parallel tasks using an effect system built on the tradition of region-based effect analysis. One of the goals of our work is to provide an effect

system with more lightweight effect annotations than typical region-based systems.

There is a large body of related work in the area of program verification. For instance, regional logic [1] uses a similar notion of freshness and expresses framing conditions as effects. Parkinson and Bierman apply separation logic to languages with inheritance [14].

6. CONCLUSION

We presented an effect system for purity based on the common definition that a method is pure if it only modifies freshly allocated state. For example, finding an element in a list using an iterator is a pure procedure, even though the iterator modifies some state when it advances. Our system is flow-insensitive and modular, yet precise enough to capture common patterns of programming in Scala, which mixes higher-order functional and imperative code. We achieve precision through lightweight annotations of effects on function types and of ownership on fields.

In the future we will continue to work on the soundness proofs for the type system. We will also continue to study the interaction between the purity type system and our framework for polymorphic effect checking [17], and we will evaluate the practicality and scalability of our approach by applying the implementation to larger bodies of code such as the Scala standard library.

7. ACKNOWLEDGMENTS

We would like to thank Philipp Haller for his helpful comments and for taking the time to review drafts of this paper.

8. REFERENCES

[1] A. Banerjee, D. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In J. Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 387–411. Springer Berlin Heidelberg, 2008.

[2] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. ACM.

[3] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 48–64, New York, NY, USA, 1998. ACM.

[4] W. Dietl, S. Drossopoulou, and P. Müller. Generic universe types. In E. Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 28–53. Springer Berlin Heidelberg, 2007.

[5] M. Felleisen. *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.

[6] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. *SIGPLAN Not.*, 28(6):237–247, June 1993.

[7] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. ReIm & ReImInfer: checking and inference of reference immutability and method purity. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 879–896, New York, NY, USA, 2012. ACM.

[8] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 24–35, New York, NY, USA, 1994. ACM.

[9] D. Leijen. Koka: A language with effect inference. http://research.microsoft.com/en-us/projects/koka/2012-overviewkoka.pdf, April 2012.

[10] D. Marino and T. Millstein. A generic type-and-effect system. In *Proceedings of the 4th international workshop on Types in language design and implementation*, TLDI '09, pages 39–50, New York, NY, USA, 2009. ACM.

[11] M. Might. Writing an interpreter, CESK-style. http://matt.might.net/articles/cesk-machines/.

[12] M. Odersky. The Scala language specification. http://www.scala-lang.org/archives/downloads/distrib/files/nightly/pdfs/ScalaReference.pdf, 2013.

[13] M. Odersky and A. Moors. Fighting bit rot with types (experience report: Scala collections). In R. Kannan and K. N. Kumar, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2009)*, volume 4 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 427–451, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[14] M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 75–86, New York, NY, USA, 2008. ACM.

[15] D. Pearce. JPure: A modular purity system for Java. In J. Knoop, editor, *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 104–123. Springer Berlin Heidelberg, 2011.

[16] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[17] L. Rytz, M. Odersky, and P. Haller. Lightweight polymorphic effects. In J. Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 258–282. Springer Berlin Heidelberg, 2012.

[18] A. Sălcianu and M. Rinard. Purity and side effect analysis for Java programs. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 199–215. Springer Berlin Heidelberg, 2005.