Gentrification Gone too Far? Affordable 2nd-Class Values for Fun and (Co-)Effect

Leo Osvald Grégory Essertel Xilun Wu Lilliam I. González-Alayón Tiark Rompf

Purdue University, USA: {losvald,gesserte,wu636,gonza304,tiark}@purdue.edu

Abstract

First-class functions dramatically increase expressiveness, at the expense of static guarantees. In ALGOL or PASCAL, functions could be passed as arguments but never escape their defining scope. Therefore, function arguments could serve as temporary access tokens or capabilities, enabling callees to perform some action, but only for the duration of the call. In modern languages, such programming patterns are no longer available.

The central thrust of this paper is to re-introduce second-class functions and other values alongside first-class entities in modern languages. We formalize second-class values with stack-bounded lifetimes as an extension to simply-typed λ calculus, and for richer type systems such as $F_{<:}$ and systems with path-dependent types. We generalize the binary first- vs second-class distinction to arbitrary privilege lattices, with the underlying type lattice as a special case. In this setting, abstract types naturally enable privilege parametricity. We prove type soundness and lifetime properties in Coq.

We implement our system as an extension of Scala, and present several case studies. First, we modify the Scala Collections library and add privilege annotations to all higher-order functions. Privilege parametricity is key to retain the high degree of code-reuse between sequential and parallel as well as lazy and eager collections. Second, we use scoped capabilities to introduce a model of checked exceptions in the Scala library, with only few changes to the code. Third, we employ second-class capabilities for memory safety in a region-based off-heap memory library.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords first-class, second-class, types, effects, capabilities, object lifetimes

1. Introduction

Modern programming languages offer much greater expressiveness than their ancestors from the 1960s and '70s. Many of the advancements that directly translate to programmer productivity are the result of removing restrictions on how certain entities can be used, and granting "first-class" status to more and more language constructs.

Strachey [57] gave a taxonomy of first- and second-class objects in 1967:

> **First and second class objects.** In ALGOL a real number may appear in an expression or be assigned to a variable, and either may appear as an actual parameter in a procedure call. A procedure, on the other hand, may only appear in another procedure call either as the operator (the most common case) or as one of the actual parameters. There are no other expressions involving procedures or whose results are procedures. Thus in a sense procedures in ALGOL are second class citizens—they always have to appear in person and can never be represented by a variable or expression (except in the case of a formal parameter).

Most modern languages have abolished these restrictions and admit functions (or objects with methods) as first-class citizens alongside integers and real numbers. Even conservative languages, like Java and C++, have added closures, albeit with some limitations. But uniformly replacing second-class with first-class constructs[1] is a process not unlike gentrification in urban development, where inexpensive living space is transformed into posh condos in an effort of modernization, but ultimately leading to an undesirable situation where inexpensive and restricted "second-class" constructs are no longer available.

Why would programmers care? Second-class values in the sense of ALGOL have the benefit of following a strict stack discipline ("downward funargs"), i.e., they cannot escape their defining scope. This makes them cheaper to implement, but more importantly, phasing out second-class en-

[1] Technically, many languages still distinguish between, e.g., normal functions and closures, but most allow converting second-class to first-class values via eta-expansion, which effectively removes the distinction.

tities has eliminated some useful programming patterns and static guarantees. Since first-class objects *may* escape their defining scope, they cannot be used to represent static capabilities or access tokens – a task that second-class values are ideally suited to because they have bounded lifetimes and they have to "show up in person".

The central thrust of this paper is to re-introduce second-class values alongside first-class entities in modern languages, and to demonstrate that this combination leads to novel and elegant implementation techniques for desirable static guarantees. The two key ideas for this combination are as follows:

1. First-class functions may not refer to second-class values through free variables
2. All functions must return first-class values, and only first-class values may be stored in object fields or mutable variables

Together, these rules ensure that second-class values do not escape their defining scope.

In contrast to systems that expose a distinct category of second-class functions, reference cells, or other entities, our system supports objects of any type as second-class values. This allows library developers to build more specific systems on top of it.

The restrictions imposed on second-class values in our system are similar to those on *borrowed references* [19, 33] in ownership type systems, e.g., as implemented in Rust [29]. However, two things set our work apart. First, we illustrate that such restrictions have important benefits as a *programming model*, orthogonal to the goals of ownership types (controlling aliasing, ensuring uniqueness, preventing race conditions, etc). Second, our system is straightforward to formalize and integrate with existing languages and other advanced type system features, something that is not true for sophisticated ownership type systems.

We make the following individual contributions:

- We present a simple type system extension for second-class values with stack-bounded lifetimes, as an extension of simply-typed λ-calculus. We prove type soundness and lifetime properties in Coq (Section 3).
- We generalize our model to polymorphic type systems such as $F_{<:}$ and calculi with path-dependent types of the DOT family. We further generalize the binary first- vs second-class distinction to an arbitrary privilege lattice, with the underlying type lattice as a special case. In this setting, abstract types naturally enable *privilege parametricity* (Section 4).
- We implement our system as an extension of the Scala language. Based on this implementation we present several case studies (Section 5).
- We modify the Scala Collections library to annotate all higher-order functions with privilege annotations. Privilege parametricity is key to retain the high degree of code-reuse between sequential and parallel as well as lazy and eager collections (Section 6).
- We introduce a model of checked exceptions in the Scala standard library, using scoped capabilities. This is significant, because Scala had not adopted a checked exception model in the past, mainly due to the difficulty of supporting effect polymorphism in the presence of generics, as, for example, in Java (Section 7).
- We employ second-class capabilities for memory safety in a region-based off-heap memory library (Section 8).

We discuss related work in Section 10 and conclude in Section 11. Our Coq proofs and Scala implementation are available from:

```
github.com/tiarkrompf/scala-escape
github.com/losvald/scala/tree/esc
```

2. Motivating Examples

To demonstrate the versatility and usefulness of our programming model, we discuss a series of motivating examples. These are presented in Scala but would directly map to other modern call-by-value languages.

Scoped Capabilities Many entities come with a lifecycle protocol that guards access. For example, when accessing a file or network connection, a program needs to *open* it, and *close* it when it is done. Accessing a file after closing it or forgetting to close a file is an error. A common and extremely useful pattern is to associate the *dynamic lifetime* of the access window with a *lexical scope*. In `C++` this can be realized with constructors and destructors for stack-allocated objects, Python has `with`, Go has `defer`, and in Scala we can define a higher-order function `withFile` that takes care of opening and closing the file, delegating to a handler `fn` for the actual processing:

```
def withFile[U](n: String)(fn: File => U): U = {
  val f = new File(n); try fn(f) finally f.close()
}
```

Client code can use `withFile` as follows:

```
withFile("out.txt") { file => file.print("Hello, World!") }
```

Thus, `file` can be seen as a *capability*: to write data to disk, we need to be given access to a `File` object via `withFile`, and when `withFile` exits, this capability is revoked.

Unfortunately, in Scala, or any other language where `file` is a first-class value, this programming pattern is merely a convention, but nothing *actually* prevents `file` from being accessed outside its lifetime window. This can lead to subtle errors, undesirable exceptions, or potential security vulnerabilities. Here are two easy ways to thwart the pattern, by assigning the file to a mutable variable or by returning it as result from the `withFile` block:

```
var f1: File = null;  withFile(n) { f => f1 = f }
val f1: File =        withFile(n) { f => f }
```

The file may also escape indirectly, through a closure:

```
val print: Str=>Unit = withFile(n) { f => (s => f.print(s)) }
```

In addition, a programmer might call other functions from within `withFile` which are unaware of the protocol, and might attempt to store the `File` for later use.

Our solution is a type system extension that lets us define `file` as a *second-class* value, and that ensures that such second-class values will not escape their defining scope. We introduce an annotation `@local` to mark second-class values, and change the signature of `withFile` as follows:

```
def withFile[U](n: String)(@local fn: (@local File) => U): U
```

Now whatever handler is passed as callback `fn` has to be a function that expects a second-class, non-escaping, argument. Note that the callback function `fn` itself is also required to be second-class, so that it can close over other second-class values. This enables, for example, nesting calls to `withFile`.

Since function types like `(@local File) => U` are so common, we provide a shorter notation: `File -> U`:

```
def withFile[U](n: String)(@local fn: File -> U): U
```

Second-class values cannot be stored in mutable variables, they cannot be returned from functions, and they cannot be accessed by first-class (named or anonymous) functions through free variables. Therefore, our earlier problem cases, instead of failing at runtime, now produce compile-time errors:

```
var f1: File = null;  withFile(n) { f => f1 = f } // error
val f1: File =        withFile(n) { f => f }      // error
val print    =        withFile(n) { f =>
                        (s => f.print(s)) }       // error
```

Second-Class Composes Can we still do anything useful with second-class values? Yes, we can pass them to other functions or methods that expect second-class arguments. For example:

```
val data = new Data { def dump(@local f: File): Unit = ... }
withFile("out.txt") { f => data.dump(f) }
```

Inside the `dump` method, the same second-class restrictions apply to the argument `f` as directly in a `withFile` block: `f` cannot be stored, captured, returned, or otherwise escape its scope.

In addition, functions with second-class *arguments* remain first-class values. This means that we can freely use patterns such as decorators, currying, or η-expansion, on them, as long as we do not capture any second-class arguments. For example, we can capture `data.dump` in a closure, and wrap it in some code that prints additional text:

```
def prettify(wrapped: File -> Unit): (File -> Unit) = { f =>
  f.print("BEGIN [");  wrapped(f);  f.print("] END")
}
val pretty = prettify(data.dump)
```

Note that variable `f` will not be allowed to escape. The result of this transformation, `pretty`, is again a first-class function that expects a second-class `File` argument. We can safely store it wherever we like and use it at our convenience:

```
withFile("out") { f => pretty(f) }
```

Thus, by cleverly combining first- and second-class values, we obtain safety without giving up expressiveness.

Higher-Order Functions and Second-Class Closures We have seen above how second-class values cannot be captured by first-class closures. Does this rule out the following code, where a closure closing over `file` is passed to `map`?

```
withFile("out.txt") { file =>
  List("Hell", "o, ", "World!") map { x => file.print(x) }
}
```

Not necessarily. We can define `map` in class `List[T]` to take a second-class closure argument as follows:

```
class List[T] {
  def map[U](@local fn: T => U): List[U] =
    if (isEmpty) Nil else fn(head) :: tail.map(fn)
  ...
}
```

The key observation here is that `map` itself treats `fn` in a strictly second-class way. The above snippet type-checks because the closure closing over `file` type-checks as a second-class value, and second-class functions are allowed to refer to other second-class values through their free variables.

One might wonder: would the same work with a *lazy* collection such as `Stream` or `Iterator`?

Suppose we would like to print in a fashion that allows for truncation of long lines and counting printed characters. For that purpose, we define a function that returns an iterator whose `next()` method prints a chunk and return its length:

```
def printingIter(ss: String*)(@local f: File): Iterator[Int] =
  ss.iterator.map(s => { f.print(s); s.length })
```

It seems as though the following code might leak a file:

```
val chunkPrinter = withFile("out.txt") { file =>
  printingIter("Hell", "o, ", "World!")(file)
}
chunkPrinter.next()  // prints to a file (?)
```

Fortunately, this is impossible. Closing over a `File` argument in `printingIter` would require `Iterator`'s `map` parameter to be second-class, i.e.:

```
class Iterator[A] { self =>  // self is alias for this
  def next(): A = ...
  def map[B](@local fn: A => B) = new Iterator[B] {
    def next(): B = fn(self.next()) // error: 1st-class next()
    ...                             //  refers to 2nd-class fn
  }
}
```

Consequently, the `next` method which accesses the mapping function `fn` and in fact the whole `Iterator` object that is returned from `map` would also need to be second-class, which our type system disallows.

We discuss our modifications to the Scala Collections library to deal with second-class values in detail in Section 6.

Implicit Capabilities as (Co-)Effects In the code above, we have already regarded `File` objects as capabilities, guarding access to their associated functionality, including `print`. We can extend this model to other kinds of capabilities. Opening a file and creating a `File` object should perhaps be guarded by a general `CanIO` capability. Likewise, a second-class `throw` function or a `CanThrow` object can embody the capability to throw an exception:

```
def withFile[U](...)(implicit @local c: CanIO): U
def throw(e: Exception)(implicit @local c: CanThrow)
```

Using Scala's `implicit` parameters, such capabilities need not be passed explicitly. For a call like `throw(e)` to type-check, it suffices to have a `CanThrow` capability in scope.

More generally, second-class values as capabilities enable a radical new take on static effect checking: instead of making effects explicit in the *type* of an expression, the capabilities available in scope characterize the effects an operation can have. Thus, it is instructive to compare this approach with other methods of statically checking side effect behavior, such as monads or traditional type-effect systems [16].

Monads and effect systems encode computational properties in the type of an expression, on the right of the turnstile;

$$G \vdash e : \texttt{CanIO}[T] \qquad \texttt{(monad)}$$
$$G \vdash e : T\,@\texttt{canIO} \qquad \texttt{(effect type)},$$

whereas our `@local` annotations are *co-effects* [40, 41], encoded on the left of the turnstile:

$$G, (@\texttt{local}\ c : \text{CanIO}), G' \vdash e : T.$$

This is a subtle but important detail. The major benefits are that the type of an expression remains standard and that it allows for easier encoding of fine-grained information. In particular, different capabilities, such as multiple open files, can be present in the environment without interference, and without picking an ordering:

```
def copyFile(@local src: File, @local dst: File): Unit = {
  dst.print(src.readAll())
}
```

In further comparison, monads offer additional power by abstracting over sequential composition through the *bind* operator. It is well known that monads essentially correspond to delimited continuations, and therefore easily encode patterns like non-determinism, probabilistic evaluation, and so on. Our second-class values, by contrast, use the normal control flow of the existing language. Thus, continuations need to be provided as an additional language feature to achieve comparable functionality. Monads further encapsulate computation as first-class values. A similar effect can be achieved with second-class capabilities, by η-expanding expressions that require capabilities in the environment. A function `(@local CanIO) => T` can be seen as roughly equivalent to the monadic `CanIO[T]`.

Effect Polymorphism Second-class capabilities also provide an elegant solution to the *effect polymorphism* problem for higher-order functions such as `map`. By taking a second-class function argument, the given definition of `map` in `List[T]` is oblivious to what effect capabilities an actual argument closure uses. The effect (as in: required capabilities) of an expression `map(f => ...)` is exactly the effect of the function `(f => ...)`. By contrast, type-and-effect systems, such as Java's checked exceptions or monads in Haskell, require two implementations of `map`, one for pure and one for impure/monadic function arguments.

Syntax

n	$::=$	$1 \mid 2$	1st/2nd class
t	$::=$	$c \mid x^n \mid \lambda x^n.t \mid t\ t$	Terms
v	$::=$	$c \mid \langle H, \lambda x^n.t \rangle$	Values
T	$::=$	$B \mid T_1^n \to T_2$	Types
G	$::=$	$\emptyset \mid G, x^n : T$	Type Envs
H	$::=$	$\emptyset \mid H, x^n : v$	Value Envs

$$G/H^{[\leq n]} = \{x^m : _ \in G/H \mid m \leq n\}$$

Operational Semantics $\boxed{H \vdash t \Downarrow^n v}$

$$H \vdash c \Downarrow^n c \qquad \text{(ECST)}$$

$$\frac{x^m : v \in H^{[\leq n]}}{H \vdash x \Downarrow^n v} \qquad \text{(EVAR)}$$

$$H \vdash \lambda x^m.t \Downarrow^n \left\langle H^{[\leq n]}, \lambda x^m.t \right\rangle \qquad \text{(EABS)}$$

$$\frac{H \vdash t_1 \Downarrow^2 \langle H', \lambda x^m.t_3 \rangle \qquad H \vdash t_2 \Downarrow^m v_2 \qquad H', x^m : v_2 \vdash t_3 \Downarrow^1 v_3}{H \vdash t_1\ t_2 \Downarrow^n v_3} \qquad \text{(EAPP)}$$

Type System $\boxed{G \vdash t :^n T}$

$$G \vdash c :^n B \qquad \text{(TCST)}$$

$$\frac{x^m : T \in G^{[\leq n]}}{G \vdash x :^n T} \qquad \text{(TVAR)}$$

$$\frac{G^{[\leq n]}, x^m : T_1 \vdash t :^1 T_2}{G \vdash \lambda x^m.t :^n T_1^m \to T_2} \qquad \text{(TABS)}$$

$$\frac{G \vdash t_1 :^2 T_1^m \to T_2 \qquad G \vdash t_2 :^m T_1}{G \vdash t_1\ t_2 :^n T_2} \qquad \text{(TAPP)}$$

Figure 1. $\lambda^{1/2}$: Syntax, Operational Semantics, and Type System

That it could be possible to build general-purpose effect systems based on implicit capabilites has been suggested previously by Odersky [36]. We present the first instantiation of such a system, as a case-study on effect-tracking for checked exceptions in Section 7.

3. Formal Development

We develop our theoretical foundation as an operational semantics for a λ-calculus with first- and second-class bindings and evaluation, along with a sound type system that enforces stack-based lifetimes for second-class bindings.

3.1 Dynamic Semantics

We formalize our model as an extended λ-calculus $\lambda^{1/2}$, where first-class and second-class identifiers use different binding forms x^1 and x^2. These correspond to names without and with `@local` annotations from Section 2. The syntax, operational semantics, and type system for this $\lambda^{1/2}$ calculus is shown in Figure 1. The semantics is defined in big-step call-by-value style with explicit closures. We can think of evaluation as being split between two judgements $H \vdash t \Downarrow^1 v$ and $H \vdash t \Downarrow^2 v$ for first-class and second-class evaluation, respectively, or as one parameterized judgement $H \vdash t \Downarrow^n v$. An auxiliary definition $H^{[\leq n]}$ restricts H to bindings of names x^m with $m \leq n$. For identifiers, first-class evaluation requires a first-class identifier (Evar). For abstractions, first-class evaluation removes all second-class identifiers from the environment that is to be stored in the closure, rendering them inaccessible (Eabs). For applications, the function itself is evaluated second-class, the function body is always evaluated first-class, and for the argument, it depends on whether the formal parameter is a first-class or second-class symbol (Eapp). These evaluation rules formalize the key ideas stated earlier for combining first-class and second-class values in the same language.

3.2 Mechanized Implementation

To prove various properties of our system, we have mechanized it in Coq. For this implementation, we had to pick a representation of bindings and environments. We chose a representation based on DeBrujin levels, where names are numeric indexes into the environment, from outermost to innermost. In this setting, we assume that all names x in the program are annoted as x^1 or x^2. This structure is canonical taking the environment bindings as a well-formedness condition. To model the two kinds of bindings for x^1 and x^2, as well as the restriction operator $H^{[\leq n]}$, we found it useful to implement environments as triple $H = (H^1, H^2, k)$, where H^1 holds the x^1 bindings, H^2 holds the x^2 bindings, and k is a lower bound on the accessible bindings in H^2. The last bit deserves some further explanation. We can picture an environment H as

$$H = \{v_1^1, \ldots, v_m^1\}, \{\underbrace{v_1^2, \ldots, v_{k-1}^2}_{\text{inaccessible}} \mid v_k^2, \ldots, v_n^2\}$$

where the vertical bar $|$ is at position k in the list of x^2 bindings, denoting that only binding to the right of it, i.e., for names represented by DeBrujin levels $\geq k$ are valid indexes. Restricting H to $H^{[\leq 1]}$ moves the bar k all the way to the right, disabling all existing second-class bindings:

$$H^{[\leq 1]} = \{v_1^1, \ldots, v_m^1\}, \{\underbrace{v_1^2, \ldots, v_{k-1}^2, v_k^2, \ldots, v_n^2}_{\text{inaccessible}} \mid\}$$

However, new second-class bindings can be added to the right. A restriction $H^{[\leq 2]}$ leaves the environment unchanged.

This representation, which preserves the structure of environments, considerably simplifies the proofs, as we do not need to worry about substitution or reasoning about sets of names. A variation would be to use DeBrujin indexes, i.e., to index environments from the right instead of the left. This removes the need for a numeric bound k at this point, at the expense of complicating developments for type systems with abstract types, which require shifting of indexes when moving type variables across contexts.

To prove properties about evaluation, such as type soundness, we follow the technique of Siek [55] and Ernst, Ostermann and Cook [13], which consists in extending a big-step operational semantics $\Downarrow$ to a total evaluation function `eval` by adding a numeric fuel value and explicit `Timeout` and `Error` results:

$$r ::= \text{Timeout} \mid \text{Done } (\text{Error} \mid \text{Val } v)$$

The fuel value can serve as induction measure.

3.3 Lifetime Properties

Based on this high-level semantics, which is just an annotated simply-typed λ-calculus, we prove that second-class values exhibit the expected second-class characteristics. In particular, we show that the lifetimes of second-class values follow a stack discipline. To do this, we define a lower-level operational semantics $H, S_1..S_k \vdash t \Downarrow_s^n v$, shown in Figure 2, that again splits environments into first-class and second-class parts, but in addition maintains a stack of second-class environments through all function calls. Closures contain a first-class environment but only a stack pointer to represent the second-class part. When invoking a closure, the stack pointer will be used to find the correct caller environment S_i in which to resolve the callee's free second-class variables. This S_i will become the new top stack frame. If the stack pointer is 0, as is the case for first-class functions, the empty environment will be used. Function arguments will be either added to the environment (first-class) or to the top stack frame (second-class).

We define a predicate wf^n to define well-formedness of values v and classify them as first- or second-class value. An environment can be first or second-class, only if all elements are well-formed first- or second-class values, respectively. Well-formed first-class values include exactly the constants c and closures with no second-class references: $wf^1\ c$ and if $wf^1\ H$, then $wf^1\ \langle H, 0, \lambda x^n.t\rangle$. Well-formed second-class values are all well-formed values, since first-class values are also second-class. The abstractions need to have a first-class environment heap reference: $wf^1\ H$, then $wf^2\ \langle H, i, \lambda x^n.t\rangle$.

Lemma 3.1. *Evaluation produces only well-formed values:*

$$\frac{wf^1\ H \qquad wf^2\ S_1..S_k \qquad H, S_1..S_k \vdash t \Downarrow^n v}{wf^n\ v}$$

Proof. By induction on the derivation. □

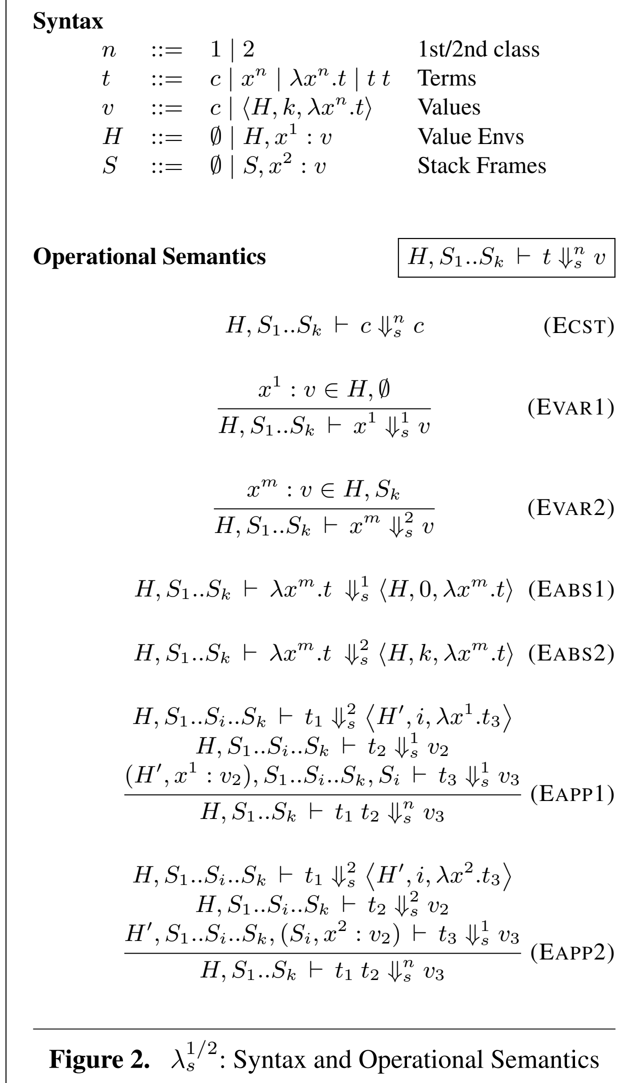


Figure 2. $\lambda_s^{1/2}$: Syntax and Operational Semantics

This result establishes that first-class evaluation can only yield values that contain no stack references. The interesting case in the proof is in (Eapp1), when H is extended with a new binding. We know by induction that the new value is well-formed, too. Thus, we can establish the following stronger result.

Theorem 3.2. *Evaluation never leaks stack references: If $wf^1\ H$, then for all H' encountered in a derivation of $H, S_1..S_k \vdash t \Downarrow^n v$, we have $wf^1\ H'$.*

Proof. By induction on the derivation, and Lemma 3.1. □

We now define equivalence relations $\sim$ between values and environments from $\lambda^{1/2}$ and $\lambda_s^{1/2}$, respectively. In order to make the notation clearer, the environment of $\lambda^{1/2}$ will be explicitly (H^1, H^2) and the closures $\langle H^1, H^2, \lambda x^n.t \rangle$. For $\lambda_s^{1/2}$, closures take the shape $\langle H, i, \lambda x^n.t \rangle$. Equivalence between values is with respect to a stack $S_1..S_k$. The key case for closures looks up the correct stack frame given the stack pointer:

$$\frac{S_1..S_i..S_k \vdash (H^1, H^2) \sim (H, S_i)}{S_1..S_i..S_k \vdash \langle H^1, H^2, \lambda x^n.t \rangle \sim \langle H, i, \lambda x^n.t \rangle}$$

With these correspondences at hand, we can show that the total formulations of the high-level semantics $\Downarrow^n$ and low-level semantics $\Downarrow^n_s$, `eval`^{n} and `eval`^n_s, are equivalent.

Theorem 3.3. *The fully environment-based and (second-class) stack-based semantics are equivalent. For all k,*

$$\frac{S_1..S_k \vdash (H^1, H^2) \sim (H, S_k)}{S_1..S_k \vdash eval^{\,n} k\ (H^1, H^2)\ t \sim eval^{\,n}_{\,s} k\ (H, S_1..S_k)\ t}$$

Proof. By induction on the fuel value k. □

Using `eval`^{n} and `eval`^n_s instead of $\Downarrow^n$ and $\Downarrow^n_s$ in the proofs yields a result that includes equivalent error and divergence behavior. Importantly, the result holds for empty environments, as $(\emptyset, \emptyset) \sim (\emptyset, \emptyset)$.

Corollary 3.4. *The lifetimes of second-class bindings in $\lambda^{1/2}$ follow a stack discipline.*

From this result follows that a realistic implementation can use the more efficient stack-based semantics as a basis, and also that second-class values can be used as temporary access tokens.

3.4 Type System and Static Checking

Having defined the correct desired runtime behavior, we would like to be able to rule out erroneous executions statically. To this end, we define a type system for $\lambda^{1/2}$, shown in Figure 1, and prove it sound with respect to the given operational semantics. The syntax of types contains a function type $T_1^n \to T_2$ where n distinguishes second-class and first-class parameters, respectively.

Type assignment aims to mirror the operational semantics. Again the rules can be read as two judgements, $G \vdash t :^1 T$ and $G \vdash t :^2 T$ for first-class and second-class type assignment, or as one parameterized judgement $G \vdash t :^n T$. For identifiers, first-class typing requires a first-class identifier (Tvar). For abstractions, first-class typing removes all second-class identifiers from the environment and all function bodies are treated as first-class (Tabs). For applications, the function itself is second-class, and the formal parameter type decides the type assignment of the argument (Tapp).

For the proof of type soundness, we follow the technique of Siek [55]. We need straightforward auxiliary judgements $v :^n T$ that assign types to runtime values and $G \vDash H$ that establishes consistency between type and value environments.

Theorem 3.5. *The type system is sound with respect to the operational semantics: for all k, if* `eval` *does not time out, its result is also not stuck, and the result is well typed.*

$$\frac{G \vdash t :^n T \qquad G \vDash H \qquad eval^n k\ H\ t = Done\ r}{r = Val\ v \qquad v :^n T}$$

Proof. By induction on the fuel value k, and case analysis on the term t, using helper lemmas to establish soundness of environment lookup. □

This result implies that "well-typed programs don't go wrong", i.e., that all runtime failures are transformed into compile errors. This includes failures caused by trying to access second-class values that have been removed from an environment via a $H^{[\leq n]}$ operation.

Corollary 3.6. *All well-typed programs are guaranteed to respect stack-based lifetimes for second-class values.*

This basic model based on simply-typed λ-calculus captures the essence of combining first- and second-class values in a single language, and it already enables us to write interesting programs with second-class capabilities. The motivating examples from Section 2 are almost entirely expressible with just the λ-calculus fragment, except for some simple uses of parametric types, and of course assuming that we access to the filesystem. However, we can gain additional expressiveness by moving to richer type systems, as we motivate and formalize next.

4. Extension to Richer Types

We now move beyond simply-typed λ-calculus as a base calculus. Our motivation is twofold. First, we would like to gain confidence that our model scales to realistic languages, in particular Scala, since this is the testbed for our case studies. Second, we show that specific features, such as subtyping and path-dependent types, enable interesting programming patterns with second-class capabiliites.

Parametric Polymorphism In a realistic language, we clearly want some form of parametric polymorphism to support generic data structures, and we could base our model on System F instead of λ-calculus without much difficulty. For second-class capabilities, there are also many specific use cases: for example, an exception throwing capability `CanThrow` can be refined to designate specific kinds of exceptions it enables to throw by using `CanThrow[IOException]`, `CanThrow[NullPointerException]`, and so on.

Subtyping Subtyping is specifically useful to create hierarchies of capabilities, some more general than others. For example, instead of a simple `CanIO` capability, we can envision a hierarchy as follows:

```
type CanIO                // unspecific IO
type CanDisk   <: CanIO   // local filesystem
type CanNet    <: CanIO   // network send/receive
type CanHadoop <: CanNet  // remote filesystem
```

Using advanced language features like mixin-composition, reflected as intersection types on the type level, we can create and request capabilities like `CanDisk & CanHadoop` that enable sets of functionality as a whole, and specific capabilities can be masked via up-casts; for example, treating a `CanDisk & CanHadoop` capability as its supertype `CanNet`.

Path-Dependent Types In Section 2, we have used second-class `File` objects directly as capabilities. Sometimes this is undesireable, for example, when only parts of the functionality of `File` objects should be guarded by a capability. For those cases, we can use *path-dependent types* to associate an external capability with a *specific* file object, and require this capability only for some of the operations:

```
class File(val path: String) {
  type Cap
  def read(implicit @local c: Cap): String = ...
}
```

Each `File` object now has an abstract type member `Cap`, and reading the file requires a second-class capability of that type. The `File`'s path, by contrast, can be used freely without accessing the filesystem, and extracting it hence does not require the file to be opened.

Method `withFile` now introduces both the file, which is first-class, and the implicit capability `c`, which is second-class and has type `file.Cap`, i.e., a path-dependent type referencing a *specific* file object. Here is a possible usage scenario:

```
val usedFiles = new ArrayBuffer[File]()
withFile("out.txt") { file => implicit c =>
  usedFiles += file
  ... file.read() ... // ok, capability available
}
println("this program used the following files:")
for (f <- usedFiles)
  println(f.path)
```

This means that we can freely let the file object escape, knowing that we will not be able to read from it outside of a `withFile` scope without the capability. We make key use of a similar model in our case study on region-based memory (Section 8) and for checked exceptions in the presence of parallel collections (Section 7).

4.1 Formal Model

We have shown why we want richer type systems than λ-calculus as our base. We could extend System F for parametric polymorphism alone, or $F_{<:}$ for parametric polymorphism plus subtyping. But in order to cover all the features we want, including path-dependent types, we base our exposition on the DOT (Dependent Object Types) calculus [2, 48, 49], that has been proposed as a foundation for Scala's type system. More precisely, we use a slightly restricted variant of DOT called $D_{<:}$ [48], which encodes $F_{<:}$ in a relatively straightforward way, and which we extend to $D_{<:}^{1/2}$.

System $D_{<:}$ is at its core a system of first-class type objects and path-dependent types. Type objects can be seen as single-field records containing an abstract type member. Type selections, or path-dependent types serve to access these abstract type members.

Syntax

$$
\begin{array}{lcl}
T & ::= & \bot \mid \top \mid \text{Type}\ T..T \mid x.\text{Type} \mid (x^n : T) \to T \\
t & ::= & x \mid \text{Type}\ T \mid \lambda x^n.t \mid t\ t \\
\Gamma & ::= & \emptyset \mid \Gamma, x^n : T \\
v & ::= & \langle H, \lambda x^n : T.t\rangle \mid \langle H, \text{Type}\ T\rangle
\end{array}
$$

Subtyping $\boxed{\Gamma \vdash S <: U}$

$$\Gamma \vdash \bot <: T\ (\text{SBOT}) \qquad \Gamma \vdash T <: \top\ (\text{STOP})$$

$$\frac{\Gamma \vdash x : \text{Type}\ T..\top}{\Gamma \vdash T <: x.\text{Type}}\ (\text{SSEL1}) \qquad \frac{\Gamma \vdash x : \text{Type}\ \bot..T}{\Gamma \vdash x.\text{Type} <: T}\ (\text{SSEL2})$$

$$\Gamma \vdash x.\text{Type} <: x.\text{Type} \quad (\text{SSELX})$$

$$\frac{\Gamma \vdash S_2 <: S_1\ ,\ U_1 <: U_2}{\Gamma \vdash \text{Type}\ S_1..U_1 <: \text{Type}\ S_2..U_2}\ (\text{SSELAX})$$

$$\frac{m_2 \le m_1 \quad \Gamma \vdash S_2 <: S_1 \quad \Gamma, x : S_2 \vdash U_1 <: U_2}{\Gamma \vdash (x^{m_1} : S_1) \to U_1 <: (x^{m_2} : S_2) \to U_2}\ (\text{SALL})$$

Type assignment $\boxed{\Gamma \vdash t : T}$

$$\frac{x^m : T \in \Gamma^{[\le n]}}{\Gamma \vdash x :^n T}\ (\text{TVAR})$$

$$\Gamma \vdash \text{Type}\ T :^n \text{Type}\ T..T \quad (\text{TTYP})$$

$$\frac{\Gamma^{[\le n]}, x^m : T_1 \vdash t_2 :^1 T_2}{\Gamma \vdash \lambda x^m.t_2 :^n (x^m : T_1) \to T_2}\ (\text{TABS})$$

$$\frac{\Gamma \vdash t :^2 (x^m : T_1) \to T_2\ ,\ y :^m T_1}{\Gamma \vdash t\ y :^n T_2[y/x]}\ (\text{TDAPP})$$

$$\frac{\Gamma \vdash t :^2 (x^m : T_1) \to T_2\ ,\ t_2 :^m T_1}{\Gamma \vdash t\ t_2 :^n T_2}\ (\text{TAPP})$$

$$\frac{\Gamma \vdash t :^n T_1\ ,\ T_1 <: T_2}{\Gamma \vdash t :^n T_2}\ (\text{TSUB})$$

(runtime sybtyping and value type assignment not shown)

Figure 3. System $D^{1/2}_{<:}$: a generalization of $F_{<:}$ with value types and path-dependent types.

The syntax and typing rules are shown in Figure 3. The type language includes $\bot$ and $\top$, as least and greatest element of the subtyping relation, first-class abstract types (Type $T_1..T_2$), lower-bounded by T_1 and upper bounded by T_2, type selections on a variable x.Type (i.e., path-dependent types), where x is a term variable bound to a type object, and finally dependent function types $(x^n : T) \to T$. The term language includes variables x, creation of type objects (Type T), λ-abstractions $\lambda x^n.t$, and applications $t_1\ t_2$.

The subtyping relation can compare type selections with the bounds of the underlying abstract types, and compare type objects and dependent functions, respectively. Type assignment contains fairly standard cases for dependent abstraction and application.

To relate System $D_{<:}$ to Scala, let us take a step back and consider two ways to define a standard `List` data type:

```
class List[E]          // parametric, functional style
class List { type E }  // modular style, with type member
```

The first one is the standard parametric version. The second one defines the element type `E` as a type member, which can be referenced using a path-dependent type. To see the difference in use, here are the two respective signatures of a standard `map` function:

```
def map[E,T](xs: List[E])(fn: E=>T): List[T] = ...
def map[T](xs: List)(fn: xs.E=>T): List & { type E=T } = ...
```

Again, the first one is the standard parametric version. The second one uses the path-dependent type `xs.E` to denote the element type of the particular list `xs` passed as argument, and uses a refined type `List & { type E=T }` to define the result of `map`.

It is easy to see how the modular surface syntax directly maps to the formal $D_{<:}$ syntax, if we express fully abstract types `{ type E }` as (Type $\bot..\top$) and concrete type aliases `{ type E=T }` as (Type $T..T$). It is also important to note that the modular style with first-class type objects can directly encode the functional style, which corresponds to bounded parametric polymorphism as in System $F_{<:}$, but with increased expressiveness due to the $\bot$ type and potential lower bounds on type variables.

First-Class and Second-Class Values Since the stratification between first- and second-class values happens on the level of identifiers and bindings, not types, parametric polymorphism does not pose major difficulties. Still, moving to a system based on subtyping requires an additional result:

Lemma 4.1. *First-class values can be treated as second-class values:*

$$\frac{H \vdash t \Downarrow^n v \qquad n \le m}{H \vdash t \Downarrow^m v} \qquad \frac{G \vdash t :^n T \qquad n \le m}{G \vdash t :^m T}$$

Proof. By induction over the respective derivations, showing that the evaluation and type assignment rules for second-class values subsume those for first-class values. ☐

This result entails that one can admit coercions from first-class to second-class values, and thus eta-expand t of type $T_1^2 \rightarrow T_2$ to $\lambda x^1.t\, x^1$ of type $T_1^1 \rightarrow T_2$. Thus, we can define a subtyping relation that justifies $T_1^2 \rightarrow T_2 <: T_1^1 \rightarrow T_2$.

The operational semantics for $\mathrm{D}_{<:}^{1/2}$ is the same as for $\lambda^{1/2}$, with an additional rule for construction of type values:

$$H \vdash \text{Type } T \Downarrow^n \langle H, \text{Type } T \rangle$$

We can prove type soundness using the same overall technique as for $\lambda^{1/2}$. The proof follows the one given for $\mathrm{D}_{<:}$ by Rompf et al. [48].

Theorem 4.2. *Type soundness for $D_{<:}^{1/2}$. If* `eval` *does not time out, it returns a well-typed value:*

$$\frac{\Gamma \vdash t :^n T \qquad \Gamma \vDash H \qquad \mathit{eval}^{\,n} k\ H\ t = \mathit{Done}\ r}{r = \mathit{Val}\ v \qquad H \vdash v :^n T}$$

Proof. By induction on the fuel value k. □

4.2 Arbitrary Privilege Lattice

The model presented so far enables us to control the lifetimes of capabilities, but in many settings, not all capabilities have the same status. What if we want to have a more control over the relative visibilities of capabilities, while ensuring their non-escaping status as non-first-class values? Suppose we want to prevent race conditions or out-of-order writes when a file is passed to a non-deterministic higher-order function such as a parallel `reduce` operation, yet allow non-deterministic reads, which are far less dangerous:

```
withFile("file.txt") { f =>
  f.readCharAt(0)      // ok
  f.print(...)         // ok: deterministic context
  reduce(data) { (a,b) =>
    f.readCharAt(a)    // ok
    f.print(...)       // error: race condition
    a+b
} }
```

To model such scenarios, we need to treat capabilities for reading and writing differently. We informally introduce a degree of "second classiness", which we achieve by parameterizing `@local` as `@local[P]`, where `P` denotes a *privilege level* and is in *contravariant* position. Implicitly, a `@local` annotation denotes the most restricted privilege level, while its absence denotes no restrictions (first class). In general, annotating a function parameter with `@local[P]` requires each free reference of a passed closure to be annotated with `@local[T]`, for some `T <: P`. In Scala, we can represent privileges directly as types, and their relationships via subtyping: `@local[Nothing]` denotes first-class, equivalent to no annotation, and `@local[Any]` denotes second-class, equivalent to just `@local`, and any other type `P` defines a level inbetween.

We now exploit this mechanics to implement the example above. The key is that files themselves will live at a less restricted (i.e. smaller) level than write capabilities:

```
trait R // privilege level >: Nothing (1st) and <: Any (2nd)
class File(val path: String) {
  def print(s: String)(implicit @local w: CanWrite) { ... }
  def readCharAt(i: Int) = { ... }
}
def withFile[U](...)(@local fn: (@local[R] File) => U): U
def reduce[U](...)(@local[R] fn: (U,U) => U)
```

We introduce a privilege level `R` inbetween first- and second-class and implement `withFile` to make file objects available at this new level. In the simplest model, files serve as their own read capabilities, but the `print` method requires an additional *second-class* `CanWrite` capability.

Method `reduce` takes its function argument as `@local[R]`, so files can be accessed from the closure, but truly second-class objects and in particular write capabilites will be precluded. A single global `CanWrite` capability is all that is left to complete the example.

As an alternative, we can model read and write capabilities specific to a given file as path-dependent types, extending the example from the beginning of Section 4:

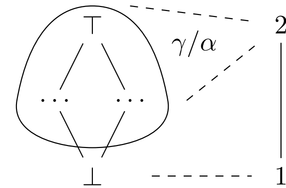
```
class File(val path: String) {            // path-dependent
  type CapW <: CanIO; type CapR <: CanIO  // capabilities
  def print(s: String)(implicit @local w: CapW) { ... }
  def readCharAt(i: Int)(implicit @local[R] r: CapR) = ... }
}
```

In this model, the definition of `withFile` needs to introduce both the `CapR` and the `CapW` objects as separate "fractional" [7] capabilities, with different privilege levels:

```
withFile(path) { f => implicit cr => implicit cw => ... }
```

One could go further and require unequal privilege for sequential reads or random-access writes, thus extending the privilege lattice to more than three levels.

Formal Model We generalize the binary first- vs second-class distinction to an arbitrary privilege lattice L. We require a Galois connection γ, α between L and the lattice $\{1, 2\}_{\leq}$, which maps $\top$ to 2 and $\bot$ to 1 via its concretization function γ. All values except $\top$ and $\bot$ can be mapped to either 1 or 2. In the limit, where everything except $\bot$ is mapped to 2, the previous second-class lifetime guarantees extend to all non-first-class bindings:



While picking specific static lattices may be of interest, the key application relies on a much more general insight: in a system with subtyping, we can use the underlying type lattice as privilege lattice.

In the case of $\mathrm{D}_{<:}^{1/2}$ and similar systems, we can use the *types* $\bot$ and $\top$ to denote first- and second-class values, respectively. Any desired privilege lattice can be built within a program from phantom types that are in a corresponding sybtyping relation. As already discussed, in Scala, we

achieve this by parameterizing `@local` as `@local[P]`, where `@local[Nothing]` denotes first-class, equivalent to no annotation, and `@local[Any]` denotes second-class, equivalent to just `@local`. Any other Scala type `P` must be inbetween `Nothing` $= \bot$ and `Any` $= \top$, and gives rise to a more fine-grained lattice structure, subject to existing subtyping relations between `T` and other types.

To make this change explicit in the context of the formal model in Figure 3, interpret all m as types and replace all occurrences of $m_1 \leq m_2$ with $m_1 <: m_2$.

Privilege Parametricity It is sometimes desirable to abstract over the *level* of privilege in order to prevent code duplication and keep an existing interface unmodified. If a type system includes abstract types, as is the case in $D_{<:}^{1/2}$ and in Scala, abstract types naturally enable such *privilege parametricity*. This means that we can abstract over whether a variable holds first-class or second-class values in a more specific context. The main motivation here is code reuse: we need to write a function or class only once, and we can use it with both first-class and second-class instantiations.

A key use case comes from our handling of the Scala collection library in Section 6. We have already mentioned that method `map` should behave differently for eager and lazy collections:

```
@local def println(x: Any): Unit = ...
list.map(x => println(x))   // ok
stream.map(x => println(x)) // error
```

Thus, these collection implementations need to have different signatures for `map`:

```
def map[B](@local fn: A => B) = ... // eager
def map[B](fn: A => B) = ...        // lazy
```

Lazy collections like `Stream[A]` may leak the closure argument to `map`, and therefore it needs to be first-class. Conversely, for eager collections like `List[A]`, we would like a second-class closure argument.

How can we achieve that `List[A]` and `Stream[A]` can be derived from a common superclass? We use `@local[LT]` in the generic `map` signature, where `LT` is an abstract type parameter defined in base class `Iterable[A]`, and refined to `Nothing` or `Any` (first- or second-class) for specific subclasses:

```
// Abstract base class:
trait Iterable[A] {
  type LT
  type plocal = local[LT]
  def map[B](@plocal fn: A => B)
}
// Implementation classes:
class List[A] extends Iterable[A] {
  type LT = Any
  def map[B](@local fn: A => B) = {
    // implement eager version here
} }
class Stream[A] extends MySeq[A] {
  type LT = Nothing
  def map[B](fn: A => B) = {
    // implement lazy version here
} }
```

This design enables the desired usage patterns shown above.

As we can see, abstract base classes can have abstract privileges that are instantiated to second- or first-class in implementation subclasses. In Section 6, we will discuss code sharing between collections further and demonstrate that we can indeed share large pieces of the internal implementation in our modified version of the Scala library.

4.3 Recursive Functions

Our development so far did not consider recursive functions. Adding recursion does not pose particular difficulties. The simplest and most practical implementation of recursive functions extends rule (Eapp) from Figure 1 to pass the closure object itself as argument to the function. The λ syntax is extended to include the self identifier f^k where k denotes first- or second-class binding as usual:

$$\frac{\begin{array}{c} H \vdash t_1 \Downarrow^k v_1 \\ v_1 = \langle H', \lambda f^k(x^m).t_3 \rangle \\ H \vdash t_2 \Downarrow^m v_2 \\ H', f^k : v_1, x^m : v_2 \vdash t_3 \Downarrow^1 v_3 \end{array}}{H \vdash t_1\ t_2 \Downarrow^n v_3} \quad (\text{EAPP})$$

Note that this modified (Eapp) rule is no longer deterministic, as the evaluation rule for the function needs to match the class of the closure type. A simple way to make the rule deterministic in the formalism is to extend the syntax of function application to determine if the function is first- or second-class: $t_1^k\ t_2$.

For a realistic implementation, this piece of information can easily be extracted from the type assigned to expression t_1. In this setting, recursive functions are also related to the treatment of objects and `this` pointers, as we will discuss.

5. Implementation in Scala

We have implemented a plug-in for the Scala compiler that closely implements the formal system described in Section 3 and Section 4. Given the nature of the Scala language, and the structure of the Scala compiler, a number of aspects needed additional work. First, Scala is a large language with many constructs in addition to λ-calculus and $D_{<:}$. In particular, objects, classes, traits, and separate compilation posed some challenges. Second, the Scala compiler is structured around a global, hierarchical symbol table as opposed to flat environments, so the formal model of removing certain bindings required different implementation techniques, e.g., traversing scope chains to find common ancestors.

To implement the API introduced in Section 2, we define a class `local` as a piece of library code, which the compiler plug-in knows about:

```
package scala.util
class local[-T] extends StaticAnnotation
```

This class can be used as annotation on declarations:

```
@local val log = new File("out.txt")
```

Since the type parameter `T` is contravariant, writing `@local` is equivalent to `@local[Any]`, which denotes a second-class

binding. By contrast `@local[Nothing]` denotes a first-class binding, equivalent to no annotation at all. Any type between `Nothing` and `Any` can be used for finer-grained control, and an abstract type can be used to abstract over the class of binding (Section 4.2).

Scala's first-class functions map to anonymous classes that implement a given base trait `Function1`, with the usual `A=>B` notation as type alias:

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

To model functions with second-class arguments, we provide a subtrait `FunctionEsc1`:

```
trait FunctionEsc1[-A,+B,-LA,+LS] extends Function1[A,B] {
  @local[LS] def apply(@local[LA] x:A): B
}
type '->'[-A,+B] = FunctionEsc1[A,B,Any,Nothing]
```

If `A->B` is the expected type for some closure expression `(x => ...)`, the Scala compiler will automatically synthesize a corresponding object creation with the right signature.

Compared to the theoretical model, we need to worry about objects, traits, and classes in addition to lexical functions. These object-oriented constructs have a more complicated scope structure due to inheritance. Our current implementation is conservative and focuses primarily on the lexical level. Class definitions are treated like first-class functions and cannot access second-class values from their defining scope. The following code is thus illegal,

```
@local val log = ...
class Handler {
  def func() = log.println("A") // error
}
val a = new Handler; a.func()
```

but the same functionality can be implemented like this:

```
@local val log = ...
class Handler {
  def func(@local val log: File) = log.println("A")
}
val a = new Handler; a.func(log)
```

We plan to extend our implementation with a notion of `@local` classes, once all the implications are worked out. This would enable writing the same code snippet above as `@local class Handler`. In practice, we have not found the absence of such a facility limiting.

A key goal of this implementation was to investigate how well second-class values map to real world Scala code. To this end we conducted several case studies, described next.

6. Case Study: Scala Collections

The cornerstone of the Scala standard library is its set of collection classes, supporting a variety of sequence data structures (`List`, `Array`, ...), as well as `Sets`, `Maps` and so on. Methods to traverse and transform collections use higher-order and first-class functions pervasively, making Scala Collections an excellent testbed to evaluate the expressiveness of our implementation of second-class values. The goal of this experiment is to assess how precisely we can model second-class behavior for functions passed as arguments. As described in Section 2, we would like a standard `List.map` call to treat its argument function in a second-class way, whereas a distributed or lazy collection would demand a true first-class function.

The key problem is that, for example, `List` is eager but `Stream` is lazy, and `Array` is sequential but `ParArray` is parallel. Yet, all the classes share the same base class hierarchy [44]. Most functionality is implemented only once, and reused among leaf classes. The Scala Collections library already has a large number of classes and traits (`GenTraversableOnce`, `IterableLike`, ...), so that adding another dimension to distinguish eager and lazy collections would not work well.

The solution we found makes crucial use of privilege parametricity. To handle lazy and eager collections in a uniform way, we use `@local[LT]`, where `LT` is an abstract type parameter defined in a base class, that can be instantiated to `Nothing` or `Any` (first- or second-class) depending on the collection type. The corresponding code has been shown already in Section 4.2.

Note that method `foreach`, in contrast to `map` is eager for all collections. It uses `@local` directly instead of `@plocal`. Note further that we have omitted the return type of `map` above. In practice the situation is slightly more complicated, as transformer methods on collections use F-bounded polymorphism to return an instance of the same class (or a compatible one) as the object itself.

Evaluation We have achieved the abovementioned behavior without any code duplication or addition of new types, by changing <1%[2] of SLOC in the Scala Collections API, comprising 29310 SLOC total. Out of the 277 lines changed, over 75% are global search-replace that inserts `@local` annotations. The main challenge was to propagate the type-dependant type `LT` and deal with `*Proxy[Like]` traits (which we eventually removed as they are deprecated anyway).

7. Case Study: Checked Exceptions

Given our modified version of the Scala Collections library, whose higher-order traversal and transformer methods correctly track first-class and second-class arguments, we would like to put these facilites to some good use. We have already seen how we can model operations, like `println`, as second-class functions. These serve as capabilities and control when and where the associated operation and its side effect can happen. Thus, the question bears asking whether we can use the same model for more general classes of side effects.

We have extended the Scala Library further, with a notion of checked exceptions. Checked exceptions can be seen as an instance of a type-and-effect system [16], and in fact, Java's support for checked exceptions is probably the only type-and-effect system in practical use today. The key idea is to include the side effects of an expression in its type. However,

[2] Only meaningful lines of code, i.e., not Scala docs, were counted.

a fundamental trade-off between usefulness (larger, more precise types) and usability (smaller, more comprehensible types) makes such effect systems hard to use in practice.

In our case, exceptions might only be allowed to be thrown if an appropriate `throw` function is available, and we would like to enforce that this can only happen within a try/catch block. With our support for second-class values, we can define `try` blocks as follows:

```
def try[T](fn: (@local Exception => Nothing) => T): Option[T]
```

A realistic implementation would also contain a `catch` block, but here we content ourselves with returning `Option[T]` values. Given the definition of `fn`'s parameter as `local`, client code may use `try` as follows,

```
try { throw =>
  throw(new Exception)  // ok: throw cannot escape
}
```

but the function passed as argument to `try` cannot leak the value of `throw`. Inside such a `try` block we can use `throw` in other safe (i.e., second-class) positions but not in unsafe ones, where it could escape:

```
def safe(@local fn: () => Any): Int = ...
def unsafe(fn: () => Any): Int = ...
try { throw =>
  safe { () => throw(new Exception) }    // ok: safe
  unsafe { () => throw(new Exception) }  // not ok
}
```

Effect Polymorphism It is easy to see that we have utilized the same pattern in `safe` as in the previous definition of `map` on `List`s. In fact, the following code is perfectly legal:

```
try { throw =>
  map(xs) { x =>
    if (x > 0) x else throw(new Exception)
  }
}
```

As we would expect, we can use `throw` in nested second-class functions within the dynamic scope of `try` but not as a first-class value that might escape.

It is important to note that we are using the same `map` implementation independently of whether the function we are passing as argument may throw an exception or not. This would not be the case with monads or with Java's checked exceptions, where the following two different `map` declarations would be needed (example from Rytz [54]):

```
public <U> List<U> map(Function <T, U> f);
public <U, E extends Exception> List<U>
                  mapE(FunctionE<T, U, E> f) throws E;
```

Similar effect polymorphism can also be achieved in the context of type-and-effect systems but with significant effort [54, 53].

Implicit Capabilities It is also worth noting that we do not have to use the object `throw` itself as a capability. We might as well define the `throw` method globally and have it require an additional argument of a designated capability type.

```
def throw(e: Exception)(implicit cap: CanThrow): Unit = ...
```

In fact, it has been proposed to use such a pattern for more flexible handling of side effects in general [36], for example:

```
def println(s: String)(implicit @local cap: CanIO): Unit = ...
```

As we will see below, this pattern is especially useful when the main object in question needs to be first-class for some other reason. In Scala, parameters declared as `implicit` will have the arguments resolved and inserted automatically by the compiler, so one can write

```
throw(new Exception)
```

and the Scala compiler would automatically insert `cap` as the missing capability argument for `throw` from the context.

In summary, scoping rules for second-class values ensure that such objects cannot be copied, stored, or escape by other means, which makes them ideally suited to serve as access tokens or capabilities. With effect capabilities as regular program values, specifying new classes of effects becomes almost trivial, an important benefit for expressive libraries and embedded DSLs (domain-specific languages).

Parallel Collections A subtlety that arises from the inherently blocking nature of parallel operations has a rather unexpected implication with respect to effects. Since a blocking thread may be interrupted, it needs to handle an `InterruptedException`, which means that all parallel collection operations need the exception-throwing capability `CanThrow`. There are two choices: a pragmatic one, merely converting `InterruptedException` to `RuntimeException`; or the rigorous one, requiring a proper capability. We went with the latter, to investigate the effort of propagating exception capabilities, thus stress-testing our type system. To accommodate this without breaking the API, we exploit abstract types, type bounds and *implicit default arguments*:

```
type CanSeq  // non-parallel dummy capability
type CanPar <: CanSeq with CanThrow
trait GenIterable[A] {  // common super-trait
  type Cap >: CanPar
  def foreach[U](@local fn: A => U)(
      implicit @local cap: Cap)
}
trait Iterable[+A] extends GenIterable[A] {
  type Cap = CanSeq
  implicit val capDummy = new CanSeq {}
  override def foreach[U](@local fn: A => U)(
      implicit @local cap: CanSeq = capDummy) { ... }
}
trait ParIterable[+A] extends GenIterable[A] {
  type Cap = CanPar
  override def foreach[U](@local fn: A => U)(
  implicit @local cap: CanPar) { // note CanPar <: CanThrow
    ...
    doInterruptible(...)  // using cap as CanThrow
  }
}
```

The above implementation ensures that a (potentially) parallel method can only be called if the corresponding implicit `CanPar` capability is in scope, e.g.:

```
val coll: Iterable[Int] = ...
val collPar: ParIterable[Int] = ...
val collGen: GenIterable[Int] = collPar  // common base type
coll foreach { x => ... }  // ok (using default capDummy)
collPar foreach { x => ... }  // error: missing capability
collGen foreach { x => ... }  // error: could be parallel
```

Annotation overhead The default implicit arguments are essential, since they allow the compiler to insert `capDummys` based on a scope of callee's (super)type rather than leaving this burden at the call site. In the above case, putting capability arguments was the responsibility of non-parallel collections, rather than relying on callers to have them availiable in their scopes, which is fragile (prone to shadowing or ambiguity, and not resistant to passing other implicit arguments). For user functions we can alleviate this burden by providing an implicit dummy capability that can be imported as a first-class from a module. To show this eliminates overhead in dispatching capabilities, consider the following example:

```
def process[A](coll: GenIterable[A])(
  implicit @local cap: coll.Cap)
```

Note a path-dependent capability argument. It enables reuse of a *single* implementation for subtypes that require different levels of capabilities (forming a lattice), and subsumes optional capabilities. Our function works with both parallel and sequential collections, as the following snippet illustrates:

```
import CapDummy._
process(Range(0, 9))          // ok (using imported capDummy)
process(ParRange(0, 9))       // error (missing CanPar cap.)
...
def parallelContext(implicit @local canPar: CanPar) {
  process(ParRange(0, 9))     // ok
}
```

Evaluation We modified the Scala compiler to signal all uses of checked exceptions according to the Java definition (excluding `Errors` and `RuntimeExceptions`) as compile errors, thus requiring the use of our `try` facility above. Additionally, `throw` markers were required for interfacing with Java methods, and finally the `no` unsafe hooks were used to comply to signatures of inherited Java methods.

We have evaluated the effort of using the above three facilities, as well as propagating our `CanThrow` (and `CanPar`) capabilities required for throwing exceptions, on the entire Scala standard library, comprising 43040 SLOC. Manual effort was due to the former and placing `Cap` type definitions in: a few Collection types (deep hierarchy) and many subtypes of mixins (shallow hierarchy). Adding capability parameters was largely automated (using a PERL-based regular expression engine), guided by compile errors. In total, $\sim$3% SLOC is affected, and the breakdown is as follows:

	try	throw	no	types	CanThrow	Cap
#	54	75	38	26	264	971

In the above effort breakdown, most `throws` and `nos` come from code related to IO and processes (which exploits JVM). A high number of `trys` is due to a trade-off we needed to make to keep compatibility with user code; we could not require a capability in an `Any`'s core method such as `==` just because it might be comparable with a parallel collection.

8. Case Study: Region-Based Memory

Most modern high-level languages run on managed runtimes such as the JVM, .NET CLR, or JavaScript VMs. All these platforms come with automatic memory management, garbage collection, and built-in memory safety. Sometimes it is, however, desirable to allocate memory outside the managed heap: to reduce garbage collection overhead, to address larger amounts of memory, or just to have more control over memory layout. Unfortunately, then the safety guarantees of the platform are invalidated and segfaults bound to happen.

We present a small off-heap memory library based on scoped capabilities that preserves memory safety by imposing a region-based object lifetime policy. Our implementation is inspired by a recent Scala library[3] by Shabalin et al. with much larger functionality, but without such guarantees.

Our implementation is based on two interfaces: `Data`, corresponding to an off-heap chunk of memory, and `Region`, from which such chunks can be allocated. We will discuss the role of the type parameter and the implicit arguments.

```
trait Data[T] {
  def size: Long
  def apply(i: Long)(implicit @local cc: T): Long
  def update(i: Long, x:Long)(implicit @local cc: T): Unit
}
trait Region {
  type Cap
  def alloc(n: Long)(implicit @local c: Cap): Data[Cap]
}
```

The interface further provides a scoped method `withRegion` that can be used as follows:

```
withRegion[Long](1000) { region => implicit c =>
  val arr = region.alloc(300)  // type: Data[r.Cap]
  arr(0) = 1; println(arr(0))
  ...
}
```

The types ensure statically that data object `arr` cannot be used outside the scope of the `withRegion` call. Here is the implementation of `withRegion`:

```
abstract class F[B] { def apply(r: Region): r.Cap -> B }
def withRegion[T](n: Long)(f: F[T]): T = {
  object cap
  val r = new Region {
    type Cap = cap.type
    var data = malloc(n)
    var p = 0L
    def alloc(n: Long)(@local c: Cap) = new Data[Cap] {
      def size = n
      val addr = p
      p += n
      def apply(i: Long)(implicit @local c: Cap) =
        data((addr+i).toInt)
      def update(i: Long, x:Long)(implicit @local cc: Cap) =
        data((addr+i).toInt) = x
    }
  }
  try f(r)(cap) finally free(r.data)
}
```

For safety, all `Data` objects need to be guarded by their `Region`. On the other hand, we cannot mark the `Region` `@local`, because data objects actually need to store a reference to the region. The solution is to introduce external capabilities. The way `withRegion` is implemented, a region and its capability always obey the same scope.

[3] https://github.com/densh/scala-offheap

As an extension, we might add bounds checking with the checked exceptions implementation from Section 7. Now, we need to use two scoped introduction forms:

```
withRegion[Long](1000) { r => c =>  try { throw =>  ... } }
```

Instead, we can just as well use the alternative form:

```
try { throw =>  withRegion[Long](1000) { r => c =>  ... } }
```

Region-based memory systems have also been proposed based on monads, phantom types, and rank-2 polymorphism [22]. These and other approaches based on (layered) monads offer comparable guarantees, but they require users to rewrite their code in monadic style throughout, which has well-established shortcomings.

Systems that enforce a non-escaping property using rank-2 polymorphism do so by introducing additional type constraints, requiring the function passed to the `withRegion` equivalent to return a monad instance which is parameterized with the phantom type. By contrast, our `withRegion` blocks can return any type, and we just require capabilities to be present in the context.

Since types are flexible, we can independently define "checked" features like regions, exceptions, and IO, and use them together, whereas composition is more complicated even with monad transformers and has to be planned ahead. We have also no issues changing the order of our scoped constructs, which would lead to *different* monadic types.

9. Case Study: Program Generation

Multi-stage programming [63, 52], a form of runtime code generation, is a popular way to implement high-performance DSLs [10, 11, 9, 59, 23, 60, 58] and specialized numeric kernels [45, 38]. In Scala, we can provide a shallow DSL interface on top of low-level code generation facilities, so that users can write, for example,

```
genloop(200) { x => ... }
```

to emit corresponding C code:

```
for (int x37 = 0; x37 < 200; x37++) { ... }
```

This can be achieved by implementatng `genloop` as follows:

```
case class Code[T](s: String)
def genloop[T](size: Code[Int])
  (@local body: (@local Code[Int]) => Code[T]) = {
  @local val x = Code(freshVar[Int])
  emit(s"for (int $x = 0; $x < $size; $x++) { ${body(x)} }")
}
```

Inside the body of `genloop(200) { x => ... }`, the variable `x` is a regular program value of type `Code[Int]`, representing the auto-generated identifier `x37`. Without the `@local` annotations, it could be stored into a variable and used to construct another piece of code that refers to `x37`, but where `x37` is not in scope. This situation is known as scope extrusion in the literature on program generation, and elaborate type systems have been proposed to prohibit such pitfalls [62, 61]. Here, we prevent scope extrusion using just three `local` annotations in the definition of `genloop`.

Note that there is a problem: we could not write

```
genloop(200) { x => ... genloop(x) { y => ... }}
```

because `genloop` requires a first-class `size` value. We cannot easily change the definition of `genloop`, either, because `size` actually escapes through code generation. In fact, we will encounter this issue anywhere we want to use `x`.

The solution is to leverage a split between interface and implementation traits, which already exists in popular code generation frameworks [52]:

```
trait Interface {
  type LT; type clocal = local[LT]
  def genloop[T](@clocal size: Code[Int])
    (@local body: (@clocal Code[Int]) => Code[T])
}
trait Impl extends Interface {
  type LT = Nothing
  def genloop[T](size: Code[Int])
    (@local body: (Code[Int]) => Code[T]) = {
    ... emit ...
} }
```

Now the argument to `genloop` can be second-class in the user-visible code (as abstract type `LT` is unknown to be different from `Any`), but first-class on the implementation side.

Another potential downside is that we cannot store local `Code` objects in a data structure, even temporary, or return them from functions. Thus, we would rule out many useful generative programming patterns [50].

We can solve this final issue in a similar way to the region-based memory system in Section 8, by not making the code object itself `@local`, but instead adding a capability token. All operations on `Code` types require such a capability, which is specific to the enclosing region.

```
def genloop[T,L0](size: Code[Int,L0])(@local Cap[L0]): {
  type L1 >: L0
  def apply(body:Code[Int,L1]=>(@local Cap[L1] => Code[T,L1]))
}
```

The type bound `L1 >: L0` provides us with a notion of nested regions, ensuring that inner capabilities are more specific subtypes of outer capabilities.

10. Related Work

Strachey [57] publicized the terminology of first-class and second-class entities. The issues around stack-implementability of functions in LISP is also known as the funarg problem [32, 74], and conditions for stack implementation of the simply-typed call-by-value lambda calculus have been given by Banerjee and Schmidt [4]. Hannan presented a type-based escape analysis [18], to infer when variables can be allocated on the stack. The type systems in this paper are similar to Hannan's internal formulation. Taha and Nielson have proposed environment classifiers [62] to ensure non-escaping behavior in the context of program generation. Tanter has proposed notions of scope more fine grained than the usual notions of lexical vs dynamic scope [66].

Capabilities Capabilities as a programming model in dynamic languages were made popular by Miller's E language [30]. The capabilities we study take a similar approach to static checking as recent work on co-effects [41]. The idea is to view program behavior such as side effects not as part of

the program term, but as part of the context, where an appropriate license or capability must be present. Recent proposals call for their use in more general effect systems [36].

Types, Regions and Effect Systems Early work on memory regions based on RC, a dialect of C proposed by David E. Gay [14] that guarantees temporal safety. Effect and region polymorphism [27], for example in the FX programming language [15]. Talpin and Jouvelot [65, 64] introduce subeffecting and present the first effect and region inference algorithm. Lippmeier [26] extends Haskell with mutable state and call-by-value semantics for effectful parts of programs. Tofte and Talpin [70] show how type, region and effect inference can lead to a stack based implementation for languages with reference allocations and updates, as implemented in MLKit [69]. Siek, Vitousek, and Turner present a type and effect system focused on supporting both stack-allocation and expressive higher-order programming patterns (e.g. currying) [56].

Ownership type systems [35, 75, 12] were devised to protect against unintentional aliasing and unexpected side effects in object-oriented programs. The notion of borrowing [19, 33], denoting a temporary transfer of ownership for the duration of a method call, greatly improves the usability of such systems. Borrowed references are subject to similar constraints as our `@local` values. Our contribution is to show that such second-class constraints are useful as a programming model independent of ownership, aliasing, and even of mutable state and a store abstraction altogether. We are also not aware of any ownership type system that provides facilities like our privilege lattice and privilege parametricity (Section 4.2), leveraging host language features such as abstract type members and path-dependent types.

Rust [29] is a recent language by Mozilla that incorporates region-like memory handling based on ownership and borrowing of references. Formalizing Rust's type system is an active area of research, with ongoing efforts [46]. Cyclone [20] is an earlier approach to build a safe dialect of C based on similar ideas.

Type-and-effect systems were proposed by Gifford [16]. Particular systems have been designed for exceptions [17], purity [39], and atomicity [1], among others. Work by Marino and Millstein [28] and by Rytz [53, 54] abstracts such individual systems into generic frameworks for larger classes of effect domains. Nielson and Nielson [34] go from flow-insensitive to flow-sensitive effects.

In the presence of global type inference as in Haskell or ML, it is natural to look for similar procedures for global effect inference. This paper, however, has a different focus, and seeks to provide *programming abstractions* for describing and checking effects. It aims at languages like Scala that combine object-oriented and functional programming with subtyping, parametric polymorphism, and that in general do not support global type inference [37]. In this setting, small and comprehensible type annotations are of key importance.

Monads Monads [31] are a popular approach to encapsulate side effects in pure functional languages, especially Haskell [72, 42]. Despite their great success, they are not without issues. First, programs that use more than one kind of side effect has to combine multiple monads, which is not straightforward [8]. Monad transformers [25] help, but they often require programmers to explicitly lift operations. Second, introducing side effects into existing code requires refactoring that code into monadic style, and also any other code that uses it. The fact that monadic and pure code have incompatible types leads to code duplication, as evidenced by functions `map` and `mapM` in Haskell [26]. Monads have been linked to type-and-effect systems [73] and generalized in a variety of ways, e.g., as parameterized monads [3]. Tate formalized the sequential semantics of "producer" effects using indexed monads [67].

Kiselyov and Shan [21, 22] introduced an SIO monad for lightweight monadic regions, based on phantom types and rank-2 polymorphism, that can also manage file handlers safely and efficiently. Their approach ensures that all resources used are deallocated exactly once, and they support improperly nested lifetimes using explicit lifting operations.

Alternative Systems for Controlling Effects Algebraic effects have gained attention recently [6, 43]. Unlike monads, combining effects is straightforward, but most systems do not check effects statically. Potentially, a program might evaluate to an undefined state where an effect operation appears outside a handler. The situation is different in languages with dependent types [8]. Other lines of work worth noting are linear types [71], uniqueness types [5], witnesses for side effects [68]. Koka [24] is a programming language that can express effect-polymorphism and also constructs like exception handlers that mask effects. In the context of Scala, simple type-and-effect systems have been used to implement Delimited continuations, based on a type-directed selective CPS transform [51]. Effects and static checking are particularly important in the context of domain-specific languages [9, 59, 23, 60, 58]. Applications such as preventing scope extrusion are important in the context of generative programming using Lightweight Modular Staging [50, 52, 47]

11. Conclusions

In this paper, we have studied the interplay of modern first-class values with second-class values, as they were commonplace in the days of ALGOL. While second-class values have largely disappeared from modern languages, a process not unlike gentrification in urban development, we find that second-class values can provide important and practically relevant static guarantees, due to their statically bounded lifetimes. We have formalized type systems containing both first-class and second-class values, proving type soundness and lifetime properties with mechanized proofs in Coq. We

have also implemented our system as an extension of the Scala language, and conducted several case studies. These demonstrate that ideas from the days of ALGOL complement and play well with cutting edge functional and object-oriented programming facilities such as path-dependent types. Our case studies underline the usefulness and practicality of our system and of second-class values as a programming model.

Acknowledgments

We thank Martin Odersky and members of LAMP at EPFL for insightful discussions around effects as capabilities, and for comments on draft versions of this paper. We thank Jeremy Siek, Chung-chieh Shan, Stefan Marr, and the anonymous reviewers for helpful pointers. This research was supported through NSF CAREER award 1553471, a PRF Research Grant, and a faculty startup package from Purdue University.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 63–74, New York, NY, USA, 2008. ACM.

[2] N. Amin, A. Moors, and M. Odersky. Dependent object types. In *FOOL*, 2012.

[3] R. Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376, 2009.

[4] A. Banerjee and D. A. Schmidt. Stackability in the simply-typed call-by-value lambda calculus. *Sci. Comput. Program.*, 31(1):47–73, 1998.

[5] E. Barendsen and S. Smetsers. Conventional and uniqueness typing in graph rewrite systems. In R. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 41–51. Springer Berlin Heidelberg, 1993.

[6] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.*, 84(1):108–123, 2015.

[7] J. Boyland. Checking interference with fractional permissions. In *SAS*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

[8] E. Brady. Programming and reasoning with algebraic effects and dependent types. To Appear in Proceedings of the ACM SIGPLAN international conference on Functional programming, 2013.

[9] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *PACT*, 2011.

[10] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language Virtualization for Heterogeneous Parallel Computing. Onward!, 2010.

[11] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP, 2011.

[12] D. Clarke, J. Östlund, I. Sergey, and T. Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming*, volume 7850 of *Lecture Notes in Computer Science*, pages 15–58. Springer, 2013.

[13] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In *POPL*, 2006.

[14] D. Gay. *Memory management with explicit regions*. PhD thesis, 1997.

[15] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole. Report on the FX programming language. Technical report, MIT/LCS/TR-531, 1992.

[16] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 28–38, New York, NY, USA, 1986. ACM.

[17] J. Gosling, B. Joy, G. L. Steele, Jr., G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 2013.

[18] J. Hannan. A type-based escape analysis for functional languages. *J. Funct. Program.*, 8(3):239–273, 1998.

[19] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA*, pages 271–285. ACM, 1991.

[20] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In C. S. Ellis, editor, *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, pages 275–288. USENIX, 2002.

[21] O. Kiselyov and C. Shan. Lightweight static capabilities. *Electr. Notes Theor. Comput. Sci.*, 174(7):79–104, 2007.

[22] O. Kiselyov and C. Shan. Lightweight monadic regions. In *Haskell*, pages 1–12. ACM, 2008.

[23] H. Lee, K. J. Brown, A. K. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, 2011.

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

[25] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 333–343, New York, NY, USA, 1995. ACM.

[26] B. Lippmeier. *Type Inference and Optimisation for an Impure World*. PhD thesis, Australian National University, 2010.

[27] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL

'88, pages 47–57, New York, NY, USA, 1988. ACM.

[28] 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.

[29] N. D. Matsakis and F. S. Klock, II. The Rust language. *Ada Lett.*, 34(3):103–104, Oct. 2014.

[30] M. S. Miller. The E language. `http://erights.org/elang/index.html`, 1998.

[31] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991.

[32] J. Moses. The function of function in lisp or why the funarg problem should be called the environment problem. *ACM Sigsam Bulletin*, (15):13–27, 1970.

[33] K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. A type system for borrowing permissions. In *POPL*, pages 557–570. ACM, 2012.

[34] F. Nielson and H. Nielson. Type and effect systems. In E.-R. Olderog and B. Steffen, editors, *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 114–136. Springer Berlin Heidelberg, 1999.

[35] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *ECOOP*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer, 1998.

[36] M. Odersky. Scala - where it came from, where it is going. `http://www.slideshare.net/Odersky/scala-days-san-francisco-45917092`, 2015.

[37] M. Odersky and T. Rompf. Unifying functional and object-oriented programming with scala. *Commun. ACM*, 57(4):76–86, 2014.

[38] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, and M. Püschel. Spiral in scala: towards the systematic construction of generators for performance libraries. In J. Järvi and C. Kästner, editors, *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*, pages 125–134. ACM, 2013.

[39] D. J. 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.

[40] T. Petricek, D. A. Orchard, and A. Mycroft. Coeffects: Unified static analysis of context-dependence. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, pages 385–397, 2013.

[41] T. Petricek, D. A. Orchard, and A. Mycroft. Coeffects: a calculus of context-dependent computation. In J. Jeuring and M. M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 123–135. ACM, 2014.

[42] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 71–84, New York, NY, USA, 1993. ACM.

[43] G. D. Plotkin and M. Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013.

[44] A. Prokopec, P. Bagwell, and T. R. abd Martin Odersky. A generic parallel collection framework. Euro-Par, 2010.

[45] M. Püschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232 –275, feb. 2005.

[46] E. Reed. Patina : A Formalization of the Rust Programming Language. (February):1–37, 2015.

[47] T. Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming*. PhD thesis, EPFL, 2012.

[48] T. Rompf and N. Amin. From F to DOT: Type soundness proofs with definitional interpreters. Technical report, Purdue University, July 2015. `http://arxiv.org/abs/1510.05216`.

[49] T. Rompf and N. Amin. Type soundness for dependent object types (dot). In *OOPSLA*, 2016.

[50] T. Rompf, K. J. Brown, H. Lee, A. K. Sujeeth, M. Jonnalagedda, N. Amin, G. Ofenbeck, A. Stojanov, Y. Klonatos, M. Dashti, C. Koch, M. Püschel, and K. Olukotun. Go meta! A case for generative programming and dsls in performance critical systems. In T. Ball, R. Bodík, S. Krishnamurthi, B. S. Lerner, and G. Morrisett, editors, *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, volume 32 of *LIPIcs*, pages 238–261. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[51] T. Rompf, I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 317–328, New York, NY, USA, 2009. ACM.

[52] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

[53] L. Rytz, N. Amin, and M. Odersky. A flow-insensitive, modular effect system for purity. In W. Dietl, editor, *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs, FTfJP 2013, Montpellier, France, July 1, 2013*, pages 4:1–4:7. ACM, 2013.

[54] L. Rytz, M. Odersky, and P. Haller. Lightweight polymorphic effects. In J. Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 258–282. Springer, 2012.

[55] J. Siek. Type safety in three easy lemmas. `http://siek.blogspot.ch/2013/05/type-safety-in-three-easy-lemmas.html`, 2013.

[56] J. G. Siek, M. M. Vitousek, and J. D. Turner. Effects for funargs. *CoRR*, abs/1201.0023, 2012.

[57] C. Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):11–49, 2000.

[58] A. K. Sujeeth, A. Gibbons, K. J. Brown, H. Lee, T. Rompf, M. Odersky, and K. Olukotun. Forge: Generating a high performance dsl implementation from a declarative specification. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE, 2013.

[59] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning*, ICML, 2011.

[60] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun. Composition and reuse with compiled domain-specific languages. In *European Conference on Object Oriented Programming*, ECOOP, 2013.

[61] K. N. Swadi, W. Taha, O. Kiselyov, and E. Pasalic. A monadic approach for avoiding code duplication when staging memoized functions. In J. Hatcliff and F. Tip, editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2006, Charleston, South Carolina, USA, January 9-10, 2006*, pages 160–169. ACM, 2006.

[62] W. Taha and M. F. Nielsen. Environment classifiers. In A. Aiken and G. Morrisett, editors, *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pages 26–37. ACM, 2003.

[63] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[64] J. Talpin and P. Jouvelot. The type and effect discipline. In *Logic in Computer Science, 1992. LICS '92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 162–173, 1992.

[65] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2:245–271, 6 1992.

[66] É. Tanter. Beyond static and dynamic scope. In J. Noble, editor, *Proceedings of the 5th Symposium on Dynamic Languages, DLS 2009, October 26, 2010, Orlando, Florida, USA*, pages 3–14. ACM, 2009.

[67] R. Tate. The sequential semantics of producer effect systems. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 15–26, New York, NY, USA, 2013. ACM.

[68] T. Terauchi and A. Aiken. Witnessing side-effects. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 105–115, New York, NY, USA, 2005. ACM.

[69] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. Højfeld, and O. P. Sestoft. Programming with regions in the MLKit (revised for version 4.3.0). Technical report, IT University of Copenhagen, January 2006.

[70] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 188–201, New York, NY, USA, 1994. ACM.

[71] P. Wadler. Linear types can change the world! In C. Jones, editor, *Proceedings of the IFIP Working Group 2.2/2.3 Working Conference on Programming Concepts and Methods*. North-Holland, 1990.

[72] P. Wadler. The essence of functional programming. In R. Sethi, editor, *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, pages 1–14. ACM Press, 1992.

[73] P. Wadler. The marriage of effects and monads. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, pages 63–74, New York, NY, USA, 1998. ACM.

[74] J. Weizenbaum. The funarg problem explained. Technical report, MIT, Cambridge, Massachusetts, 1968.

[75] T. Zhao, J. Baker, J. Hunt, J. Noble, and J. Vitek. Implicit ownership types for memory management. *Sci. Comput. Program.*, 71(3):213–241, 2008.