System Capybara

Capture Checking for Ownership and Borrowing

Yichen Xu
yichen.xu@epfl.ch
EPFL
Switzerland

Oliver Bračevac
oliver.bracevac@epfl.ch
EPFL
Switzerland

Martin Odersky
martin.odersky@epfl.ch
EPFL
Switzerland

Abstract

Ownership and borrowing principles are powerful tools for preventing unintended aliasing, enabling safe memory management, data-race-free concurrency, and more. These principles, popularized by Rust, are often seen as too rigid to be integrated into mainstream languages like Scala without disrupting existing codebases.

System Capybara introduces flexible and expressive ownership tracking to Scala by extending capture checking, a recently introduced system for effect and capability tracking. It enriches capture checking with alias prevention guarantees, enabling fine-grained control over exclusive resources such as mutable buffers and file handles—while staying lightweight and flexible.

The talk presents System Capybara in three parts: (1) a brief overview of the problem and background; (2) a quick walkthrough of how capture checking can serve as a foundation for alias tracking; and (3) an introduction to System Capybara's key ideas. This part is interactive, with examples presented in Cavia, a prototype compiler that implements System Capybara in a Scala-like language.

Problem Statement. Rust is well-known for its ownership and borrowing principles [8, 9]. These principles trace their roots to linear type systems [6] and ownership systems [3]. They have been shown to provide safe yet performant memory management, enable aggressive compiler optimizations, and ensure data-race-free concurrency—famously referred to as "fearless concurrency" in Rust. These principles have found applications in more areas, such as safe GPU programming [4].

The ownership and borrowing principles, in their essence, are principles for *alias controlling*. An owned value must have a unique reference, and mutable borrows must not coexist with other aliases in a given scope. Type-systematic alias controlling is gaining increasing research interest in the programming language community [5, 7, 6, 2].

Although Scala is a garbage-collected language, it stands to benefit from similar alias control mechanisms. It will provide static guarantees in a wide range of scenarios concerning aliases, such as safe resource management and data-race freedom. Consider the following example:

Authors' Contact Information: Yichen Xu, yichen.xu@epfl.ch, EPFL, Switzerland; Oliver Bračevac, oliver.bracevac@epfl.ch, EPFL, Switzerland; Martin Odersky, martin.odersky@epfl.ch, EPFL, Switzerland.

```
val f = File("test.txt")
f.write("Hello")
f.close()
println(f.size()) // Runtime error: accessing a closed file
```

An ownership principle will prevent the last line from being type-checked, since the type system would be able to track that closing a file handle *consumes* it, which prevents it from being accessed further. Another example:

```
def matmul(a: Tensor, b: Tensor, out: Tensor): Unit = ...
val t = torch.randn((100, 100))
val t1 = t
matmul(t, t, t1) // Unintended mutation through aliasing
```

The `matmul` function takes two tensors and writes the result of multiplication to the third tensor. The last line causes unintended behavior when executed, since it mutates the tensor `t` when its self-multiplication is being computed. An alias controlling principle should recognize that `t1` is an alias of `t` and reject the last line.

But there is a catch. Rust is frequently criticized for its lack of flexibility. Its stringent ownership principles impose strict restrictions on coding styles. Programmers are typically required to internalize the ownership principles and adopt entirely new mental models to appease the borrow checker. If such a principle were introduced into Scala, it would become massively disruptive to the existing code base, requiring both significant refactoring efforts and a whole new mindset for programmers. This is infeasible for a language that has a highly established ecosystem. Therefore, integrating the proven-useful alias control principles into an established language like Scala remains a significant challenge.

Capture Checking for Alias Tracking. Capture Checking (CC) represents a recent advancement in bringing lightweight and flexible effect tracking to Scala [1]. It elegantly unifies effect and resource tracking with object capabilities. It introduces *capturing types*, which augment regular types with a *capture set*. The capture set of a type overapproximates the set of variables that can be captured by values of this type. For example,

```
val cnt = Counter(0)
val f1 = () => cnt.inc()
```

We assume `Counter` is a class that implements a counter. Under CC, the variable `f1` is assigned the capturing type `(() -> Unit)^{cnt}`. It consists of (1) the shape type `() -> Unit` indicating that the function takes no argument and returns

`Unit`, and (2) the capture set `{cnt}` which indicates that the body of the function *at most* captures the variable `cnt`.

Here lies a key insight: capture checking naturally serves as an *alias tracking* device. The capture set essentially indicates which variables a value may reference. Recall the previous example:

```
val t = torch.randn((100, 100))
val t1 = t
  // : Tensor^{t}
```

With CC, the type of `t1` is `Tensor^{t}`, making it evident that `t1` is *an alias* to `t`.

A Gentle Introduction to System Capybara. System Capybara builds upon this foundation, extending capture checking to support flexible and lightweight alias control in Scala. By leveraging capturing types as an alias tracking mechanism, it enriches capture checking with alias control mechanisms. The following examples will be demonstrated interactively using Cavia, our prototype compiler. The system operates on the fundamental principle that *aliases are permitted when explicitly tracked*, while *hidden aliases must be prevented*. For example, the `par` function takes two operations and runs them in parallel.

```
def par(f: () => Unit, g: () => Unit): Unit =
  // ... Running f and g in parallel ...
```

The type `() => Unit` is a shorthand for `(() -> Unit)^{cap}`. Here, **`cap`** is the *universal capability*, the top capability in CC that subsumes all others. The type indicates that the operation may capture arbitrary capabilities. But the captures of `f` and `g` should be *separate*, since there is no visible alias between them in the signature. In the following code:

```
val cnt = Counter(0)
val f1 = () => cnt.inc()
// : () ->{cnt} Unit
val f2 = () => cnt.dec()
// : () ->{cnt} Unit
val cnt2 = cnt
// : Counter^{cnt}
f1() // ok
f2() // ok
println(cnt2.get) // ok
```

`cnt` is a counter; Functions `f1` and `f2` both reference this counter; and `cnt2` is an alias to `cnt`. They are all allowed to coexist and be used simultaneously, since the fact that they reference or alias the counter `cnt` is evident in their capturing types. However, the following is disallowed:

```
par(f1, f2) // separation error
```

When `f1` and `f2` are passed to `par`, their types are both widened to `(() -> Unit)^{cap}`, *hiding* the fact that they reference the same counter, which should be prevented. Another angle is that when the `par` function is *defined*, there are no visible aliases or overlap between the signatures of the formal parameters `f` and `g` (which are both `() => Unit`). In other words, the `par` function *assumes* `f` and `g` to be separate (which is also intended so that data races can be prevented). Therefore, at the call site of `par`, such separation should be enforced.

Read-Only References. System Capybara supports reasoning about *read-only* references. Two read-only aliases are considered separate. For instance, it is possible to take two read-only references to a tensor and pass them to `par`:

```
val t = torch.randn((100, 100))
val t1: Tensor^ro{t} = t
val t2: Tensor^ro{t} = t
par(() => println(t1.sum), () => println(t2.sum)) // ok
```

`Tensor^ro{t}` is a capturing type qualified at *read-only* mode: it indicates that the tensor can *only* be used to perform read-only operations.

Freshness and Consuming. *Fresh* capabilities carry a global exclusiveness guarantee. Newly-allocated resources like counters are inherently fresh. Moreover, any universal capability returned from a function must be fresh. For instance, the following is disallowed:

```
def mkIncrementer(c: Counter^{cap}): () => Unit =
  () => c.inc()
```

Without this restriction, we would lose crucial aliasing information between the returned closure and the input counter. To make this function well-typed in Capybara, the parameter must be annotated as **`consume`**:

```
def mkIncrementer(consume c: Counter^{cap}): () => Unit =
  () => c.inc()
```

The **`consume`** annotation marks a parameter as consumed by the function. Consuming a value prevents further access to it, making it safe to treat the consumed argument as *fresh*. For instance, the following will be rejected:

```
val cnt = Counter(0)
val f = mkIncrementer(cnt)
println(cnt.get) // error: accessing a consumed value
```

Current Development. System Capybara is actively under development with three key components:

- a formalization with mechanized metatheory in Lean 4,
- Cavia, a prototype compiler implementing the system,
- and a Scala 3 compiler implementation.

This talk introduces the core ideas behind System Capybara and demonstrates its potential through interactive examples. With System Capybara, we are exploring a promising direction for bringing ownership-like guarantees to established languages in a flexible yet expressive way.

Talk Outline.

1. **Problem and Motivation** (4 minutes)
   Overview of ownership and borrowing benefits, and the challenges of integrating them into established languages like Scala.

2. **Capture Checking Foundation** (6 minutes)
   Introduction to capture checking and how it naturally serves as an alias tracking mechanism.
3. **System Capybara Overview** (10 minutes)
   Interactive demonstration of the core principles: tracked aliases, separation enforcement, read-only references, and consuming mechanisms using Cavia.
4. **Discussion and Q&A** (5 minutes)
   Open discussion about the approach, potential applications, and future directions.

References

[1] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondřej Lhoták, and Jonathan Brachthäuser. 2023. Capturing Types. *ACM Trans. Program. Lang. Syst.*, 45, 4, (Nov. 20, 2023), 21:1–21:52. DOI: 10.1145/3618003 (cit. on p. 1).

[2] Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control, 26 pages. DOI: 10.4230/LIPICS.ECOOP.2016.5 (cit. on p. 1).

[3] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Vol. 7850. Dave Clarke, James Noble, and Tobias Wrigstad, (Eds.) Red. by David Hutchison et al. Springer Berlin Heidelberg, Berlin, Heidelberg, 15–58. ISBN: 978-3-642-36945-2 978-3-642-36946-9. DOI: 10.1007/978-3-642-36946-9_3 (cit. on p. 1).

[4] Bastian Köpcke, Sergei Gorlatch, and Michel Steuwer. 2024. Descend: A Safe GPU Systems Programming Language. *Proceedings of the ACM on Programming Languages*, 8, (June 20, 2024), 841–864, PLDI, (June 20, 2024). DOI: 10.1145/3656411 (cit. on p. 1).

[5] Anton Lorenzen, Leo White, Jane Street, Stephen Dolan, Richard A Eisenberg, and Sam Lindley. [n. d.] Oxidizing OCaml with Modal Memory Management (cit. on p. 1).

[6] Danielle Marshall, Michael Vollmer, and Dominic Orchard. 2022. Linearity and Uniqueness: An Entente Cordiale. In *Programming Languages and Systems*. Vol. 13240. Ilya Sergey, (Ed.) Springer International Publishing, Cham, 346–375. ISBN: 978-3-030-99335-1 978-3-030-99336-8. DOI: 10.1007/978-3-030-99336-8_13 (cit. on p. 1).

[7] Mae Milano, Joshua Turcotti, and Andrew C. Myers. 2022. A flexible type system for fearless concurrency. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. ACM, San Diego CA USA, (June 9, 2022), 458–473. ISBN: 978-1-4503-9265-5. DOI: 10.1145/3519939.3523443 (cit. on p. 1).

[8] David J. Pearce. 2021. A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust. *ACM Transactions on Programming Languages and Systems*, 43, 1, (Mar. 2021), 1–73. DOI: 10.1145/3443420 (cit. on p. 1).

[9] Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. 2021. Oxide: The Essence of Rust. (Oct. 19, 2021). arXiv: 1903.00982 [cs]. Retrieved Jan. 23, 2024 from http://arxiv.org/abs/1903.00982. Pre-published (cit. on p. 1).