Rust-Like Borrowing with 2nd-Class Values (Short Paper)

Leo Osvald
Purdue University
USA
losvald@purdue.edu

Tiark Rompf
Purdue University
USA
tiark@purdue.edu

Abstract

The Rust programming language demonstrates that memory safety can be achieved in a practical systems language, based on a sophisticated type system that controls object lifetimes and aliasing through notions of ownership and borrowing. While Scala has traditionally targeted only managed language runtimes, the ScalaNative effort makes Scala a viable low-level language as well. Thus, memory safety becomes an important concern, and the question bears asking what, if anything, Scala can learn from Rust. In addition, Rust's type system can encode forms of protocols, state machines, and session types, which would also be useful for Scala in general. A key challenge is that Rust's typing rules are inherently flow-sensitive, but Scala's type system is not. In this paper, we sketch one possible method of achieving static guarantees similar to Rust with only mild extensions to Scala's type system. Our solution is based on two components: First, the observation that continuation passing style (CPS) or monadic style can transform a flow-sensitive checking problem into a type-checking problem based on scopes. Second, on a previously presented type system extension with second-class values, which we use to model scope-based lifetimes.

CCS Concepts • **Theory of computation → Program constructs**; *Type structures*; • **Software and its engineering → Software safety**; *Patterns*; Control structures;

Keywords Rust, Scala, programming model, safety, memory model, borrowing, second-class

ACM Reference Format:
Leo Osvald and Tiark Rompf. 2017. Rust-Like Borrowing with 2nd-Class Values (Short Paper) . In *Proceedings of 8th ACM SIGPLAN International Scala Symposium (SCALA'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3136000.3136010

ACM ISBN 978-1-4503-5529-2/17/10…$15.00
https://doi.org/10.1145/3136000.3136010

1 Introduction

It is well known that *mutability* in imperative (as well as non-pure functional) languages may lead to obscure bugs, not only in a concurrent setting but also in sequential programs, so long as multiple *aliased* references exist at a program point. Examples of the former are data races, which occur whenever there are two concurrent accesses to the same memory location with at least one being a write, unless both are atomic (which incurs synchronization overhead). A perhaps unusual but important example of the latter is iterator invalidation, which occurs due to concurrent modifications of the traversed data structure through an aliased mutable reference (the other reference is needed for traversal).

To restrict mutability to cases when it is safe, different programming languages take different approaches. Two extreme approaches are to disallow mutability or aliasing whatsoever; these are arguably best witnessed by Haskell, a purely functional language, and R, which copies data whenever aliasing may occur (copy on write). Rust, a recent statically typed language by Mozilla, on the other hand, offers zero-cost abstractions and provides static guarantees that programs are free of data races, null pointer dereferencing, as well as certain bugs present even in sequential programs with garbage collection such as iterator invalidation. Its type system includes a *borrow checker* component that statically checks that the following rules hold:

- one or more references (`&T`) to a resource; and
- no more than one *live* mutable reference (`&mut T`).

A live reference is roughly a reference that can be (safely) dereferenced. Immutable references are always live unless they are unreachable; i.e., their pointed-to values have been moved out due to *ownership* transfer. However, a (reachable) mutable reference can be *reborrowed* for a shorter lifetime, such as through a function call, during which time the reference is *not* live. (For example, it is safe to pass a mutable reference to a function while holding one at the call site.)

1.1 Motivation

Rust's type system is flow-sensitive. Consequently, an expression may have different types depending on control flow; e.g., the following snippet in Rust,

```
let mut x = 5;
let y = &mut x;
*y += 1;
println!("{}", x);
```

fails to compile, producing the following error message:

```
error: cannot borrow 'x' as immutable because it is
       also borrowed as mutable
                  println!("{}", x);
```

This is in contrast to Scala (and most other languages), where types are flow-insensitive. To be more specific, the `x` variable does not have the same "type" throughout the scope in the above snippet, which means programs in Rust (or another flow-sensitive language) are more complicated to reason about in general. The line at fault here is `let y = &mut x`, which restricts the "type" of `x` by disallowing any further accesses to it for the remainder of the scope. (Indeed, if the above snippet is rewritten as follows,

```
let mut x = 5;
{
  let y = &mut x;
  *y += 1;
}
println!("{}", x);
```

it does compile, since limiting the mutable alias of `x` (via `*y`) to an inner scope makes `x` unaliased in the outer scope.)

For our goal of extending Scala with similar static checking capabilities, a direct translation of Rust's typing rules would therefore lead to a quite different, and certainly much more complicated language. Inspired by recent work on object capabilities in Scala [13], we observe that we can remove the dependence on flow sensitivity by introducing auxiliary scopes whenever flow-dependent information changes. This idea naturally leads to expressing programs in continuation passing style (CPS) or monadic style. The example `let y = &mut x` becomes in Scala syntax:

```
  bindMut(x) { y => ... }
```

Observe that the visibility of `y` is based on the `{ y => ... }` scope, thus it suffices to solve a simpler checking problem that is based on scopes. For clarity, we will use explicit monadic syntax throughout this paper, but the transformation can easily be automated and hidden from users [25].

In this paper, we show one way of retrofitting the concepts of borrowing and alias control from Rust into Scala. We describe a general method of adding flow sensitivity to Scala's type system, which is applicable to a broad set of flow-insensitive languages. More precisely, we describe a Scala extension that statically enforces the following as long as the variables are introduced via the appropriate wrappers:

1. lexical scoping of variables;
2. no creation of mutable variables out of other variables;
3. no assignments on immutable variables.

On a high level, we use second-class values from a recent work [23], subtyping and implicit conversions, and macros and virtualization, to enforce 1., 2., and 3., respectively. As noted before, the burden of writing programs in monadic style can be eliminated through macros, like Scala async/await[1], or by using Scala's existing CPS transformation plug-in [25].

1.2 Syntax and Examples

To demonstrate the mechanics of our facilities for enforcing the aforementioned rules 1.–3., consider the following snippet:

```
bindMut(42) { mut =>
  bindImm(mut) { imm => ... }  // error
  bindMut(mut) { mutAlias => ... }  // error
  val mutAlias = mut  // error
  mut.value = 0  // ok
}
```

First, the literal integral value 42 is bound to a mutable variable `mut` for the remainder of the snippet. In the second line, an attempt is made to re-bind the mutable value as immutable, which fails as expected; if our system was to allow it, that would be unsafe because the same location that holds 42 would be *mutably aliased*: writeable through `mut` and readable through `imm`, two variables (aliases in our case). The third line fails for the same but stronger reason. The next line is yet another attempt to work around the type checker; it fails for a less obvious reason that will be described shortly, but intuitively it is because all variables in our system are introduced as second-class (parameters of closures in the continuation passing style (CPS)). (Being parameters, and thus values, their re-assignment is disallowed in Scala by design.) Finally, we provide a means to mutate mutable variables by assigning new values to them via a setter method `value_=`, as demonstrated in the penultimate line.

Conversely, immutable variables can share their pointed-to values, e.g.:

```
bindImm(1) { imm =>
  bindImm(imm) { immAlias => ... }  // ok
  bindMut(imm) { mutAlias => ... }  // error
  imm.value = 0  // error
}
```

In the snippet above, an immutable variable `imm` is initially assigned a 1 in the first line. In the second line, an alias to the same value (1) is created by binding `imm` to a new variable, `immAlias`; this is safe because the shared value (1) cannot be changed due to absence of any mutable variable, hence reading through either variable yields the same value. In the third line, however, this would be invalidated, therefore our system raises a compile-time error. The next line does not type-check either, due to the lack of a hidden but required implicit parameter in the setter method that is only available for mutable variables.

[1]https://github.com/scala/async

2 Design

In order to simplify our implementation, as well as the reasoning behind it, we break our system into two levels: the *library* level, which enforces the rules but still provides escape hatches (similar to the usage of `unsafe` in Rust); and the *meta* level, which enforces that unsafe workarounds are prohibited. Of course, both of these are enforced statically, albeit using different facilities. In the former, we rely on an extended Scala type system that is proven to be sound, and enforce the CPS-style let bindings to make variable bindings explicit. In the latter, we use Scala Macros [6] to disallow certain syntactic patterns when binding to variables, and override the usual behavior of assignments using Scala-Virtualized [19] to confine the aliasing only to occur through the facilities introduced by our library.

2.1 Library-level (core Scala)

For variables of type `T`, we introduce a wrapper type `Var[T,A]`, where `A` is either of following types: `Mut[T]` or `Imm[T]`, depending on whether the binding is mutable (and thus also reassignable) or immutable, as follows:

```
class Mut[T]
class Imm[T]
class Var[T,A](private var v: T) {
  def value = v
  def value_=(v2: T)(implicit ev: A =:= Mut[T]) = v = v2
}
```

In order to prevent mutability due to reassignments, we enable the setter method `value_=` only if type parameter `A` is `Mut[T]`; i.e., if the variable is mutable. (This is done by requiring an implicit evidence that `A` is the same type as `Mut[T]`, which exists in instantiation `Var[T,Mut[T]]`.)

Next, we introduce the `bind*` methods that bind a literal value or an existing variable to a new variable. In case of the immutable binding, it is safe to pass not only a literal value but also an existing immutable variable. However, aliased (shared) variable bindings are permitted only if none of them is mutable. Therefore, we disallow conversion of variables from immutable to mutable by ensuring that access type parameter `A` is invariant and/or types `Mut[T]` and `Imm[T]` are unrelated, which invalidates the subtyping relationship `Var[V,Imm[S]] <: Var[V,Mut[T]]`, for all types `S`,`T` and `V`, and in turn disallows upcasting an immutable variable to a mutable one.

Additionally, we need to prevent creation of shared mutable variables, which we achieve by using second-class values [23]. 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. These rules are statically enforced through our existing compiler plug-in, thereby ensuring that second-class values have stack-based lifetimes. The trick is to introduce mutable variables as second-class but require their sources to be first-class, as follows:

```
def bindImm[T, U](@local r: Var[T,Imm[T]])(
  @local f: Var[T,Imm[T]] -> U) = f(new Var[T,Imm[T]](r.value))
def bindMut[T, U](r: Var[T,Mut[T]])(
  @local f: Var[T,Mut[T]] -> U) = f(r)
```

The `A -> B` denotes a function in which the parameter of type `A` is second-class but the return type is first-class; i.e., `(@local A) => B`, where `@local` annotation denotes a second-class type. More specifically, introducing variables as second-class values restricts their lifetime to the enclosing scope defined by the passed closure, e.g.,

```
bindImm(42) { x =>
  bindMut(0) { y =>
    y.value = x
  }
} // x cannot be returned/stored as a regular (1st-class) value
```

The function parameter must also be second-class to allow the usage of `x` in an inner closure, such as in the line `y.value = x`. (Informally, using free second-class values lifts the closure to second-class, and first-class values can be promoted to second-class values but not vice versa.)

Lastly, we introduce bridge methods to create variables out of literals so that the above snippet actually type-checks. To be as close to Rust as possible, we treat values as immutable by default, and mutable only when required to appease the type checker or explicitly requested. In Scala, this can be done automatically through unambiguous implicit conversions from values of type `T` to variables of type `Var[T,A]`, such that the conversion to mutable variables has less priority. We achieve this by declaring an implicit conversion to a mutable variable in a supertype, which is searched after the corresponding `Var` companion object, as follows:

```
class LowPrioMut
object LowPrioMut {
  implicit def valToMut[T](v: T): Var[T,Mut[T]] =
          new Var[T,Mut[T]](v)                        }
object Var extends LowPrioMut {
  implicit def valToImm[T](v: T): Var[T,Imm[T]] =
          new Var[T,Imm[T]](v)                        }
```

2.2 Meta-level (Scala Macros & Scala-Virtualized)

What remains to enforce that `ref.value` for any variable `ref` is not inadvertently passed to `bindMut` or `bindImm`, which would bypass the above type-checking rules in cases such as the following ones, respectively:

```
bindMut(123) { mut =>
  bindImm(ref.value) { imm => ... /* ouch */  }
}
bindImm("foo") { imm =>
  bindMut(ref.value) { mut => ... /* ouch */ }
}
```

To prevent this, we hide methods `bindImm` and `bindMut`, instead encouraging the usage of `let` and `letMut` macros.

These macros statically check that either another variable or an r-value—such as `123` or `new StringBuilder()`—other than `ref.value`, for any `ref` of type `Var[_,_]`, is passed; otherwise, it raises a compile error. Such a syntactic inspection is performed by a straightforward pattern matching on the AST of the first argument passed to `let(Mut)`. (The second argument is a closure, as in the case of `bind*`.) Similarly, we disallow assignment of non-wrapped values to local variables by overriding `__newVar` and `__assign` in Scala-Virtualized. Hence, none of the following attempts type-check anymore:

```
letMut(123) { mut =>
  letMut(ref.value) { mut => ... }  // error (Var.value as arg.)
  var indirect = mut.value  // error (Var.value in assignment)
  let(indirect) { imm => ... }  // error (not an r-value/Var)
}
```

3 Borrowing

With the above API in place, we can model borrowing; i.e., permit *temporary* aliasing for the duration of a method call (or an inner scope). It suffices to a turn function parameter (or a local variable) into a second-class variable wrapper, e.g.:

```
def doWithBorrowed[T](@local ref: Var[T,Mut[T]]) = ...

bindMut(new MutableObject()) { mut =>
  ...
  doWithBorrowed(mut)
  ...
  { @local val borrowed = mut  // requires @local to type-check
    ...
} }
```

For performance and convenience of a reduced number of changes to the existing functions when all parameters are immutable, we introduce wrappers,

```
def call[T, R](f: T -> R)(@local ref: Var[T,_])
def call[T1,T2, R](f: (T1,T2) -> R)(
  @local ref1: Var[T1,_])(@local ref2: Var[T2,_])
...
```

which unwrap the pointed-to values and pass them as second-class arguments to a function. Pure functions that do not store parameters can have all their parameters annotated as second-class, and our system can statically check that they indeed store no values and thus not create any permanent aliases, which could be unsafe (or unexpected) if the arguments are borrowed through a mutable reference, e.g.:

```
def storeMut(@local sb: StringBuilder,
             @local store: Store): String = {
  store.field = sb  // error (cannot store 2nd-class/borrowed)
  sb.toString
}
bindMut(new Store()) { storeThatLeaksMutable =>
  bindMut(new StringBuilder()) { sb =>
    val s = call(storeMut)(sb)(storeThatLeaksMutable)
    sb.value.append("brakes encapsulation")
    assert(s == storeThatLeaksMutable.field)  // would fail
} }
```

4 Related Work

Stack-bounded Lifetimes and Escaping Strachey [26] publicized the terminology of first- and second-class objects. The issues around stack-implementability of functions in LISP is also known as the funarg problem [20, 29], and conditions for stack implementation of the simply-typed call-by-value lambda calculus have been given by Banerjee and Schmidt [2]. Hannan presented a type-based escape analysis [15], 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 [27] 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 [28].

Unique Ownership and Borrowing Ownership type systems [7, 9] were developed to protect against unintentional aliasing and unexpected side effects in object-oriented programs. Some researchers allow *temporary* aliasing and borrowing [16, 22], which do not require destructive reads, and is thus similar to our approach. Clarke and Wrigstad [8, 30] allow it in the form of borrowing, and statically enforce external uniqueness otherwise. In AliasJava [1], this is done via lent references, which cannot be stored to fields, and thus can safely point to any ownership context. Boyland uses technique called alias burying [4], which is based on static analysis that tracks live aliases. Haller and Odersky [12, 14] model unique and borrowed references in Scala using capabilities, and also support ownership transfer. In contrast, our work exploits the fact that second-class values cannot be stored in a field or returned from a function, and is entirely type-directed. It is similar to generic universe types [11, 21], except that we do not support ownership transfer; however, this is not limiting because our references may be temporarily aliased when they are passed to second-class function parameters during a function call. Overall, our design is similar to LaCasa [13]—their boxes map to our variable wrappers (references), and opening a box is similar to introducing a scope-based dereferenced value in our case—but we also distinguish between immutable and mutable references, like the Pony language [10].

Rust borrowing and ownership semantics is informally explained by its authors [18] and developers [3]. A formalization of its semantics is still an ongoing effort [24], and/or is currently awaiting peer review [17].

5 Conclusions and Future Work

We presented a minimalistic design for statically enforcing Rust-like notion of borrowing and alias control for references, which prevents various bugs in concurrent and sequential settings alike, but without putting a burden of appeasing a flow-sensitive type checking on the programmer as Rust does.

Therefore, our system is both practical and integrates well with the existing context-insensitive type system with local type inference rules, in particular, Scala. Moreover, our approach requires only a minor extension to Scala's (or another context-insensitive) type system—a support for second-class values—and the code is mostly self-contained in this paper.

In the future, we plan to further investigate how to precisely model ownership (transfer) and lifetimes of bound values, perhaps using state-of-the-art capability-based approaches [5, 14] in conjunction with subtyping rules for a generalization of second-class values (i.e., a privilege lattice) [23]. Another promising direction is static resource management using ownership tracking, which would give rise to ScalaNative and Off-heap libraries, to avoid unnecessary performance overhead and latencies due to JVM garbage collection in the light of some previous approaches [31].

Acknowledgments

This research was supported by the National Science Foundation through awards No. 1553471 and No. 1564207.

References

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 311–330, New York, NY, USA, 2002. ACM.

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

[3] A. Beingessner and S. Klabnik. The Rustonomicon: The Dark Arts of Advanced and Unsafe Rust Programming. https://doc.rust-lang.org/nomicon/references.html, 2016.

[4] J. Boyland. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience*, 31(6):533–553, 2001.

[5] J. Boyland, J. Noble, and W. Retert. Capabilities for Sharing. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming: 15th European Conference Budapest, Hungary, June 18–22, 2001 Proceedings*, pages 2–27. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[6] E. Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 3:1—3:10, New York, NY, USA, 2013. ACM.

[7] 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.

[8] D. Clarke and T. Wrigstad. *External Uniqueness Is Unique Enough*, pages 176–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[9] D. G. Clarke, J. M. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 48–64, New York, NY, USA, 1998. ACM.

[10] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil. Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2015, pages 1–12, New York, NY, USA, 2015. ACM.

[11] W. Dietl, S. Drossopoulou, and P. Müller. *Generic Universe Types*, pages 28–53. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[12] P. Haller. *Isolated Actors for Race-Free Concurrent Programming*. PhD thesis, 2010.

[13] P. Haller and A. Loiko. LaCasa: Lightweight Affinity and Object Capabilities in Scala. *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, (Section 2):272–291, 2016.

[14] P. Haller and M. Odersky. *Capabilities for Uniqueness and Borrowing*, pages 354–378. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

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

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

[17] R. Jung and J.-h. Jourdan. RustBelt: Securing the Foundations of the Rust Programming Language. https://people.mpi-sws.org/~dreyer/papers/rustbelt/paper.pdf (under peer review), 2017.

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

[19] A. Moors, T. Rompf, P. Haller, and M. Odersky. Scala-virtualized. *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, pages 117–120, 2012.

[20] 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.

[21] P. Müller and A. Rudich. Ownership Transfer in Universe Types. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, volume 42 of *OOPSLA '07*, pages 461–478, New York, NY, USA, 2007. ACM.

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

[23] L. Osvald, G. Essertel, X. Wu, L. I. G. Alayón, and T. Rompf. Gentrification gone too far? affordable 2nd-class values for fun and (co-)effect. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 234–251, New York, NY, USA, 2016. ACM.

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

[25] 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.

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

[27] 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.

[28] 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.

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

[30] T. Wrigstad. *Ownership-Based Alias Management.* PhD thesis, 2006.

[31] 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.