Mutable Value Semantics Through a Runtime-Enforced Framework in Scala

Hamza Remmal
École Polytechnique Fédérale de Lausanne, Switzerland

Abstract

This work presents a minimal, runtime-based, framework for enforcing mutable value semantics in Scala. Unlike languages with MVS foundations, Scala lacks support for ownership and borrowing, often leading to aliasing issues and unintended mutations. This library addresses these challenges by defining a single abstraction that enforces MVS semantics entirely at runtime via precise assertions. We showcase the framework's guarantees with a practical example, a mutable linked list.

2012 ACM Subject Classification Software and its engineering → Language features; Software and its engineering → Semantics; Software and its engineering → Frameworks

Keywords and phrases Mutable Value Semantics, Value Independence, Runtime Verification

Digital Object Identifier 10.4230/OASIcs.Programming.2025.26

Category Extended Abstract

1 Introduction

Mutable value semantics [3] combines the safety of value semantics with the efficiency of controlled mutations through strict ownership guarantees. While languages like Hylo have native support for MVS via type-level ownership [1] and borrowing, Scala lacks built-in mechanisms for enforcing such constraints, resulting in unintended aliasing and subtle bugs [2]. This work presents a minimal, purely runtime-based, Scala library to explicitly enforce MVS principles, including storage initialisation, borrowing, and safe move semantics. With a single abstraction, `Storage[T]`, and no changes to the compiler, the framework provides clear, runtime-checked, guarantees for value independence and safe mutability. We validate our framework using practical examples like a mutable singly linked list.

2 Design of the Runtime Framework

The framework introduces a single abstraction, `Storage[T]`, to provide the fundamental operations required to ensure mutable value semantics.

```
final class Storage[T] private :
  def store(value: T): this.type = /* ... */ // initialise the storage with a value
  def load: T = /* ... */ // fetch the value from the storage
  def move: T = /* ... */ // consume the value from the storage
  def deinit: this.type = /* ... */ // deinitialise the value from the storage
  // scoped, exclusive, borrow of the value
  def borrow[R](action: Storage[T] => R): R = /* ... */
object Storage:
  def alloc[T]: Storage[T] = /* ... */ // allocate an uninitialised storage
  def alloc[T](value: T): Storage[T] = /* ... */ // allocate an initialised storage
```

Each instance represents an independent, initialisable, and runtime managed cell that holds a single value. It explicitly manages the semantics of ownership, lifecycle, and borrowing to guarantee value independence by enforcing the following key properties at runtime:

- **Single Initialisation:** A `Storage[T]` instance can be initialised exactly once using `store`. Subsequent attempts to initialise the storage without explicitly clearing the wrapped value will result in a runtime error;
- **Exclusive Borrowing:** `Storage[T]` supports a scoped borrow operation which grants exclusive mutable access to its content, ensuring no simultaneous conflicting accesses;
- **Move Semantics:** values can be explicitly moved out ensuring a safe transfer of ownership;
- **Explicit De-initialisation:** values can be explicitly cleared, providing deterministic lifecycle management

3 Use-Case Example: Mutable Singly Linked List

To demonstrate the effectiveness of the framework, an in-place, mutable, singly linked list was implemented.

```
enum List[+T]:
  private case Nil extends List[Nothing]
  private case Cons[T](value: Storage[T], next: Storage[List[T]]) extends List[T]
object List:
  def empty[T]: Storage[List[T]] = Storage.alloc(List.Nil)
  extension [T](list: Storage[List[T]])
    def prepend(value: Storage[T]): Unit =
      list.borrow: list =>
        // in-place and controlled mutation of the list
        list.store(List.Cons(Storage.alloc(value.move), Storage.alloc(list.move)))
    def reverse: Unit =
      def reverse(xs: Storage[List[T]], ys: Storage[List[T]]): Unit =
        xs.borrow: xs => // require an exclusive borrow of xs
          ys.borrow: ys => // require an exclusive borrow of xs
            xs.load match
              case Nil => swap(xs, ys)
              case Cons[T @unchecked](_, next) =>
                // swap verifies the exclusive borrow of next
                swap(next, ys); swap(xs, ys); reverse(xs, ys)
      list.borrow: list =>
        reverse(list, List.empty)
    end reverse
```

4 Conclusion

We presented a minimal, runtime-enforced, framework for mutable value semantics in Scala. Its design ensures value independence via explicit control over ownership, borrowing, and lifecycle management. Centred around a single abstraction, `Storage[T]`, the framework prevents aliasing and unsafe mutations through effective runtime checks. This approach offers a practical alternative to compile-time systems, allowing users to model value semantics clearly and safely within Scala's existing type system. While this solution does not eliminate all classes of errors at compile-time, it provides an immediate base for reasoning about value independence in Scala programs. Additionally, the framework can serve as a target language for compilers to automatically instrument programs with runtime enforcement of ownership and borrowing constraints, making it suitable for prototyping.

References

1 Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership Types: A Survey. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 15–58. Springer, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-36946-9_3.

2 Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. Immutability. In *Aliasing in Object-Oriented Programming*, volume 7850 of *LNCS*, pages 233–269. Springer-Verlag, April 2013. doi:10.1007/978-3-642-36946-9_9.

3 Dimitri Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, and Brennan Saeta. Native Implementation of Mutable Value Semantics, June 2021. arXiv:2106.12678. doi:10.48550/arXiv.2106.12678.