A Practical Type System for Safe Aliasing

Dimitri Racordon
University of Geneva
Centre Universitaire d'Informatique
dimitri.racordon@unige.ch

Didier Buchs
University of Geneva
Centre Universitaire d'Informatique
didier.buchs@unige.ch

Abstract

Aliasing is a vital concept of programming, but it comes with a plethora of challenging issues, such as the problems related to race safety. This has motivated years of research, and promising solutions such as ownership or linear types have found their way into modern programming languages. Unfortunately, most current approaches are restrictive. In particular, they often enforce a single-writer constraint, which prohibits the creation of mutable self-referential structures. While this constraint is often indispensable in the context of preemptive multithreading, it can be worked around in the case of single threaded programs. With the recent resurgence of cooperative multitasking, where processes voluntarily share control over a single execution thread, this appears to be interesting trade-off. In this paper, we propose a type system that relaxes the usual single-writer constraint for single threaded programs, without sacrificing race safety properties. We present it in the form of a simple reference-based language, for which we provide a formal semantics, as well as an interpreter.

ACM Reference format:
Dimitri Racordon and Didier Buchs. 2022. A Practical Type System for Safe Aliasing. In *Proceedings of International Conference on Software Language Engineering, Boston, USA, November 5–6, 2018 (SLE2018),* 14 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Aliasing, which denotes situations where multiple names (or aliases) actually refer to the same object or memory location, is a vital programming concept. Some form of aliasing is very often indispensable in realistic programming languages: not only does it allow to save space and time by avoiding unnecessary data duplication, it is also paramount to efficiently represent shared and self-referential structures.

SLE2018, November 5–6, 2018, Boston, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
1: function SIEVE(inout arr)
2:     for x in arr do
3:         for i := 0 to arr.length do
4:             if (arr[i] ≠ x) and (arr[i] mod x = 0) then
5:                 arr.remove(i)
6:             end if
7:         end for
8:     end for
9: end function
```

Figure 1. A poorly designed implementation of the sieve of Eratosthenes. Because the array `arr` is mutated while iterated over, the program is likely to behave unexpectedly.

Unfortunately, these benefits come at the price of concurrent mutation[1]. An object or memory location may be mutated (i.e. changed) without other references being aware of that mutation, which in turn may violate invariants, cause unexpected behavior or even crash the program altogether. One well known situation highlighting this problem usually occurs when mutating a structure, while iterating over it [33]. Consider for example the implementation of the algorithm of the sieve of Eratosthenes presented in Figure 1. Depending on the iteration semantics of the language in which we would write this algorithm, we could get very different results. For instance, Python will raise an "out-of-bound" exception at line 4, because the loop at line 3 will generate indices past the end of the array if a single value is removed. Java will do a little better, since its collections are programmed to detect that problem at runtime [37]. Therefore, a `ConcurrentModificationException` will be thrown as soon as it reaches line 5. Swift will always produce correct results, because its for-each style iterators keep internal copies of the collection over which they iterate. As a result, the values `x` will take at line 2 will actually be picked from a different array than that from which non-prime numbers are removed. But maybe more worrisome are semantics that could silently yield incorrect results. This is the case of C++, whose vector iterators are simply pointers to contiguous memory. Hence, removing a value with an index preceding that of the iterator invalidates the latter. We argue that none of these implementations are satisfactory, and that the problems we highlight should be detected statically.

[1] Note that here we do not limit the term "*concurrent*" to the context of multithreaded applications. Instead, we consider that any pair of references to the same object are concurrent if their lifetimes overlap.

It is interesting to note that aliasing is only harmful when mutation is involved. Indeed, even in a traditional multi-threaded environment, concurrent reads to the same object or memory location are harmless. From that observation, it is easy to see how guarantees on the immutability of shared objects can offer an invaluable help to determine the correctness of a program. Most modern mainstream programming languages acknowledge this observation, and propose built-in mechanisms to declare and/or ensure some level of immutability. For instance, Swift statically ensures the transitive immutability of its structure instances, unless explicitly declared otherwise. Since ECMAScript 2015, JavaScript lets references be declared so that their reassignment is disallowed. Worth mentioning are also the numerous tools and libraries which provide support for better immutability handling in languages that do not support it natively [42, 45].

Even though this evolution is very positive, there is still room for improvement. Mechanisms to ensure immutability (and semantics thereof) can greatly vary from language to language, and are often insufficient. Immutability constraints are too often offered as an opt-in, therefore not giving a strong incentive for programmers to adopt it. Indeed, in sectors with very short deadlines such as web and mobile application development, bypassing good practices in favor of faster development cycles is not uncommon [22]. Paradoxically, languages that advocate for a very constrained built-in immutability model, such as Rust[27] or Cyclone[40], have difficulty seducing mainstream developers, because of the complexity they involve in implementing otherwise simple patterns [20]. The fact is, safe mutation of aliased objects is inherently complex to handle, especially with respect to concurrency. The existence of a single mutable reference can threaten a program's correctness entirely. Yet assuming all aliased objects to become and remain immutable after a certain point is cumbersome at best and impractical at worst [26]. For instance, modeling a graph-like structure in Rust is known to be a non-trivial problem [7], whereas it is a school example in mainstream programming languages such as C or Java. We therefore can see a clear trade-off drawing itself between defensive approaches that barely or completely ban mutation of aliased objects, and more relaxed ones that focus on less invasive mechanisms, but push the burden of preventing non-obvious problems onto the developer.

Another interesting trend is the resurgence of *cooperative multitasking* [28], also known as non-preemptive multitasking. In this concurrency paradigm, processes voluntarily transfer control between each other, whereas in the more widespread preemptive alternative, this task is left to the operating system. The advantage is that context switching points are predictable, making reasoning about the control flow of a program easier. In particular, one can rely on the fact that a given portion of code will run uninterrupted, eliminating the possibility of another process violating invariants. The reader may remark this concept is reminiscent of transactional memory [16]. Cooperative multitasking enables the design of concurrent programs on single threaded environments, which not only can prove beneficial performance-wise [24], but also significantly reduces the need for locks and other synchronization mechanisms, and opens the door for more relaxed immutability models.

This paper proposes a type system that leverages the locality offered by cooperation over a single threaded (yet possibly concurrent) environment, based on the observation that most existing approaches are overly defensive in that context. Our goal is to relax the usual single-writer constraint, so as to allow mutable self-referential data structures (e.g. doubly linked lists) to be represented. This is not possible in more constrained type systems, such as that of ("safe"[2]) Rust for instance, without resorting to elaborate workarounds or bypassing the type checker altogether. Nevertheless, our type system does not sacrifice other usual guarantees on immutability. Objects referenced by immutable references are guaranteed to remain immutable for the lifetime of said references. We illustrate our approach in the form of a simple programming language which we call SafeScript, and present various examples to highlight interesting features. An implementation of a compiler for SafeScript, as well as various example programs showcasing the type system, are available at https://github.com/kyouko-taiga/SafeScript.

Outline The remainder of this paper is organized as follows: Section 2, presents background on cooperative multitasking (through the lens of JavaScript) and different forms of immutability, before discussing related work in Section 3. An informal introduction to SafeScript's type system is given in Section 4, which is then formalized in Section 5 and 6. Finally, Section 7 concludes.

2 Preliminaries

We now introduce the concepts we will discuss throughout the remainder of this paper.

2.1 Cooperative Multitasking

In this section, we discuss the notion of cooperative multitasking through the lens of an inherently cooperative language: Javascript. In this language, all variables are references (i.e. pointers) to heap-allocated memory blocks, which are automatically garbage collected [43]. In the remainder of this paper, we will use the term *object* to denote the memory occupied by some value (e.g. a number), and *reference* to denote an alias (or name) that refers to an object. For instance, in the statement `let x = "Hello, World!"`, `x` is a reference on the memory location the object `"Hello, World!"` occupies.

[2] https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html

```
async function load() {
  const res = await fetch("/img.png")
  const data = await res.json()
  return data
}
```

Listing 1. A coroutine loading resources (JavaScript). Notice the use of the `async` and `await` keywords. The former declares an asynchronous function (i.e. a coroutine), while the latter indicates that the coroutine yields control while other asynchronous operations are being performed.

As it initially targeted the development of user interfaces, JavaScript is well suited to event-driven programming. Its concurrency model is based on an event loop which dispatches messages – for instance events – to callback functions whenever are made available to it. These callbacks then run uninterrupted until completion, sequentially. Except for a few legacy primitives (whose use is discouraged in modern JavaScript), all operations are non-blocking. Instead, cooperation between said callbacks is advocated. In a nutshell, cooperative functions (a.k.a. *coroutines* [19]) should *voluntarily* suspend their execution while waiting for the resource they need (e.g. some data on a socket) to be available. Their execution context – in other words, the state of their local variables – is preserved during these suspensions, so they can be resumed from the exact point they suspended. It is interesting to remark that cooperative multitasking actually bears a lot of similarities with from its preemptive alternative. Both concurrency models rely on interleaving, with the difference that context switching points are made explicit in the former, while they may happen anywhere in the latter. Listing 1 illustrates a coroutine in JavaScript that loads a resource from a distant server.

We distinguish between two kinds of coroutine implementations: *stackful* coroutines, which can be suspended from within nested function calls, and *stackless* coroutines that cannot. Despite their differences, these two implementations have the same level of expressiveness [28], and therefore we will focus on the latter, as it is simpler to implement. Indeed, a stackless coroutine is, at its core, nothing more than a regular function with an environment that stores its execution context, and the point from where it should run the next time it is resumed. Hence, they can be emulated on the top of more primitive constructions [35]. Other distinctions exists between coroutines implementations, but a comprehensive discussion extends the scope of this paper. A survey is proposed in [28].

2.2 Immutability Semantics

The term "*immutability*" can represent vastly different concepts, depending on the programming language. We briefly introduce these concepts, as well as the terms we will use to distinguish between them in the remainder of this paper. A more elaborate discussion is proposed in [32].

```
const foo = 0
foo = 2     // Error: 'foo' is constant
const bar = {}
bar.baz = 2 // OK
```

Listing 2. Reassignability vs. object immutability (JavaScript). As the keyword `const` declares non-reassignable references, the second line triggers an error, because it attempts to *reassign* the variable `foo` to a new object. However, as it is not paired with a mechanism to enforce object immutability, the fourth line is perfectly legal. The object is not reassigned, even if its value is mutated.

Non-Reassignability (Non-)reassignability is a property of references, indicating whether or not they may be reassigned to point to another object, after they have been assigned once. Preventing reassignability is usually considered a best practice, as it can help avoiding obvious programming mistakes. However, it does not pose any restriction on the *value* of the object being referred to by a non-reassignable alias.

Object Immutability Object immutability is a restriction on the *value* of an object, or more precisely on the bits stored at a memory location that consitute it. We can distinguish two types of object immutability: *shallow immutability*, which only prevents the mutation of the bits an object is made of, and *deep immutability*, which also applies to the other objects that constitute it. In other words, the latter can be understood as the transitive application of the former to each field of an object.

Reference Immutability Reference immutability is a property of references, indicating whether or not the object (or memory location) it refers to might be mutated *through* it. In other words, an immutable reference is an alias through which the object's mutation is not allowed. Just as object immutability, reference immutability can be shallow or deep, that is restricted to the fields of an object, or transitively applied to all values whose path contains a deeply immutable reference.

These three notions are orthogonal to each other. Interestingly, reassignability seems to be more widespread than the other two in mainstream programming languages. This can prove problematic for unexperienced developers, as it may often be confused with object immutability, as showcased in Listing 2.

3 Related Work

There is a great body of work dedicated to the subject of aliasing, which materialized into several branches. One that

is central to our approach is the notion of *type permissions* (a.k.a capabilities). Originally proposed as a generalization of reference (or pointer) annotations [4], various type systems have been proposed to address a broad spectrum of programming aspects, including uniqueness and immutability [30], encapsulation [5], safe concurrency [23] or even memory management [13].

Directly linked is the notion of ownership [29]. Objects are associated with one [10] or several [8] owners, which control what part of their owned domain is exposed to other components [36], and under what policy [31]. Two main interpretations are commonly offered: *owners-as-dominators*, in which all read and write accesses to an object must go through its owner(s), and *owners-as-modifiers*, where only write accesses are restricted. While we do not explicitly rely on a notion of ownership — that is, objects in our type system do not have an *owner* — our approach resembles an owners-as-modifiers system, where owners may revoke permissions on the objects they collectively own between each other.

Other ownership schemes have been proposed to address different concerns, such as *owners-as-locks* [14], to prevent concurrent accesses in the context of multithreaded applications. Just like our own approach, *owners-as-locks* may temporary freeze mutable references, so as to avoid concurrent mutations. Multiple mutable references are prohibited, but mutable self-referential structures are expressible, as long as they expose a single external mutable reference [9]. Our approach relaxes on the single mutable reference constraint, as we place ourselves in the context of single threaded applications, where concurrency is carried out by the means of cooperative multitasking.

Ownership types are often associated with programming language that undergo static compilation (e.g. [20, 27, 42]). If some works have been proposed to include notions of immutability [1] and uniqueness [15] in dynamic languages, the general issue is that runtime checks are hard and/or costly to enforce. Although we present our approach in the context of an interpreted scripting language, we do not suffer this problem, as we perform our analysis statically before the source is processed by an interpreter. Interestingly, this has became a widespread technique to extends small and simple languages with more elaborate concepts (e.g. [2, 34]).

Less tightly connected to our work are other approaches that leverage single-threading to perform static checks on cooperative programs. A few calculi have been proposed to support model checking (e.g. [25, 39]). Indeed, as context switching points are known and vastly less frequent than in a preemptive setting, the size of the state space of a cooperative program is far more manageable, enabling the use of classic model checking and/or testing techniques to ensure safety properties [6].

```
1 class Vector2D {
2   let x: mut = 0
3   let y: mut = 0
4 }
```

Listing 3. Class definition (SafeScript).

4 SafeScript

In this section, we present SafeScript, the language we will use to showcase our type system. Its specificity is that it associates objects with permissions, and comes with first-class mechanisms to transfer or borrow them. For spatial reasons, we will gloss over the specifics of its syntax, and rather focus on its semantics, with a particular emphasis on the the concepts related to aliasing safety.

4.1 Types

SafeScript is an reference-based, imperative programming language. The language features three built-in types, namely `number`, `string`, and `boolean`, as well as the possibility to declare user defined types by the means of class declarations. Listing 3 gives an example of such declaration. The meaning of the `let` and `mut` keywords will be explained in the next section.

4.2 Non-Reassignability and Object Immutability

SafeScript's variable declaration syntax is:

```
(let|var) name: [cst|mut] [= expression]
```

The keywords `let` and `var` declare non-reassignable and reassignable references, respectively. A non-reassignable reference must be bound to an object at its creation, since it cannot be reassigned later. The qualifiers `cst` and `mut` annotate a variable declaration to specify whether the declared reference is *read-only* or *read-write*. This is a departure from most reference-based languages, where all variables are read-write references. In the absence of a qualifier, the declaration is assumed to be read-only. For instance `var foo = Vector(1, 2)` declares a reassignable alias `foo`, but that is not allowed to mutate the object it refers to. Hence, the statement `foo.x = 3` is illegal here.

The mutability of an object is decided at its creation and depends on the first reference it is bound to. If the reference is read-only, then the object is immutable and shall remain so until its deallocation. If it is read-write, then the object is mutable, but may be demoted to immutable later on. SafeScript always enforces deep immutability (Section 2.2). The rationale behind this design choice is that shallow immutability implicitly creates additional aliases, which in turn may lead to incorrect assumptions on the absence of relationships between two objects. An example is given in Listing 4. Although it is not the case in our current implementation, performance issues can usually be addressed with *copy-on-write* semantics[41].

```
1  const productA = {
2    manufacturer: {
3      name: "Thunder LTD",
4      address: "1452 Company blvd",
5    },
6    category: "appliance",
7  }
8  const productB = { ...productA }
9  productB.manufacturer.name = "Spark SA"
10 productB.category = "food"
```

Listing 4. Example of implicit aliasing due to a shallow copy (JavaScript). The expression `{ ...productA }` at line 8 is called *object spreading* [38], and is the idiomatic way to clone an object in JavaScript. However, as it performs a shallow copy, the manufacturer of `productB` is mutated at line 9, while its category is left intact at line 10.

4.3 Binding Semantics

Almost all imperative languages feature an assignment operator, whose role is to bind values to variables, or in our case objects to references. Before concerns for safer aliasing management reached mainstream programming languages, there used to be generally only one semantics associated with that operator. Therefore, each assignment statement would read as "*bind the value on the right to the name on the left*". Even in a language like C++ which features both value and pointer types, this semantics remains mostly valid because the language requires explicit referencing and dereferencing. Hence, the C++ statement "`a = *b`" can be read as "*bind the value pointed by the name* `b` *to the name* `a`". Featuring a single assignment operator seems to have become the norm, yet modern programming languages no longer have a single assignment semantics. Take Listing 5, written in Swift. In this example, while used in seemingly identical contexts, the binding operator (i.e. "=") has two different semantics. Similar issues can be observed in languages that treat primitive types differently, such as Python or Java.

We assert that overloading the semantics of a single assignment operator is harmful to software development, especially when determining which semantics will be applied for a particular statement requires an extensive knowledge of the language. Instead, we advocate for different operators, with a clear and unambiguous semantics. Following this idea, SafeScript features two assignment operators: a *copy operator* "=" and a *borrow operator* "&-". The former copies the object on referred by the reference on its right, while the latter copies the reference itself. The term *borrow* captures the idea that a reference uses the same access as another one on a particular object. Therefore, the example of Listing 5 can be rewritten with a completely unambiguous syntax in SafeScript, as shown in Listing 6.

```
1  struct Student { var name: String }
2  var x1 = Student(name: "Jane")
3  var x2 = x1
4  x2.name = "Ann"
5  print(x1.name) // Prints "Jane"
6
7  class Teacher { var name: String }
8  var y1 = Teacher(name: "Jane")
9  var y2 = y1
10 y2.name = "Ann"
11 print(y1.name) // Prints "Ann"
```

Listing 5. Example of overloaded assignment semantics (Swift). The assignment of line 2 is a copy, while that of line 9 is a *reference* copy, the reason being that Swift treats value (struct) and reference (class) types differently.

```
1  class Student { let name: mut }
2  let x1: mut = new Student(name = "Jane")
3  let x2: mut = x1
4  x2.name = "Ann"
5  console.log(x1.name) // Prints "Jane"
6
7  let x3: mut &- x1
8  x3.name = "Ann"
9  console.log(x1.name) // Prints "Ann"
```

Listing 6. Unambiguous assignment semantics (SafeScript).

4.4 Permission Borrowing

So far we have carefully avoided the subject of temporality and discussed immutability as a permanent state. Of course, this is unrealistic, since objects should at least be mutable during their initialization. The classic solution is to consider an object immutable (deeply or shallowly) only once said initialization is complete, hence leaving the constructor free to mutate the object's fields [17]. The problem with this approach is that it does not allow for an object to *become* immutable at any other point of its lifecycle. Other initialization schemes, such as multi-phase initialization [12], become therefore impossible. Another more pernicious problem can arise when a read-only reference incorrectly assumes immutability of the object it refers to, as it can lead to undetected invariant violations, as outlined in Section 2.2. In a language such as JavaScript where most errors are silent, these kinds of problems often prove very difficult to debug, and can have far reaching consequences. An example is given in Listing 7.

Type permissions offer one solution to tackle this issue. A particular reference enjoys privileged permissions to the object it refers to, such as exclusive read and/or write access, while other references to the same object do not. The latter however may *borrow* privileges [30] for a limited time. Permission borrowing allows us to safely express more complex

```
1 const v = [ 3, 7, 2 ]
2 doSomething(v)
3 console.log(v.reduce(add, 0) / v.length)
```

Listing 7. Example of invariant violation (JavaScript). Since there is no immutability constraint on the object bound to `v`, the function `doSomething` is free to mutate the array, and hence could potentially empty it. That would result in an unexpected result (in this case `NaN` for *Not a Number*) at line 3, which in could easily go undetected and cause problems further in the program.

patterns, such as multi-phase initialization [26] and unicity of mutable references [18].

A particularity of SafeScript is that permissions are associated with objects *and* references. The intersection of both provides the actual permissions a reference enjoys on a particular object. The language features only two permissions: *read-only* and *read-write*. By default, references receive the read-only permission when they are declared. They obtain the read-write permission if and only if they are declared with the `mut` type qualifier. At their creation, objects inherit the permission of the reference they are bound to. Read-only and read-write permissions are borrowed by the means of the *borrow operator* "`&-`". So as to avoid any additional annotation, we use the type on the left side of the operator to determine which permission is requested. In other words, if a mutable reference is placed on the left side of a borrow operator, the statement reads as "*borrow a read-write permission*". Conversely, if an immutable reference is placed on the left, the statement reads as "*borrow a read-only permission*". We call a reference borrowing a read-only permission an *immutable borrowed reference*, and a reference borrowing a read-write permission a *mutable borrowed reference*. We use the phrase *borrowed references* to designate immutable and mutable borrowed reference collectively.

SafeScript guarantees the absence of data races by enforcing the following invariant: *A reference has either n read-only borrows and 0 read-write borrows, or 0 read-only borrows and m read-write borrows at any given time.* It ensures that an object cannot be referred to by both mutable and immutable references at the same time, which may be seen as a sort of *contract* between an immutable borrow and the reference they borrow from. Namely, it states that the object pointed by an immutable reference is guaranteed to remain immutable (or frozen) during the entire lifetime of the said reference, even if it borrowed from a mutable one. Incidentally, the implementation of the sieve of Eratosthenes we presented in the introduction (in Figure 1) cannot fail under this invariant. Either the iterator created at line 2 is implemented holding an immutable borrowed reference on `array`, in which case line 5 becomes illegal, or it is implemented creating a copy of the array, the same way Swift does.

```
1 var a: mut = 1
2 var b: cst = 2
3 {
4   var c: cst &- a
5 }
6 var d: mut &- a
7 var e: cst &- d // Illegal
```

Listing 8. Example of (re)borrowed reference (SafeScript).

line	a	b	c	d	e	*a	*b
1	r/w, r/o	·	·	·	·	r/w, r/o	·
2	r/w, r/o	r/o	·	·	·	r/w, r/o	r/o
4	r/o	r/o	r/o	·	·	r/o	r/o
5	r/w, r/o	r/o	·	·	·	r/w, r/o	r/o
6	r/w	r/o	·	r/w	·	r/w	r/o

Table 1. Permissions from Listing 8.

It is easy to see how multiple immutable references are harmless. But notice that it does not restrict multiple mutable references either. This makes sense in the context of cooperation over a single thread, where write accesses cannot happen "concurrently", as long as the reference remains local to the function (or coroutine) that uses it. We describe the mechanism to preserve this guarantee below.

Example 4.1 (Borrowing)**.** Listing 8 presents an example of reference borrowing, and Table 1 shows how the *effective* permissions associated with the references and objects are updated at each line. We abbreviate read-write (resp. read-only) with *r/w* (resp. *r/o*), and we use the notation `*a` to designate "*the object bound to a*". Since `a` is declared mutable at line 1, the reference and its bound value receive both read-write and read-only permissions. Conversely, because `b` is declared immutable, it only receives the read-only permission, and so does the object it is bound to. `c` borrows an immutable permission on `*a` at line 4, which strips the object from its read-write permission, effectively turning `a` into an immutable reference. When `c` goes out of scope at line 5, the read-write permission of `*a` is restored, turning `a` back to a mutable reference. `d` borrows a mutable permission on `*a` at line 6, which strips the object from its read-only permission. Line 7 is illegal, since `d` is a mutable borrow and hence does not hold a read-only permission. Note that, due to this borrow, attempting to borrow `a` immutable at line 7 would also have been illegal.

Notice that SafeScript allows borrows to be performed on borrowed references, also known as *reborrowing*. Because borrowing does not change the permissions of the reference we borrow from, reborrowing does in fact work the exact same way as borrowing.

5 Operational Semantics

In this section, we formalize the operational semantics of SafeScript. For the sake of conciseness, we use only a minimal

subset of SafeScript, and focus on the permission borrowing mechanisms. These are sufficient to prescribe safety guarantees with respect to aliasing.

5.1 Notation

We write $dom(f)$ the subset $A' \subseteq A$ for which f is defined. We write $f[a \mapsto a']$ the function that returns a' for a and $f(x)$ for any other argument. More formally, $\forall x \in A, x = a \implies f[a \mapsto a'](x) = a'$ and $\forall x \in A, x \neq a \implies f[a \mapsto a'](x) = f(x)$. For instance, if $f(0) = 1$ and $f(1) = 2$, then $f[0 \mapsto 3](0) = 3$ and $f[0 \mapsto 3](1) = 2$. We abbreviate $f[a \mapsto c][b \mapsto c]$ with $f[a, b \mapsto c]$.

We use the usual the horizontal bar notation from [21] to denote sequences. We write $|\overline{x}|$ to denote the length of the sequence $\overline{x}$, such that $|\overline{x}| = n \Leftrightarrow \overline{x} = x_1, \ldots, x_n$. We write $\overline{x}_i$ the i-th element of $\overline{x}$. Let $\overline{x}$ be an non-empty sequence, we write $x_h|\overline{x_t}$ the split of $\overline{x}$ into a head $x_h = x_1$ and a possibly empty tail $\overline{x_t} = x_2, \ldots, x_n$, while $\overline{x} + \overline{y}$ denotes the concatenation of $\overline{x}$ and $\overline{y}$, i.e. $x_1, \ldots x_{|\overline{x}|}, y_1, \ldots, y_{|\overline{y}|}$. Finally, when the context allows it, we overload $\varnothing$ to represent the empty sequence.

5.2 Syntax

Definition 5.1 describes the abstract syntax of the minimal subset of SafeScript we will use. We call this subset *SafeScript's core language.*

Definition 5.1 (Core language abstract syntax)**.** Let x denote an identifier, the abstract syntax of SafeScript's core language is described as:

program	p ::=	$\overline{c} + \overline{f} + \overline{s}$
class. decl.	c ::=	$\texttt{class}\ x(\overline{x : q})$
func. decl.	f ::=	$\texttt{func}\ x(\overline{x : q}) : q\ \{\overline{s}\}$
type qualifier	q ::=	$\texttt{mut} \mid \texttt{cst}$
statement	s ::=	$\texttt{var}\ x : q \mid e \triangleleft e \mid \texttt{return}\ e$
expression	e ::=	$x \mid e.x \mid e(x \triangleleft e) \mid \texttt{new}\ e$
assign. op.	$\triangleleft$ ::=	$= \mid \texttt{\&-}$

Our syntax features the common constructs of structured imperative programming languages. Nevertheless, we draw the reader's attention to a few points:

- We do not formalize SafeScript's support for non reassignable (i.e. `let`) references. This does not otherwise affect its semantics, since reassignability does not contribute to aliasing safety.
- Function calls require the passing policy [11] of their arguments to be explicitly specified. We re-use our assignment operators (syntactic category $\triangleleft$) for this task. Namely, our copy assignment has a *pass-by-value* semantics, while our borrow and move operators have a *pass-by-reference* semantics.
- We do not formalize coroutines, as they can be emulated on top of our core language [35].

In the remainder of this section, we use P to denote the set of programs produced by the syntactic category p (from Definition 5.1). Similarly, we use E to denote the set of expressions produced by category e, and X to denote the set of identifiers.

5.3 Semantics

We must be able to explicitly differentiate between objects and references (i.e. values and locations), so that we may accurately describe the two assignment semantics we introduced above. To that end, we use L to denote the set of memory locations, and V the set of values a program may manipulate. The latter can be described as follows:

- $\texttt{unit} \in V$ is a built-in value (e.g. a number).
- Let $x_1, \ldots, x_n \in X$ be variable names and $\overline{s}$ be sequence of statements, $\lambda x_1, \ldots, x_n \cdot \overline{s} \in V$ is a function.
- Let $x_1, \ldots, x_n \in X$ be variable names and $l_1, \ldots, l_n \in V$ be memory locations, $[x_1 \rightarrow l_1, \ldots, x_n \rightarrow l_n] \in V$ is a class instance.

We encode class constructors as functions with an empty body. For instance, the constructor of the `Student` class defined in Listing 6 would be a value $\lambda\texttt{name} \cdot \varnothing$.

Definition 5.2 (Evaluation Context)**.** An evaluation context is a pair $\mathcal{E} = \langle \mathcal{L}, \mathcal{V} \rangle$ where:

- $\mathcal{L} : X \rightarrow L$ is a partial function that map variable names to memory locations, and
- $\mathcal{V} : L \rightarrow V$ is a partial function that maps memory locations to values.

SafeScript's big step operational semantics is presented in Figure 2. We write $[\![e, \mathcal{E}]\!]$ the evaluation of an expression e in an evaluation context $\mathcal{E}$, which produces a memory location l and an updated context $\mathcal{E}'$. We use the symbol $\perp \in L$ to denote the uninitialized memory location (i.e. the "null pointer").

Most rules are quite straightforward, so we will only focus on the complex ones. Notice that SafeScript's operational semantics does not take type permissions into account. As we mentioned earlier, the type correctness (w.r.t. aliasing safety) is checked statically before the program is executed. We do not define the rules for functions and classes declarations. Instead, we assume class and functions to be already defined in the initial evaluation context. An example is provided below.

Example 5.3. Consider the following program

```
1 func id(a: cst): cst {
2   return a
3 }
4 class Student {
5   let name: mut
6 }
```

The initial evaluation context $\mathcal{E} = \langle \mathcal{L}, \mathcal{V} \rangle$ of this program is defined such that:

$$\mathcal{V}(\mathcal{L}(\texttt{id})) = \lambda \texttt{a} \cdot \texttt{return a}$$
$$\mathcal{V}(\mathcal{L}(\texttt{Student})) = \lambda \texttt{name} \cdot \varnothing$$

Assignments (Copy) gives the semantics of SafeScript's copy assignment. The left operand must represent a memory location ($[\![e_l, \mathcal{E}]\!] = l_l, \mathcal{E}'$), in which the value at the memory location represented by the right operand will be copied. Notice that the right operand is evaluated first, which is necessary in case it uses the value currently bound to the left operand (e.g. in a function call). We use a function $\text{copy}_{\mathcal{L}} : V \to V$ to denote the transitive (a.k.a. deep) copy of a value. It is equivalent to the identity for built-in values and functions, but applied recursively on class instances. More formally:

$$\frac{x = [x_1 \to l_1, \ldots, x_n \to l_n] \quad v_i = \mathcal{L}(l_i)}{\text{copy}_{\mathcal{L}}(x) = [x_1 \to \text{copy}_{\mathcal{L}}(v_1), \ldots, x_n \to \text{copy}_{\mathcal{L}}(v_n)]}$$

(Bind) and (Bind-m) give the semantics of the borrow operator, for when the left operand is a reference, or the member of a class instance, respectively. In the former case, it suffices to (re-)assign the memory location to which the reference is bound in the context. The approach is ultimately the same in the latter case, except that the reference to be assigned is actually the class instance's member. Note that the update of the class instance in the conclusion of the rule (i.e. $[x \to l_1, \ldots]$) only updates the value of x and leaves other members unchanged.

Variable Declarations (Var) describes the semantics of variable declarations. The premise $l \notin dom(\mathcal{V})$ identifies a fresh memory location to which x should be bound. Notice that $\mathcal{V}$ is updated so that l is mapped to a built-in value, namely `unit`. This actually represents `null`, that is the absence of any value.

Function Calls As mentioned before, we reuse the assignment operators to specify the parameter passing policy on function calls. Similarly, we reuse the semantics of those operators to bind the value of function arguments, as described by the premise $[\![y_i \triangleleft e_i, \mathcal{E}_{i-1}]\!] = \bot, \mathcal{E}_i$ in (Call). We write $\overline{s}[x_1 := y_1, \ldots, x_n := y_n]$ to denote the substitution of the occurrences of x_i for y_i in $\overline{s}$. This renaming makes sure arguments do not clash with other variable names already in the context, since all y_i are assumed to be fresh. Incidentally, this also supports recursive calls. The order in which the arguments are assigned also matters, so as to handle possible side effects. Once the arguments are bound, the body of the function is evaluated with (Seq), until a return statement is reached and the memory location of the function return value is produced, as described in (Return).

5.4 Illustrating Concurrent Mutation

We now present an example of concurrent mutation through the lens of the semantics we have just presented. Consider

$$\frac{x \in X \quad l = \mathcal{L}(x)}{[\![x, \langle \mathcal{L}, \mathcal{V} \rangle]\!] = l, \langle \mathcal{L}, \mathcal{V} \rangle} \quad \text{(Name)}$$

$$\frac{[\![e, \mathcal{E}]\!] = l_e, \langle \mathcal{L}, \mathcal{V} \rangle \quad \mathcal{V}(l_e) = [x \to l_x, \ldots]}{[\![e.x, \mathcal{E}]\!] = l_x, \langle \mathcal{L}, \mathcal{V} \rangle} \quad \text{(Select)}$$

$$\frac{\begin{array}{c}[\![e, \mathcal{E}]\!] = l_e, \langle \mathcal{L}_0, \mathcal{V}_0 \rangle \quad \mathcal{V}_0(l_e) = \lambda x_1, \ldots, x_n \cdot \overline{s} \\ \forall i \in \{1, \ldots, n\}, [\![y_i \triangleleft e_i, \langle \mathcal{L}_{i-1}, \mathcal{V}_{i-1} \rangle]\!] = \bot, \langle \mathcal{L}_i, \mathcal{V}_i \rangle \\ [\![\overline{s}[x_1 := y_1, \ldots, x_n := y_n], \langle \mathcal{L}_n, \mathcal{V}_n \rangle]\!] = l_r, \mathcal{E}'\end{array}}{[\![e(x_1 \triangleleft e_1, \ldots x_n \triangleleft e_n), \mathcal{E}]\!] = l_r, \mathcal{E}'} \quad \text{(Call)}$$

$$\frac{\begin{array}{c}x \in X \quad \mathcal{V}(\mathcal{L}(x)) = \lambda x_1, \ldots, x_n \cdot \varnothing \\ l, l_1, \ldots l_n \notin dom(\mathcal{V}) \\ v = [x_1 \to l_1, \ldots, x_n \to l_n]\end{array}}{[\![\texttt{new } x, \langle \mathcal{L}, \mathcal{V} \rangle]\!] = l, \langle \mathcal{L}[x \mapsto l], \mathcal{V}[l \mapsto v] \rangle} \quad \text{(New)}$$

$$\frac{x \in X \quad l \notin dom(\mathcal{V})}{[\![\texttt{var } x : q, \langle \mathcal{L}, \mathcal{V} \rangle]\!] = \bot, \langle \mathcal{L}[x \mapsto l], \mathcal{V}[l \mapsto \texttt{unit}] \rangle} \quad \text{(Var)}$$

$$\frac{[\![e_1, \mathcal{E}]\!] = l_1, \mathcal{E}' \quad [\![e_0, \mathcal{E}']\!] = l_0, \langle \mathcal{L}, \mathcal{V} \rangle}{[\![e_0 = e_1, \mathcal{E}]\!] = \bot, \left\langle \mathcal{L}, \mathcal{V}[l_0 \mapsto \text{copy}_{\mathcal{L}}(\mathcal{V}(l_1))] \right\rangle} \quad \text{(Copy)}$$

$$\frac{x \in X \quad [\![e, \mathcal{E}]\!] = l, \langle \mathcal{L}, \mathcal{V} \rangle}{[\![x \texttt{ \&- } e, \mathcal{E}]\!] = \bot, \langle \mathcal{L}[x \mapsto l], \mathcal{V} \rangle} \quad \text{(Bind)}$$

$$\frac{\begin{array}{c}[\![e_1, \mathcal{E}]\!] = l_1, \mathcal{E}' \quad [\![e_0, \mathcal{E}']\!] = l_0, \langle \mathcal{L}, \mathcal{V} \rangle \\ \mathcal{V}(l_0) = [x \to l_x, \ldots]\end{array}}{[\![e_0.x \texttt{ \&- } e_1, \mathcal{E}]\!] = \bot, \langle \mathcal{L}, \mathcal{V}[l_0 \mapsto [x \to l_1, \ldots]] \rangle} \quad \text{(Bind-m)}$$

$$\frac{[\![e, \mathcal{E}]\!] = l, \mathcal{E}'}{[\![\texttt{return } e, \mathcal{E}]\!] = l, \mathcal{E}'} \quad \text{(Return)}$$

$$\frac{\overline{s} = s_h | \overline{s_t} \quad [\![s_h, \mathcal{E}]\!] = l_h, \mathcal{E}' \quad [\![\overline{s_t}, \mathcal{E}']\!] = l_t, \mathcal{E}''}{[\![\overline{s}, \mathcal{E}]\!] = l_t, \mathcal{E}''} \quad \text{(Seq)}$$

Figure 2. Operational semantics

the program in Listing 9 which, if executed, will fail at line 9, while attempting to divide a number by zero. The problem is that value `p.x` is mutated one line before, since `x` is bound to the same object. This can be observed under SafeScript's semantics if we pay attention on the way arguments were passed to the function `f`, as well as the constructor of `Point`. In both instances, we used a borrow assignment (i.e. "`&-`") rather than a copy, so rules (Bind) and (Bind-m) applied. The first bound `a` to $\mathcal{L}(x)$ and the second bound `p.x` to the same location.

The goal of our type system, which we formalize in the following section, is to statically detect and ultimately prevent

```
class Point { var x; var y }
func f(a, b) {
  return new Point(x &- a, y &- b)
}

var x = 2
var p = f(a &- x, b &- x)
x = 0
print(2 / p.x)
```

Listing 9. Example of concurrent mutation (SafeScript). The program fails because the value to which `p.x` is bound was mutated.

such situations. In fact, it should not be possible to capture immutable borrows to `a`, because the function `f` cannot guarantee the immutability of the object it is associated with beyond its own scope.

6 Type System

In this section, we formalize the typing semantics of SafeScript. It is flow sensitive, meaning that it does not only depend on the statements but also on their ordering. Each statement modifies a typing context, which maps references and objects to permissions.

Definition 6.1 (Type Permission)**.** The set of type permissions is given by $\mathbb{P} = \{ro, rw\}$, where ro is the read-only permission, and rw is the read-write permission.

Apart from the built-in types, which are referred to with `Unit`, we write $f(x_0 : q_0, \ldots, x_n : q_n) : q_r$ a particular function type and $C(x_0 : q_0, \ldots, x_n : q_n)$ a particular class type. We use the term *semantic type* to refer to these types, as they describe the intended use of an object's value, as well as the operations it may support, and write T for the set of all semantic types.

Definition 6.2 (Typing Context)**.** Let L denote the set of memory locations. A typing context is a triple $C = \langle \Gamma, \Psi, \Pi \rangle$ where:

- Γ is a partial function $X \rightharpoonup T$ that maps variables to types,
- Ψ is a partial function $E \rightharpoonup L$ that maps expressions to memory locations,
- Π is a partial function $E \cup L \rightharpoonup \mathcal{P}(\mathbb{P})$ that maps expressions and memory locations to a set of permissions.

Note that the domain of the function Ψ includes all expressions (and sub-expressions), not only variable names. The rationale is that all expressions actually represent some value, which itself must live at some memory location. Hence, we refer to both variable names and other expressions as *reference expressions*.

Given a typing context C, we say say that the *context type* type of the expression is the combination of its semantic type with its associated permissions.

Let $C = \langle \Gamma, \Psi, \Pi \rangle$ be a typing context. The function $R^0_C(l)$ returns the set of reference expressions $e \in E$ that refer to the memory location $l \in L$. More formally, $R^0_C(l) = \{e \in E \mid \Psi(e) = l\}$. When l holds a class instance (i.e. $e \in R^0_C(l) \implies \Gamma(e) = C(e_1 : q_1 \ldots, e_n : q_n)$), we write $R_C(l)$ the function that returns the set of reference expressions that not only refer to the memory location $l \in L$, but also to any location $l_i \in L$ that is part of the representation of the object at l. More formally, $R_C(l) = R^0_C(l) \bigcup_{e_i} R_C(\Psi(e.e_i))$. If l does not hold a class instance, then $R_C(l) = R^0_C(l)$. We write $R^{ro}_C(l) = \{e \mid e \in R_C(l) \wedge ro \in \Pi(e)\}$ for the set of immutable references on the object at l, and define the set of mutable references $R^{rw}_C(l)$ similarly. Note that this definition does distinguish between references and *borrowed* references (i.e. additional aliases created with the `&-` operator). But we can determine whether a reference is borrowed by checking whether $|R_C(l)| > 1$.

Example 6.3 (Reference expressions)**.** Consider the following program:

```
var pt: mut = new Point(x = 0, y = 1)
var r &- pt
var x &- pt.x
```

Let the typing context C represents the type state of this program after its last statement. $R^0_C(\Psi(\texttt{pt}))$ contains the references that directly refer to the object represented by `pt`, that is the set $\{\texttt{pt}, \texttt{r}\}$. $R_C(\Psi(\texttt{pt}))$ also contains `x`, because it refers to some part of the object representation of the object represented by `pt`. Therefore, $R_C(\Psi(\texttt{pt})) = \{\texttt{pt}, \texttt{r}, \texttt{x}\}$. $R^{ro}_C(\Psi(\texttt{pt.x})) = \{\texttt{x}\}$ as it is the only immutable reference on `pt.x`, but $R^{rw}_C(\Psi(\texttt{pt.x})) = \varnothing$ because there are no read-write reference on `pt.x`.

As introduced in Section 4.4, SafeScript enforces data race safety by disallowing concurrent read-only and read-write borrows. The difficulty of satisfying such an invariant stems from the fact that not only should mutable references lose their writing privileges when borrowed immutably, but also that such privileges should be restored once said immutable borrows are dead [3]. One way to implement this mechanism is by associating objects with permissions as well. Then, the actual permissions of a reference are computed as the intersection of its own permissions with that of the object it is bound to. As a result, if an object is stripped from its read-write permission, the effective permissions of its references will be updated accordingly. Conversely, if the read-write permission is given back to an object, the effective permissions of its references will be restored. Therefore, it suffices to guarantee that as long as there exists at least one read-only (resp. read-write) borrow on a reference, the object it refers

to shall be stripped from its read-write (resp. read-only) permission. More formally, we can define the aliasing safety invariant as follows:

Definition 6.4 (Aliasing Safety Invariant)**.** SafeScript enforces data race safety by guaranteeing that, for any typing context $C = \langle \Gamma, \Psi, \Pi \rangle$, a reference shall have either n read-only borrows and 0 read-write borrows, or 0 read-only borrows and m read-write borrows at any given time.

$$\forall l \in L, |R_C^{ro}(l)| > 1 \implies rw \notin \Pi(l) \quad (1)$$

$$\forall l \in L, |R_C^{rw}(l)| > 1 \implies ro \notin \Pi(l) \quad (2)$$

Notice that the restriction on the permissions of l only applies when there is more than a single read-only reference. That is because we give the read-only *and* the read-write permission to references that refer to a newly allocated mutable object (see Example 4.1). In fact, this underlines the difference between a reference and a *borrowed* reference. Our flow-sensitive judgment is of the form $C \vdash s \Rightarrow C'$, where s is a statement, C the typing context in which the statement is consumed and C' the typing context obtained after consuming the statement. A typing context is *well-formed* if it satisfies the aliasing safety invariant.

Programs Programs are defined as a sequence of function and class declarations, followed by a sequence of statements. Hence, all we have to do is apply our flow-sensitive judgment on each component of a program, sequentially:

$$\frac{\bar{x} = x_h | \overline{x_t} \quad C \vdash x_h \Rightarrow C' \quad C' \vdash \overline{x_t} \Rightarrow C''}{C \vdash \bar{x} \Rightarrow C''} \text{(Statement-List)}$$

Variable Declarations The following equations describe the typing rules for variable declarations. Both feature the premise $x \notin dom(\Pi)$ to prevent duplicate declarations, and both update the typing context with the contextual type of the new variable. Note that no new memory allocation is being performed, as this will be carried out by the assignment rules.

$$\frac{x \notin dom(\Pi)}{\Gamma, \Psi, \Pi \vdash \texttt{var}\ x : \texttt{cst} \Rightarrow \Gamma, \Psi, \Pi[x \mapsto \{ro\}]} \text{(Immut-Decl)}$$

$$\frac{x \notin dom(\Pi)}{\Gamma, \Psi, \Pi \vdash \texttt{var}\ x : \texttt{mut} \Rightarrow \Gamma, \Psi, \Pi[x \mapsto \{ro, rw\}]} \text{(Mut-Decl)}$$

Parameter Declarations Parameter declarations work just as variable declarations, except that mutable references do not get a read-only permission. This makes sure an argument passed as a borrowed mutable reference may not be borrowed immutable in the body of the function.

$$\frac{x \notin dom(\Pi)}{\Gamma, \Psi, \Pi \vdash x : \texttt{cst} \Rightarrow \Gamma, \Psi, \Pi[x \mapsto \{ro\}]} \text{(Immut-Par)}$$

$$\frac{x \notin dom(\Pi)}{\Gamma, \Psi, \Pi \vdash x : \texttt{mut} \Rightarrow \Gamma, \Psi, \Pi[x \mapsto \{rw\}]} \text{(Mut-Par)}$$

Expression Permissions The next set of rules presents the permission typing judgement $\vdash_\tau$, which infers the effective permissions of an expression. We write $e \sqsubset \tau, \rho, C$ to denote that e has the contextual type τ, ρ in the context C. We use a total alternative Π^+ to our partial permission mapping function Π, that returns the set of all permissions $\{ro, rw\}$ when Π is not defined. (Ref-Type) types references intersecting their permissions with that of their associated memory location to determine their effective permissions. (Select-Type) types instance member selection similarly, except that it also uses the effective permissions of references preceding the dot operator to compute that of the reference. Incidentally, this is how we freeze paths to immutable values.

$$\frac{x \in X \quad \Gamma(x) = \tau \quad \rho = \Pi(x) \quad \rho' = \Pi^+(\Psi(x))}{\Gamma, \Psi, \Pi \vdash_\tau x \sqsubset \tau, \rho \cap \rho', \langle \Gamma, \Psi, \Pi \rangle} \text{(Ref-Type)}$$

$$\frac{C \vdash_\tau e \sqsubset C(x_1 : \tau_1, \ldots, x_n : \tau_n), \rho_e, \langle \Gamma, \Psi, \Pi \rangle \quad x_i \in X \quad \rho = \Pi(x_i) \quad \rho' = \Pi^+(\Psi(x_i))}{C \vdash_\tau e.x_i \sqsubset \tau_i, \rho \cap \rho' \cap \rho_e, \langle \Gamma, \Psi, \Pi \rangle} \text{(Select-Type)}$$

Function calls require a little more attention, as we should pay a particular care as to how arguments and return types interact with the typing context. From the point of view of the call site, calling a function amounts to performing a sequence of assignments to set the function arguments. In more concrete terms, from the perspective of the call site, the statement `var r = f(arg1 = val1, arg2 &- val2)` can be reduced to the following program:

```
1 var arg1 =  val1
2 var arg2 &- val2
3 var r    =  return_value_of_f
```

(Call-Type) captures this intuition. Assuming f refers to a function $f(x_1 : q_1, \ldots, x_n : q_n) : q_r$, we successively apply our flow-sensitive judgement $\vdash$ on each argument assignment. We then apply each argument assignment, eventually producing C_n. Note that we are not interested about the permissions ρ associated with the function itself to call it. Instead, we use the qualifier of the return type to determine the permissions of the return value. Finally, we need to express the restoration of the permissions removed by borrowed arguments. To that end, we define a special expression `null`, which correspond to the absence of any actual value. Its contextual type is defined such that `null` $\sqsubset$ `unit`, $\{ro, rw\}, C$ for any typing context C. We then reassign all arguments to `null`, so as to compute the final context produced by the function call. The semantics of those reassignment is given by the rules (Immut-Update-Borrow) and

(Mut-Update-Borrow), which we will detail later.

$$\frac{C \vdash_\tau f \sqsubset f(x_1 : q_1, \ldots, x_n : q_n) : q_r, \rho, C_0 \quad \forall i \in \{1, \ldots, n\}, C_{i-1} \vdash x_i \triangleleft e_i \Rightarrow C_i \quad \forall i \in \{1, \ldots, n\}, C_{n+i-1} \vdash x = \texttt{null} \Rightarrow C_{n+i}}{C \vdash_\tau f(x_1 \triangleleft e_1, \ldots, x_n \triangleleft e_n) \sqsubset \tau_r, \rho_r, C_{n+n}} \text{(Call-Type)}$$

Remember that we treat class constructors just as regular functions (see Section 5). Therefore, class instantiations are treated very similarly to function calls. However, since the class instance persists after the call to the constructor, so should the update on the permissions of the reference it borrows. Hence, we do not keep the last premise of (Call-Type).

$$\frac{C \vdash_\tau f \sqsubset f(x_1 : q_1, \ldots, x_n : q_n) : q_r, \rho, C_0 \quad \forall i \in \{1, \ldots, n\}, C_{i-1} \vdash x_i \triangleleft e_i \Rightarrow C_i}{C \vdash_\tau \texttt{new } f(x_1 \triangleleft e_1, \ldots, x_n \triangleleft e_n) \sqsubset \tau_r, \rho_r, C_{n+n}} \text{(Class-Inst)}$$

Return Statements Return statements simply compute the final typing context of a function. As a result, typing them boils down to computing the contextual type of the return expression

$$\frac{C \vdash_\tau e \sqsubset C'}{C \vdash \texttt{return } e \Rightarrow C'} \text{(Return)}$$

Copy Assignments We start by defining a function alloc : $(E \to L) \to L$ that, given an memory location mapping function $\Psi : E \to L$, returns a new location l such that $\{e \in E \mid \Psi(e) = l\} = \varnothing$. Simply put, alloc allocates new memory locations. We then formalize the semantics of copy assignments. (Assign) applies when the left-hand side of the operator has been declared, but has yet to be bound to an memory location ($x \notin dom(\Psi)$). We use a total alternative Ψ^+ to our memory mapping function Ψ, that returns the special location $\perp \in L$ when Ψ is not defined. We use alloc to allocate a new memory location l_x and bind it to x. Note that the l_x receives the same permissions as that of x. (Update) applies when the left-hand side of the operator is already bound ($l_x = \Psi(x)$), and mutable ($rw \in \rho_x$).

$$\frac{C \vdash_\tau e \sqsubset \tau_e, \rho_e, C_e \quad C_e \vdash_\tau x \sqsubset \tau_x, \rho_x, \langle \Gamma, \Psi, \Pi \rangle \quad x \notin dom(\Psi) \quad l_e = \Psi^+(e) \quad l_x = \text{alloc}(\Psi)}{C \vdash x = e \Rightarrow \Gamma[x \mapsto \tau_e], \Psi[x \mapsto l_x], \Pi[l_x \mapsto \rho_x]} \text{(Assign)}$$

$$\frac{C \vdash_\tau e \sqsubset \tau_e, \rho_e, C_e \quad C_e \vdash_\tau x \sqsubset \tau_x, \rho_x, \langle \Gamma, \Psi, \Pi \rangle \quad l_x = \Psi(x) \quad l_e = \Psi^+(e) \quad rw \in \rho_x}{C \vdash x = e \Rightarrow \Gamma[x \mapsto \tau_e], \Psi, \Pi} \text{(Update)}$$

Borrow Assignments Immutable borrows to unbounded references are described by (Immut-Borrow). Remember that we use the type of the left operand to determine whether a borrow is mutable or immutable (see Section 4.4). Therefore, we use the premise $rw \notin \rho_x$ to assert the left-hand expression is immutable. We also check that x is currently unbound with $x \notin dom(\Psi)$. $ro \in \rho_e$ and $|R^{rw}_{\langle \Gamma, \Psi, \Pi \rangle}(l_e)| \leq 1$ maintains the aliasing safety invariant (Definition 6.4) by prohibiting immutable reborrows on mutable borrows. Finally, notice how we make sure memory locations are stripped from their read-write permission in the conclusion of the rule, which is the mechanism that prevents further incompatible reborrows.

$$\frac{C \vdash_\tau e \sqsubset \tau_e, \rho_e, C_e \quad C_e \vdash_\tau x \sqsubset \tau_x, \rho_x, \langle \Gamma, \Psi, \Pi \rangle \quad rw \notin \rho_x \quad ro \in \rho_e \quad x \notin dom(\Psi) \quad l_e = \Psi^+(e) \quad |R^{rw}_{\langle \Gamma, \Psi, \Pi \rangle}(l_e)| \leq 1}{C \vdash x \;\&\text{-}\; e \Rightarrow \Gamma[x \mapsto \tau_e], \Psi[x \mapsto l_e], \Pi[x, l_e \mapsto \{ro\}]} \text{(Immut-Borrow)}$$

(Mut-Borrow) is almost completely dual, with respect to permissions. A subtle difference is that we check for rw in the permissions of x, rather than checking for the absence of ro. The reason is that mutable references are declared with both, as described in (Mut-Par).

$$\frac{C \vdash_\tau e \sqsubset \tau, \rho_e, C_e \quad C_e \vdash_\tau x \sqsubset \tau, \rho_x, \langle \Gamma, \Psi, \Pi \rangle \quad rw \in \rho_x \quad rw \in \rho_e \quad l_e = \Psi^+(e) \quad |R^{ro}_{\langle \Gamma, \Psi, \Pi \rangle}(l_e)| \leq 1}{C \vdash x \;\&\text{-}\; e \Rightarrow \Gamma[x \mapsto \tau_e], \Psi[x \mapsto l_e], \Pi[x, l_e \mapsto \{rw\}]} \text{(Mut-Borrow)}$$

Borrows to bound references are slightly more complex. Indeed, in an effort to maintain the aliasing safety invariant, we updated the permissions on the memory location represented by the right-hand expressions in both (Immut-Borrow) and (Mut-Borrow). Therefore, we need to express the restoration of the permissions removed by borrowed references, once the latter are reassigned. To that end, we define a function $\text{res}_C(l, e)$ that returns the permissions $\rho \in \mathcal{P}(\mathbb{P})$ of a location $l \in L$, without the restriction imposed by a borrowed reference expression $e \in E$. We obtain such permissions by forming the intersection between that of all borrowed references on l, except e:

$$\text{res}_{\langle \Gamma, \Psi, \Pi \rangle}(l, e) = \bigcap_{x \in R_C(l) - e} \Pi(x)$$

We then present the inference rules for borrows to bounded references:

$$\frac{C \vdash_\tau e \sqsubset \tau_e, \rho_e, C_e \quad C_e \vdash_\tau x \sqsubset \tau_x, \rho_x, \langle \Gamma, \Psi, \Pi \rangle \quad rw \notin \rho_x \quad ro \in \rho_e \quad l_x = \Psi(x) \quad l_e = \Psi^+(e) \quad |R^{rw}_{\langle \Gamma, \Psi, \Pi \rangle}(l_e)| \leq 1 \quad \Pi' = \Pi[x, l_e \mapsto \{ro\}][l_x \mapsto \text{res}_{\langle \Gamma, \Psi, \Pi \rangle}(l_x, x)]}{C \vdash x \;\&\text{-}\; e \Rightarrow \Gamma[x \mapsto \tau_e], \Psi[x \mapsto l_e], \Pi'} \text{(Immut-Update-Borrow)}$$

$$\frac{C \vdash_\tau e \sqsubset \tau_e, \rho_e, C_e \quad C_e \vdash_\tau x \sqsubset \tau_x, \rho_x, \langle \Gamma, \Psi, \Pi \rangle \quad rw \in \rho_x \quad rw \in \rho_e \quad l_x = \Psi(x) \quad l_e = \Psi^+(e) \quad |R^{ro}_{\langle \Gamma, \Psi, \Pi \rangle}(l_e)| \leq 1 \quad \Pi' = \Pi[x, l_e \mapsto \{rw\}][l_x \mapsto \text{res}_{\langle \Gamma, \Psi, \Pi \rangle}(l_x, x)]}{C \vdash x \;\&\text{-}\; e \Rightarrow \Gamma[x \mapsto \tau_e], \Psi[x \mapsto l_e], \Pi'} \text{(Mut-Update-Borrow)}$$

```
1  var a: mut      Π(a) = {ro, rw}
2  var b: cst      Π(b) = {ro}
3  var r: cst      Π(r) = {ro}
4
5  a = 0           Ψ(a) = l1, Π(l1) = {ro, rw}
6  b = 0           Ψ(b) = l2, Π(l2) = {ro}
7
8  r &- a          Ψ(r) = l1, Π(l1) = {ro}
9  r &- b          Ψ(r) = l2, Π(l2) = {ro, rw}
10 a &- b          Illegal
```

Listing 10. Example of assignment semantics (SafeScript).

Example 6.5 (Borrow Assignment Semantics)**.** Consider the program in Listing 10. Each line is annotated with the changes in the contextual type. The assignment at line 8 creates a borrow on a. Here, (Immut-Borrow) applies, because r is declared immutable at line 3, and is not yet bound to a value. Furthermore, a temporary immutability constraint is made on l_1 (the memory to which a is bound). r is reassigned at line 9. This time (Immut-Update-Borrow) applies, as $x \notin dom(\Psi)$ does not hold, and the read-write permission on l_1 is restored. Line 10 is illegal because it attempts to create a mutable borrow on an immutable reference. Indeed, since a was declared mutable, (Immut-Update-Borrow) cannot apply, and neither can (Mut-Update-Borrow), because the premise $rw \in \rho_e$ is not satisfied.

Note that the same principle applies when a reference goes out of scope. In fact, the rules are nearly identical to the above rules, with the notable exception that the reference is not reassigned to another object.

6.1 Limitations

Our type system is unable to distinguish fresh return values – in other terms objects created within a function – from references a function would have got as parameter. The consequence is that, from the call site, we cannot track temporary immutability constraints placed in the body of a function. An example is given in Listing 11.

One workaround is to consider all return values as deep copies. This way, no immutability constraint can persist on the arguments of the function. A more elaborate solution would be to enrich the type of a function so that the call site may be aware of the permission update the arguments will undergo.

6.2 Soundness Result

Our soundness result is based the standard syntactic approach of preservation and progress [44]. In brief, our goal is to show that our type system either is able to compute the final typing context of a program, in which case the program is well-typed, or the computation terminates with an invalid state. Such invalid state is characterized by either a typing

```
1 func f(a: mut): cst {
2   return a
3 }
4 var x: mut = 0
5 var y: cst = f(a &- x)
6 x = 2 // Illegal if y borrows x
```

Listing 11. A concealed immutability constraint (SafeScript).

context that is not well-formed, or a statement list that our type system is unable to reduce. We formulate both theorems as follows:

Theorem 6.6 (Preservation)**.** *Let C be a well-formed typing context, if $C \vdash s \Rightarrow C'$, then C' is well-formed.*

Theorem 6.7 (Progress)**.** *Let $P = \overline{c} + \overline{f} + \overline{s}$. For any $i < |\overline{s}|$, if $C \vdash \overline{s}_i \Rightarrow C'$ then there exists C'' such that $C' \vdash \overline{s}_{i+1} \Rightarrow C''$, or there is no matching rule and the reduction is stuck.*

Theorem 6.6 guarantees that reduction preserves well-formedness of typing contexts, while theorem 6.7 ensures the reduction either progresses towards a final typing context, or is stuck and can never type check P. Soundness of the type system follows from these two theorems. The proof consists of ensuring that no reduction rule can jeopardize the well-formedness of a typing contexts, which is done by case analysis on each reduction rule. We do not include its details for spacial reasons.

Note that this is not a strong soundness result, as one would also involve SafeScript's operational semantics. We do not include the detailed proof for spacial reasons, but sketch it as follows. Only (Copy) and (Bind-m) may introduce concurrent mutation, as they are the only rules modifying $\mathcal{V}$ for a location l that may already be bound to multiple references. For the sake of the proof, the evaluation context $\mathcal{E}$, as well as the (Bind) rule must be modified, so that we may distinguish between read-write and read-only references during the evaluation of a particular expression. Finally, one has to show that an immutable reference may never appear on the left side of a copy assignment in a well-typed program.

7 Conclusion

We propose a type system that guarantees aliasing safety, so as to prevent data race, based on type permissions and borrowing. Since our work is set in the context of cooperative single-threaded programs, our type system does not restrict the number of mutable borrowed references, as it is common in most other approaches. This allows us to support multiple writers, an essential feature to represent mutable self-referential data structures, without resorting to elaborate workarounds or bypassing the language's type checking mechanisms.

We present our type system in the context of SafeScript, a reference-based language that only requires the addition

of minimal annotations on variable, type and function declarations. An implementation of a compiler for SafeScript, as well as various example programs showcasing our type system, are available here https://github.com/kyouko-taiga/SafeScript.

The methodology of our type system is agnostic to a specific programming language and can easily be extended with additional features. It has been implemented in Anzen[3], another programming language that focuses on memory safety, where it is extended by an ownership system to handle memory deallocation.

References

[1] Jean-Baptiste Arnaud, Marcus Denker, Stéphane Ducasse, Damien Pollet, Alexandre Bergel, and Mathieu Suen. 2010. Read-only Execution for Dynamic Languages. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns (TOOLS'10)*. Springer-Verlag, Berlin, Heidelberg, 117–136. http://dl.acm.org/citation.cfm?id=1894386.1894393

[2] Gavin M. Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*. 257–281. https://doi.org/10.1007/978-3-662-44202-9_11

[3] John Boyland. 2001. Alias burying: Unique variables without destructive reads. *Software - Practice and Experience* 31, 6 (2001), 533–553. https://doi.org/10.1002/spe.370

[4] John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*. 2–27. https://doi.org/10.1007/3-540-45337-7_2

[5] John Tang Boyland and William Retert. 2005. Connecting Effects and Uniqueness with Adoption. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, New York, NY, USA, 283–295. https://doi.org/10.1145/1040305.1040329

[6] Jacob Burnim, Tayfun Elmas, George C. Necula, and Koushik Sen. 2012. CONCURRIT: Testing Concurrent Programs with Programmable State-Space Exploration. In *4th USENIX Workshop on Hot Topics in Parallelism, HotPar'12, Berkeley, CA, USA, June 7-8, 2012*.

[7] Nick Cameron. 2018. Graphs and arena allocation. https://github.com/nrc/r4cppp/blob/master/graphs/README.md. (2018). Accessed: 2018-04-09.

[8] Nicholas Robert Cameron, Sophia Drossopoulou, James Noble, and Matthew J. Smith. 2007. Multiple ownership. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. 441–460. https://doi.org/10.1145/1297027.1297060

[9] Dave Clarke and Tobias Wrigstad. 2003. External uniqueness is unique enough. *ECOOP 2003–Object-Oriented Programming* (2003), 59–67.

[10] David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*. ACM, New York, NY, USA, 48–64. https://doi.org/10.1145/286936.286947

[11] Erik Crank and Matthias Felleisen. 1991. Parameter-passing and the Lambda Calculus. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '91)*. ACM, New York, NY, USA, 233–244. https://doi.org/10.1145/99583.99616

[12] Erich Gamma. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

[13] Prodromos Gerakios, Nikolaos Papaspyrou, and Konstantinos Sagonas. 2014. Static safety guarantees for a low-level multithreaded language with regions. *Science of Computer Programming* 80 (2014), 223–263.

[14] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and Reference Immutability for Safe Parallelism. *SIGPLAN Not.* 47, 10 (oct 2012), 21–40. https://doi.org/10.1145/2398857.2384619

[15] Donald Gordon and James Noble. 2007. Dynamic Ownership in a Dynamic Language. In *Proceedings of the 2007 Symposium on Dynamic Languages (DLS '07)*. ACM, New York, NY, USA, 41–52. https://doi.org/10.1145/1297081.1297090

[16] Rachid Guerraoui and Paolo Romano (Eds.). 2015. *Transactional Memory. Foundations, Algorithms, Tools, and Applications - COST Action Euro-TM IC1001*. Lecture Notes in Computer Science, Vol. 8913. Springer. https://doi.org/10.1007/978-3-319-14720-8

[17] Christian Haack, Erik Poll, Jan Schäfer, and Aleksy Schubert. 2007. Immutable Objects for a Java-Like Language. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*. 347–362. https://doi.org/10.1007/978-3-540-71316-6_24

[18] Philipp Haller and Martin Odersky. 2010. Capabilities for uniqueness and borrowing. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6183 LNCS (2010), 354–378. https://doi.org/10.1007/978-3-642-14107-2_17

[19] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. 1984. Continuations and Coroutines. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP '84)*. ACM, New York, NY, USA, 293–298. https://doi.org/10.1145/800055.802046

[20] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2004. Experience with Safe Manual Memory-management in Cyclone. In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*. ACM, New York, NY, USA, 73–84. https://doi.org/10.1145/1029873.1029883

[21] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (may 2001), 396–450. https://doi.org/10.1145/503502.503505

[22] M. E. Joorabchi, A. Mesbah, and P. Kruchten. 2013. Real Challenges in Mobile App Development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 15–24. https://doi.org/10.1109/ESEM.2013.9

[23] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang. 2, POPL, Article* (2018).

[24] Gabriel Kerneis and Juliusz Chroboczek. 2011. CPC: programming with a massive number of lightweight threads. *CoRR* abs/1102.0951 (2011). arXiv:1102.0951 http://arxiv.org/abs/1102.0951

[25] Jack A. Laird. 2006. A calculus of coroutines. *Theor. Comput. Sci.* 350, 2-3 (2006), 275–291. https://doi.org/10.1016/j.tcs.2005.10.027

[26] K. Rustan M. Leino, Peter Müller, and Angela Wallenburg. 2008. Flexible immutability with frozen objects. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5295 LNCS (2008), 192–208. https://doi.org/10.1007/978-3-540-87873-5-17

[27] Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. *Ada Lett.* 34, 3 (oct 2014), 103–104. https://doi.org/10.1145/2692956.2663188

[28] Ana Lúcia De Moura and Roberto Ierusalimschy. 2009. Revisiting Coroutines. *ACM Trans. Program. Lang. Syst.* 31, 2, Article 6 (feb 2009), 31 pages. https://doi.org/10.1145/1462166.1462167

[3] http://anzen-lang.org

[29] Alan Mycroft and Janina Voigt. 2013. Notions of aliasing and ownership. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7850 (2013), 59–83. https://doi.org/10.1007/978-3-642-36946-9-4

[30] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. 2012. A Type System for Borrowing Permissions. *SIGPLAN Not.* 47, 1 (Jan. 2012), 557–570. https://doi.org/10.1145/2103621.2103722

[31] Johan Östlund, Tobias Wrigstad, Dave Clarke, and Beatrice Åkerblom. 2008. Ownership, Uniqueness, and Immutability. In *Objects, Components, Models and Patterns, 46th International Conference, TOOLS EUROPE 2008, Zurich, Switzerland, June 30 - July 4, 2008. Proceedings*. 178–197. https://doi.org/10.1007/978-3-540-69824-1_11

[32] Alex Potanin, Johan Ostlund, Yoav Zibin, and Michael D Ernst. 2013. Immutability. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Springer Berlin Heidelberg, 233–269. https://doi.org/10.1007/978-3-642-36946-9_9

[33] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

[34] Kristian Rother. 2017. *Static Typing in Python*. Apress, Berkeley, CA, 231–244. https://doi.org/10.1007/978-1-4842-2241-6_16

[35] Rami Sarieddine. 2014. *JavaScript Promises Essentials*. Packt Publishing Ltd.

[36] Ilya Sergey and Dave Clarke. 2012. Gradual Ownership Types. In *Proceedings of the 21st European Conference on Programming Languages and Systems (ESOP'12)*. Springer-Verlag, Berlin, Heidelberg, 579–599. https://doi.org/10.1007/978-3-642-28869-2_29

[37] Kishori Sharan. 2018. Collections. In *Java Language Features*. Springer, 587–674.

[38] K. Simpson. 2015. *You Don't Know JS: ES6 & Beyond*. O'Reilly Media. https://books.google.ch/books?id=rec6CwAAQBAJ

[39] Martin Steffen. 2016. A Small-Step Semantics of a Concurrent Calculus with Goroutines and Deferred Functions. In *Essays Dedicated to Frank De Boer on Theory and Practice of Formal Methods - Volume 9660*. Springer-Verlag New York, Inc., New York, NY, USA, 393–406. https://doi.org/10.1007/978-3-319-30734-3_26

[40] Nikhil Swamy, Michael W. Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2006. Safe manual memory management in Cyclone. *Sci. Comput. Program.* 62, 2 (2006), 122–144. https://doi.org/10.1016/j.scico.2006.02.003

[41] Akihiko Tozawa, Michiaki Tatsubori, Tamiya Onodera, and Yasuhiko Minamide. 2009. Copy-on-write in the PHP Language. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, USA, 200–212. https://doi.org/10.1145/1480881.1480908

[42] Matthew S. Tschantz and Michael D. Ernst. 2005. Javari: Adding Reference Immutability to Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 211–230. https://doi.org/10.1145/1094811.1094828

[43] Paul R. Wilson. 1992. Uniprocessor Garbage Collection Techniques. In *Memory Management, International Workshop IWMM 92, St. Malo, France, September 17-19, 1992, Proceedings*. 1–42. https://doi.org/10.1007/BFb0017182

[44] Andrew K. Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and Computation* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093

[45] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kie, un, and Michael D. Ernst. 2007. Object and Reference Immutability Using Java Generics. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 75–84. https://doi.org/10.1145/1287624.1287637