JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Implementation Strategies for Mutable Value Semantics

Dimitri Racordon[*,§], Denys Shabalin[††], Daniel Zheng[†], Dave Abrahams[‡], and Brennan Saeta[†]
[*]University of Geneva, Faculty of Science, Switzerland
[§]Northeastern University, USA
[††]Google, Switzerland
[†]Google, USA
[‡]Adobe, USA

ABSTRACT Mutable value semantics is a programming discipline that upholds the independence of values to support local reasoning. In the discipline's strictest form, references become second-class citizens: they are only created implicitly, at function boundaries, and cannot be stored in variables or object fields. Hence, variables can never share mutable state. Unlike pure functional programming, however, mutable value semantics allows part-wise in-place mutation, thereby eliminating the memory traffic usually associated with functional updates of immutable data.
This paper presents implementation strategies for compiling programs with mutable value semantics into efficient native code. We study Swift, a programming language based on that discipline, through the lens of a core language that strips some of Swift's features to focus on the semantics of its value types. The strategies that we introduce leverage the inherent properties of mutable value semantics to unlock aggressive optimizations. Fixed-size values are allocated on the stack, thereby enabling numerous off-the-shelf compiler optimizations, while dynamically sized containers use copy-on-write to mitigate copying costs.

KEYWORDS Mutable value semantics, local reasoning, memory safety, borrowing, copy-on-write, compilation, optimizations.

1. Introduction

Software development continuously grows in complexity, as applications get larger and hardware more sophisticated. The essential principle required to build correct systems in the face of growing complexity is *local reasoning*, described by O'Hearn et al. (2001) as follows:

> To understand how a program works, it should be possible for reasoning and specification to be confined to the cells that the program actually accesses. The value of any other cell will automatically remain unchanged.

The ability to reason locally about program semantics is also critical for efficiency as it eliminates the need for conservative assumptions about access to the memory. Unfortunately, local reasoning is often lost in a shared memory model, as side-effectful operations in one part of the program impact seemingly-unrelated locations.

Pure functional programming addresses this problem by simply outlawing mutation. Unfortunately, this paradigm may fail to capture the programmer's mental model, or prove ill-suited to express and optimize some algorithms (O'Neill 2009), due to the inability to express in-place mutation. For instance, if `x` is a binary tree, an assignment that in Java could be written `x.left.left.right = v` must be translated into a cumbersome composition of updates `x' = updateLeft(x, updateLeft(x.left, updateRight(x.left.right, v)))`, which can only match the algorithmic efficiency of the original formulation through optimizer heroics that eliminate unnecessary copies of temporary data (Johann 2003).

Another way to uphold local reasoning is to use first-class references, but tame their aliasing. Newer programming languages have blended ideas from ownership types (Clarke et al. 2013), type capabilities (Haller & Odersky 2010), and region-based memory management (Tofte et al. 2004), offering more

JOT reference format:
Dimitri Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, and Brennan Saeta. *Implementation Strategies for Mutable Value Semantics*. Journal of Object Technology. Vol. vv, No. nn, yyyy. Licensed under Attribution - NonCommercial - No Derivatives 4.0 International (CC BY-NC-ND 4.0) http://dx.doi.org/10.5381/jot.yyyy.vv.nn.aa

freedom to write efficient and type-safe programs. These ideas, however, invariably complicate type systems with mechanisms like named lifetimes, which significantly raise the barrier to entry for inexperienced developers (Turner 2017).

```rust
fn longer_of(x: String, y: String) -> String {
  if x.len() > y.len() { x } else { y }
}

fn report_longest(x: String, y: String) {
  let z = longer_of(x, y);
  println!("longest of {:?} and {:?} is {:?}",
           x, y, z); // <- error
}
```

Consider the Rust (Matsakis & Klock 2014) example above. A simple function `longer_of` returns the longer of two character strings. Its caller, `report_longest`, also simple, is ill-typed. The compiler complains that `x` and `y` have been *moved* (into the call to `longer_of`), and can no longer be used. Ending variable lifetimes early is part of Rust's strategy for ensuring memory safety without creating expensive copies at function call boundaries. These goals are difficult to reconcile, so it's perhaps not surprising that simple-looking code exposes language complexity.

Mutable value semantics (MVS) sits at third point in the design space where both goals are satisfied and mutation is supported, without the complexity inherent to flow-sensitive type systems. The key to this balance is simple: MVS does not surface references as a first-class concept in the programming model. As such, they can neither be assigned to a variable nor stored in object fields, and all values form disjoint topological trees rooted in the program's variables.

The reader may justifiably wonder whether a discipline with these restrictions is expressive enough to write efficient, non-trivial programs. We note that a large body of software projects across multiple languages already answer this question empirically, such as the Boost Graph Library (Siek et al. 2002), a collection of generic components (Stepanov & Rose 2014) for computations on graphs in C++, and Swift for TensorFlow (Saeta et al. 2021), a high-performance platform for machine learning. Further, we observe that well-established programming languages have adopted MVS at the core of their semantics, such as R (R Core Team 2020) and Swift (Apple Inc. 2021), for safety and/or efficiency.

In more detail, Swift is a modern general-purpose programming language, used in a broad spectrum of applications. Its standard library offers a rich collection of reusable components based on MVS, while striving to show competitive performance against comparable libraries in programming languages such as C, C++ and Rust. The language is translated to native code using an LLVM (Lattner & Adve 2004) backend.

After a brief introduction of the core tenets of MVS (Section 2), we explore some of these implementation strategies and make the following contributions:

- We propose a core language called Swiftlet, a subset of Swift focused exclusively on MVS. We introduce Swiftlet through a series of examples (Section 3) and formalize its semantics (Section 4).
- We discuss a compiler for Swiftlet that supports the creation of zero-cost abstractions (Section 5).
- We present a handful of compiler optimizations relying on local reasoning and leveraging runtime knowledge to elide unnecessary copies (Section 6).
- We report performance measurements on handwritten and randomly generated programs with varying numbers of mutating operations, comparing results between Swift, Swiftlet, Scala, and C++ to demonstrate the benefits of MVS over functional updates (Section 7).

2. Mutable value semantics

Before we delve deeper into the details of a language implementation, we ought to define precisely what MVS is and how it differs from the more widespread reference semantics.

2.1. Primitive and compound types

Popular object-oriented programming languages have converged on a common mutation model that distinguishes between so-called "primitive" or "built-in" types (typically numeric types) and "compound" types (typically arrays and classes). Variables of primitive types are independent: the value assigned to a variable of a primitive type cannot change due to an operation on another variable in the program. In contrast, variables of compound type may share state with other variables. Hence, the value assigned to a variable of a compound type *can* change due to an operation on another variable.

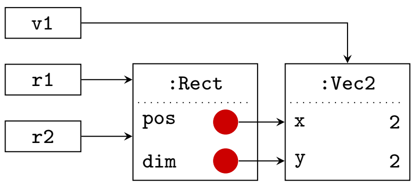
```java
 1  class Vec2 { int x, y; }       // Primitive fields
 2  class Rect { Vec2 pos, dim; } // Compound fields
 3
 4  int  i1 = 2;                  // Same pattern with
 5  Vec2 v1 = new Vec2(i1, i1);   // - int: primitive
 6  Rect r1 = new Rect(v1, v1);   // - Vec2: compound
 7  Rect r2 = r1;
 8
 9  r2.dim.x += 4                  // Mutates r2
10  System.out.println(r1.pos.x) // Now 6: r1 changed
11  System.out.println(r1.pos.y) // 2: no change
```

Listing 1 Compound types in Java have reference semantics

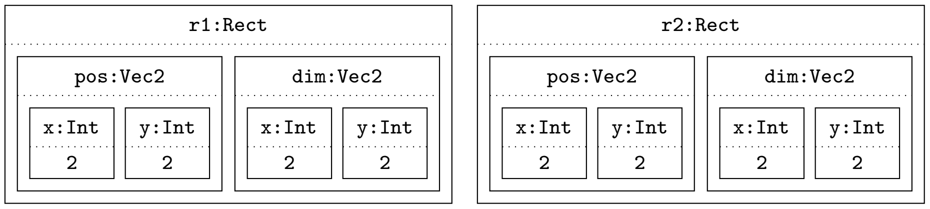
Consider the Java program above, which illustrates the distinction in full detail. In lines 1 and 2, it declares compound types `Vec2` and `Rect`, representing 2d vectors and rectangles, respectively. In line 5, both primitive-type fields of `v1` are initialized to the value of the same variable. In line 6, both compound-type fields of `r1` are initialized with `v1`, causing `r1.pos` to share state with `r1.dim`. In line 7, assignment causes `r1` to share state with `r2`. After line 7, the contents of the program's memory can be depicted as in Figure 1a.

Line 9 performs a mutation, growing the x dimension of `r2` by 4. Line 10 shows that the mutation has had a non-local effect, changing `r1.pos`, a distinct variable of compound type. Line 11 shows that, by contrast, the mutation has *not* changed the value of the field `r1.pos.y`, a distinct variable of primitive type, initialized in the same way.

This difference in behavior demonstrates that Java has two different kinds of mutation semantics: one for "primitive" types and another for "compound" types.



(a) With reference semantics



(b) With value semantics

Figure 1 Contents of the memory of a program involving compound types. Arrows represent references and boxes represent whole/part relationships.

2.2. Value and reference semantics

We can decouple these two mutation semantics from the question of whether a type is "primitive" or "compound". In fact, one could argue that it makes more sense for a notional vector value like `Vec2` to behave just like a scalar `int`. A more general distinction separates types with reference semantics, which behave like Java's compound types, from those with value semantics, which behave like Java's primitive types. Conceptually, two variables of a reference type can share mutable state, but two variables of a value type cannot.

Because the difference in behavior always involves mutation, an immutable type can be said to have value semantics trivially. Mutation by assigning a whole new value to a variable can be rewritten as binding a new variable, with no mutation, so is trivially equivalent. Therefore, to distinguish the nontrivial cases of interest, we say that a value type has *mutable* value semantics when its *parts* can be mutated in-place, without reassigning a variable of the type.

```
struct Vec2 { var x: Int, y: Int }
struct Rect { var pos: Vec2, dim: Vec2 }

var i1 = 2
var v1 = Vec2(x: i1, y: i1)
var r1 = Rect(pos: v1, dim: v1)
var r2 = r1

r2.dim.x += 4   // Mutates r2
print(r1.pos.x) // Prints 2: r1 unchanged
print(r1.pos.y) // Prints 2
```

Listing 2 Swift has compound types with value semantics

Consider the Swift program above, a direct transliteration of the Java code from Listing 1, but using only types with mutable value semantics. In Swift, structs are value types, so after line 6, `r1.pos` and `r1.dim` do not share state.

Figure 1b depicts the contents of the program's memory after line 7. Note that we use nesting rather than arrows to represents relationships between values and their parts, because values never share parts. Unlike in Java, the dimensions of `r2` are independent from those of `r1`. Hence, the mutation of `r2` at line 9 does not propagate to `r1`, as shown by the print statement at line 10.

2.3. Spooky action at a distance

Our Java example demonstrates how programming with reference types implicitly introduces *aliasing*—a condition where two or more live variables refer to the same memory location—every time a variable is passed to a function or assigned to another variable. Unfortunately, leaving alias creation implicit in the language creates a collection of problems (Noble et al. 1998) that we informally dub *spooky action at a distance*.[1] In short, neither humans nor machines (e.g., optimizing compilers) can reason locally about mutation semantics in the presence of aliases.

Consider the so-called "signing flaw", a security vulnerability discovered in a previous version of the Java platform that allowed untrusted applets to escalate access into the virtual machine (Vitek & Bokowski 2001). The vulnerability was caused by a reference leak, giving the attacker the means to mutate the system's internal list of signatures. The following snippet is a simplified excerpt of the flawed implementation:

```
public class Class {
  public Identity[] getSigners() {
    return this.signers;
  }
  private final Identity[] signers;
}
```

The field `signers` is exposed via the method `getSigners`. An attacker could thus obtain an alias on the list of trusted signers and alter it as they see fit. Although the field is `private final` and is thus neither accessible to clients nor reassignable, nothing prevents a method from accidentally leaking a reference to the object it holds, and through that reference, the list can be mutated (Potanin et al. 2013).

The standard prescription for accidental aliasing in Java is the manual insertion of defensive copies, but that is hardly an adequate cure: a missed defensive copy is a possible security vulnerability, an extra copy a source of inefficiency. Alias prevention mechanisms like ownership types (Clarke et al. 2013) and confined types (Vitek & Bokowski 2001)) are safer, but require complex annotations in code. Even when applied correctly, defensive copies must be made conservatively, without dynamic knowledge of how the objects will ultimately be used—for example, whether they are eventually mutated, or even inspected—and thus impose a heavy performance tax. Below the level of the programming model, the mere *possibility* of mutation through a shared reference creates additional costs (Shaikhha et al. 2017).

Optimizing compilers such as GCC and LLVM go to significant lengths to "model the heap" to prove references and pointers do not alias. In cases where uniqueness cannot be

[1] With apologies to Einstein.

proven, code generation becomes pessimistic. Examples of inhibited optimizations include: writing and reloading registers from memory, disabling loop-invariant code motion, and preventing vectorization.

By contrast, programming with value types rules out aliasing by construction, making it easy to reason about mutation and eliminating this class of bugs. Furthermore, a variable of value type can live exclusively in registers—even when it is compound, and across opaque function boundaries—allowing a compiler to reliably vectorize without modeling the heap. In other words, programming with values yields predictable performance without risking regression by changing code in a way that would cause optimizers to fall short, due to conservative assumptions.

2.4. The murky depths of ambiguous relationships

Implicit aliasing is not the only problem introduced by pervasive reference semantics: it has also led to a widespread fallacious mental model among programmers. Consider the routine question faced by object-oriented developers of whether copying, mutability, or comparison should be "deep" or "shallow" in the context of the following Java example:

```
interface ClickListener {
  void clickOccurred();
}

public class Button {
  private Point position;
  private ClickListener clickHandler;
}
```

It would be inappropriate to "shallow copy" a `Button`–a distinct copy of a `Button` instance needs a distinct copy of its `position`. If `Button` is "deep-copied", though, its `clickHandler` will be copied too, which is almost certainly inappropriate. Even if a "deep" copy *were* appropriate for the `clickHandler`, we'd have to ask, "how deep?" Since the details of the `clickHandler` are unknown, there is no answer. In fact, the exact same problem applies to both mutability and comparison: neither "deep" nor "shallow" will "cut it."

Each reference in this example represents a relationship between the `Button` and some other object. If we look closely at the nature of those relationships, we can see that there's something special about the `Button`'s relationship to its `position` that makes "deep" treatment appropriate: it connects a *whole* to its *part*.

While our initial example clearly demonstrates that the idea of "deep" or "shallow" copying is inadequate, recognizing the significance of whole-part relationships suggests that the idea itself is fallacious. Since a correct copy creates an independent but equivalent value of an instance *and all of its parts*, it would be fair to say there *are* no correct "deep" or "shallow" copies, only "copies".

This widespread misunderstanding would not exist but for the ambiguous meaning of stored references. It is striking, then, that despite the importance in UML of distinguishing "composition" and "aggregation" from mere "association", and despite UML's profound influence on object-oriented programming, the whole-part relationship is not surfaced by most object-oriented languages.

Aside from representing arbitrary relationships, stored references hav a second role: they *provide access to the related data*. In fact it is this access, combined with mutation, that leads to the "spooky action" discussed earlier, because when a reference is copied, access goes along with the relationship. It is interesting to ask, then, what would happen if we decoupled those roles? As it turns out, MVS does just that.

In MVS, composition *always* represents a whole-part relationship. This direct representation of composition benefits more than code clarity. The compiler is able to synthesize fundamental, tedious, and error-prone operations such as copying, equality and hashing. More importantly, it can automatically propagate immutability. This last capability has a powerful and non-obvious consequence: given a mutable type, application of a simple 'let' to a declared instance, produces a correct immutable instance. In reference-based languages such as Scala, Objective-C, and JavaScript, where the whole-part relationships are obscured, immutability is both more important—it is the only route to local reasoning—and much harder to achieve. In all of these languages it is common to see a separate immutable type created for every mutable one. (Odersky & Moors 2009; Bierema 2022).

2.5. Representing other relationships

Because whole/part relationships do not admit aliasing, they always form a tree, with the parts of a compound type being its children. It is reasonable to ask, then, how we can use mutable value types to represent self-referential data structures, such as doubly linked lists and directed graphs.

In fact, any arbitrary graph can be represented as an adjacency list. For example, a vertex set might be represented as an array, each element of which contains an array of outgoing edge destination indices. This approach can be seen as decoupling the two roles of first-class references: inner array elements represent relationships *without* conferring direct access to the related data, which is only available through the object of which it is a part.[2]

Naturally, losing the ability to directly access data through references changes the way programs are written. For example, traversing an arbitrary graph requires access to the whole graph at each step, rather than just a single vertex and its outgoing edges. In exchange, we get improved expressiveness, correctness, and even performance (Siek et al. 2002).

2.6. Semantic regularity and generic programming

Uniform mutation semantics makes it possible to create user-defined mathematical abstractions that behave like built-in numeric types. To illustrate, we can add a `+=` operator to the `Vec2` type introduced earlier.[3] Here, applying the mutating operator `+=` to `v1` affects only the value of `v1`, and not that of `v2`, just as if they were integers.

```
struct Vec2 {
  var x: Int, y: Int
```

[2] We note that the idea of dissociating the knowledge of a location from the right to access is central to capability-based approaches (Smith et al. 2000).

[3] The `inout` keyword seen here expresses argument mutation, and is explored in detail in Section 3.

```
  static func += (a: inout Self, b: Self) {
    a.x += b.x
    a.y += b.y
  }
}
var v1 = Vec2(x: 2, y: 2)
var v2 = v1
v2 += Vec2(x: 1, y: 0)
print(v1) // Vec2(x: 2, y: 2)
print(v2) // Vec2(x: 3, y: 2)
```

Semantic uniformity is a prerequisite for generic programming, the discipline of realizing algorithms and data structures so they work in the most general setting possible, without loss of efficiency (Stepanov & Rose 2014).[4] Indeed, it becomes difficult to even *describe* the semantics of an algorithm if any part of it can have non-local effects.

3. Swiftlet

Swiftlet is a subset of Swift, focusing on value types and discarding all features that are not essential to their description. Our language only features structs (i.e., compounds of heterogeneous types), arrays (i.e., dynamically sized collections of homogeneous data), functions, and numeric (integer and floating-point) values. The result is a language whose complete operational semantics fits a single page (Section 4).

A program is described by a sequence of `struct` declarations, followed by a single expression denoting an entry point (i.e., the contents of the main file in a regular Swift program).

A variable is declared with the keyword `var` followed by a name, an optional type annotation, an initial value, and the expression in which it is bound. A constant is declared similarly, with the keyword `let`. Naturally, a variable can be mutated or reassigned whereas a constant cannot.

```
var foo: Int = 4;
let bar = foo;
print(bar) // Prints 4
```

A `struct` is a compound type composed of zero or more fields. Each field is typed explicitly with an annotation and associated with a mutability qualifier (`let` or `var`) specifying whether it is constant or mutable. Fields can be of any type, but—for simplicity only—type definitions cannot be mutually recursive. Hence, all values have a finite representation.

```
struct Vec2 { ... };
var v = Vec2(x: 4, y: 2);
print(v.y) // Prints 2
```

As all types have value semantics, values form disjoint topological trees rooted at variables or constants. Conceptually, an assignment is always a copy of the right operand[5] and does not create an alias. In the program below, `u` is assigned a copy of `v`'s value, so the update of `u`'s second component in line 4 does not modify `v`.

```
1 struct Vec2 { ... };
2 var v = Vec2(x: 4, y: 2);
3 var u = v;
4 u.y = 8 // v = Vec2(x: 4, y: 2)
5         // u = Vec2(x: 4, y: 8)
```

[4] Generic programming as described by its originators depends on the concept of *regularity* (Stepanov & McJones 2009), a refinement of value semantics.

[5] We discuss how the language implementation eliminates unnecessary eager copies in Section 6.

Immutability applies transitively. All fields of a `struct` bound to a constant are also treated as immutable by the type system, regardless of their declaration. For example, the program below is ill-typed because `v.y` denotes a constant, notwithstanding that field having been declared with `var`.

```
struct Vec2 { ... };
let v = Vec2(x: 4, y: 2);
v.y = 8 // <- type error
```

Likewise, all elements of an array are constant if the array itself is bound to a constant.

```
struct Vec2 { ... };
let a = [Vec2(x: 4, y: 2), Vec2(x: 5, y: 3)];
a[0].y = 8 // <- type error
```

Functions are declared with the keyword `func` followed by a name, a list of typed parameters, a codomain, and a body. Arguments are evaluated eagerly and passed by value. Functions are allowed to be mutually recursive.

```
func fact(n: Int) -> Int {
  n > 1 ? n * fact(n: n - 1) : 1
};
fact(6) // Prints 720
```

To implement in-place part-wise mutation across function boundaries, a parameter's type is marked `inout`, which makes the parameter mutable in the callee. Conceptually, an `inout` argument is copied when a function is called and copied back when that function returns.[6]

```
1 struct Vec2 { ... };
2 func translateX(v: inout Vec2, d: Int) -> Void {
3   v.x = v.x + d
4 };
5 var v = Vec2(x: 4, y: 2);
6 _ = translateX(v: &v, d: 6);
7 print(v.x) // Prints 10
```

In the program above, the function `translateX` accepts an `inout` parameter of type `Vec2`, which it is allowed to mutate. The return type, `Void`, is Swiftlet's unit type. The function is called at line 6, effectively mutating the value of the vector `v` across function boundaries. Note that the ampersand featured in the call expression is not the "address-of" operator from C/C++. Instead, it signals in code that the argument is to be mutated—conceptually "copied out" of the callee upon return.

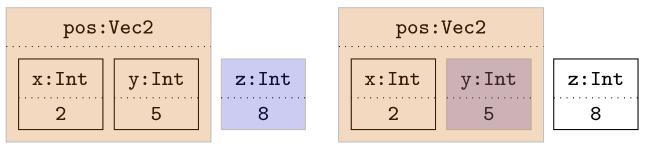
Of course, `inout` extends to multiple arguments, with one important restriction: to prevent any writeback from being discarded, overlapping mutations are prohibited. In other words, `inout` arguments must have independent values. This Law of Exclusivity (McCall 2017) creates a crucial optimization opportunity: it is *safe* to sidestep the conceptual copies by allowing the callee to write the argument's memory in the caller's context. In other words, `inout` argument passing can be implemented as pass-by-reference without surfacing reference semantics in the programming model.

Remark that `inout` parameters are reminiscent of (if not identical to) *borrowing* (Naden et al. 2012), as found in languages

[6] The Fortran enthusiast may think of the so-called "call-by-value/return" policy.

like Rust. The difference lies in the way uniqueness is guaranteed. Unlike generalized borrows, `inout` parameters are second-class citizens: they have lexically-bounded lifetimes and must "appear in person" (Strachey 2000). These restrictions ensure that aliases can be prevented simply by verifying that the "path" to a value (i.e., a list of member accesses and/or array subscripts) never appears twice in `inout` arguments to a single function call.[7] If references were granted first-class status, the type system would have to reason about the possible set of values that a variable may have at a particular point in the program, as a path alone would not be sufficient to identify the referred location.

One additional restriction applies to paths identifying elements of an array. The type system allows the same array to be indexed more than once by `inout` arguments only if it can conclude that the indices cannot overlap. For example, given an array `x`, the expression `swap(&x[0], &x[1])` is well-typed, but `swap(&x[f(0)], &x[1])` is not. In the second case, the type system conservatively assumes that `f(0)` could be evaluated as any value, including `1`.



(a) `swapX(a: &v, b: &z)` **(b)** `swapX(a: &v, b: &v.y)`

Figure 2 Visual representation of path uniqueness

We illustrate path uniqueness metaphorically. Imagine that values are represented by nested boxes, where nesting denotes a whole/part relationship. A path identifies a single box from outside in. Whenever one appears as an `inout` argument, the type checker paints the referred box with a color specific to that argument's position. At the end of the process, the program is ill-typed if one box had to be painted with two different colors.

We give an example in Figure 2. Let `swapX` be a function that accepts a vector and an integer as `inout` parameters—`a` and `b`—and swaps in-place the value of the vector's first component with that of the second parameter.

```
1 func swapX(a: inout Vec2, b: inout Int)
2   -> Void { ... };
3 var v = Vec2(x: 2, y: 5);
4 var z = 8;
5 swapX(a: &v, b: &z);  // <- well-typed
6 swapX(a: &v, b: &v.y) // <- ill-typed
```

Line 5 creates the situation shown in Figure 2a. No box had to be painted with two different colors: the call is well-typed. Line 6, however, produces Figure 2b. The box representing `v.y` was painted twice: the call is ill-typed.

A Swiftlet function must declare any variables that are to be *captured* in its closure.[8] If capture list is not provided, it is equivalent to a declaration of a zero-argument list with no captures. The value of each capture is then copied from the context surrounding its declaration, thus enforcing *capture-by-copy* semantics, and each capture is immutable.

```
var x = 2;
func f(y: inout Int) -> Void {
  [x] in      // x is captured immutably
  y += x;
};
f(y: &x);
print(x) // Prints 4
```

By contrast, Swift does not require captures to be explicitly declared, and its implicit captures are *by-reference*, so closures introduce reference semantics. Consider the following example, which is well-typed in Swift but not in Swiftlet:

```
var x = 0;
func g(y: inout Int) -> Void { y += x };
g(y: &x) // <- error
```

The above program creates overlapping mutable accesses to the same variable: the first obtained by capture, the second as an `inout` argument, which violates the Law of Exclusivity. This violation, however, is detected only at runtime. Since Swiftlet captures only by copy, it guarantees statically that closures uphold the Law of Exclusivity.

Drawing inspiration from linear type systems (Wadler 1990), languages like Rust capture free variables from the environment destructively. We refer to this policy as *capture-by-move*. Still others use capture-by-reference, but encode side effects on the environment into function types, essentially equipping the language with a type-and-effect system (Rytz et al. 2013). Both of these approaches to capture semantics introduce significant language complexity.

Swift methods are defined to be equivalent to free functions accepting an instance of the receiver `struct` as a first parameter. Therefore, without loss of expressivity, methods are omitted from Swiftlet for simplicity.

Although Swiftlet does not provide Swift's support for generic types—only arrays are generic—rudimentary polymorphism can be achieved via type-erased containers. An instance of type `Any` can store a value of any type, allowing the creation of type-erased data structures with value semantics.

```
1 struct Pair {
2   var _1: Any
3   var _2: Any
4 };
5
6 let p = Pair(_1: 4 as Any, _2: [7] as Any);
7 p._2 = p as Any; // Not a reference loop!
8
9 print(p) // Pair(_1: 4, _2: Pair(_1: 4, _2: [7]))
```

Line 1 declares `Pair`, a type that stores two values, each of arbitrary type. Line 6 creates `p`, a `Pair` storing an integer `4` as its first element and an array `[7]` as its second.[9] In line 7, `p._2` is assigned the value of `p`. Finally, line 9 prints the contents of `p`. Note that, because `Any` has value semantics, line 7 does not

[7] Note that, where `x` denotes an array of integers, a path of the form `x[x[0]]` is a valid `inout` argument, even if `x` occurs twice. Each subscript operation can be thought of as a function call, and in this case, the nested expression `x[0]` is reduced to an index `i` before the callee gets exclusive access to a path `x[i]`.

[8] Closures with explicit capture lists have a different syntax in Swift. We overlook that detail for simplicity.

[9] Unlike Swift, Swiftlet requires an explicit "`as`" cast to store an arbitrary value as `Any`. Further, the language does not provide a safe "downcast" from `Any`. An invalid cast results an error at runtime.

cause the pair to refer to itself, avoiding an infinite recursion in line 9. Instead, the value of `p` has been copied into `p._2`.

4. Formal definition

This section introduces Swiftlet formally. We start with its syntax and present a first description of its operational semantics in the form of big-step inference rules (a.k.a. natural semantics). This semantics is intended to describe the high-level user model and provide a formal framework for discussing optimization strategies.

Then, we present the Swiftlet's static semantics, and show how its type system guarantees uniqueness of `inout` arguments at function boundaries.

Although natural semantics is convenient for describing observable behaviors (Leroy & Grall 2009), its inability to distinguish failure from non-termination makes it less well-suited to the study of soundness properties. Hence, to demonstrate the guarantees provided by our static semantics, we finally present a second operational semantics in the form of small-step inference rules.

4.1. Notations

We use horizontal bar notation to denote sequences of terms. For instance, $\overline{x}$ expands to $x_1, \ldots, x_k$ for some k. We write ε for the empty sequence and, for the sake of syntactic regularity, we assume that $x_1, \ldots, x_k$ is an empty sequence if $k = 0$. We write $|\overline{x}|$ for the length of the sequence $\overline{x}$. We write $\overline{x : y}$ meaning $x_1 : y_1, \ldots, x_k : y_k$.

Let $f : A \to B$ be a function, $dom(f)$ denotes its domain. If f is a partial function, then $dom(f)$ is the subset $A' \subseteq A$ for which f is defined. We write $f = [\bot]_{A \to B}$ to represent a partial function $f : A \to B$ with $dom(f) = \varnothing$. We write $f = [a \mapsto b]_{A \to B}$ to represent a partial function f such that $f(a) = b$ with $dom(f) = \{a\}$. We write $f = [a \mapsto g(a) \mid p(a)]_{A \to B}$ for the function that returns $g(a)$ for all $a \in A$ that satisfy a predicate p. For example, $[i \mapsto -i \mid i \in \mathbb{Z} \wedge i < 0]_{\mathbb{Z} \to \mathbb{Z}}$ denotes a function that maps each negative integer to its absolute value. We omit the subscript when the function's domain and codomain are obvious from the context. We write $f[a \mapsto b]$ for the function that returns b for a and $f(x)$ for any other argument. 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 write $f[a \mapsto \bot]$ for the function that is not defined for a and returns $f(x)$ for any other argument. Given $f : A \to B$ and $g : A \to B$, we write $f[a \mapsto_? g(a)]$ for the function $f[a \mapsto g(a)]$ if $a \in dom(g)$, or f otherwise.

Let e be a term and σ a set of substitutions represented as a partial function from variables to terms, we write $e[/\sigma]$ for the term obtained by applying the substitutions σ to e, renaming free variables as necessary. For instance, if $e = \lambda a.ab$ and $\sigma = [b \mapsto c]$, then $e[/\sigma] = \lambda a.ac$.

4.2. Syntax

Figure 3 presents the formal syntax of Swiftlet. A program g is a sequence of structure declarations followed by a single functional term acting as its entry point.[10] A structure is described by a unique global name and a sequence of property declarations. A property is declared by a binding $m\ x : \tau$ where m denotes its mutability, x identifies its name, and τ specifies its type.

$$
\begin{array}{rlcl}
\textit{integer} & c & & \\
\textit{name} & x, s & & \\
\textit{context} & \mu, \phi^s & : & X \to M \times V \\
\textit{prog.} & g & ::= & \overline{d}\ e \\
\textit{struct} & d & ::= & \texttt{struct}\ s\ \{\ \overline{b}\ \}; \\
\textit{qual.} & m & ::= & \texttt{let} \mid \texttt{var} \\
\textit{bind.} & b & ::= & m\ x : \tau \\
\textit{arg.} & a & ::= & \&r \mid e \\
\textit{expr.} & e & ::= & e;e \mid b = e\ \texttt{in}\ e \mid r = e \mid [\overline{e}] \mid r \mid v \\
 & & \mid & s(\overline{e}) \mid e(\overline{a}) \mid e\ ?\ e : e \mid e\ \texttt{as}\ \tau \\
 & & \mid & \texttt{func}\ x\ (\overline{x : p}) \to \tau\ \{[\overline{x}]\ \texttt{in}\ e\}\ \texttt{in}\ e \\
\textit{param.} & p & ::= & \texttt{inout}\ \tau \mid \tau \\
\textit{type} & \tau & ::= & (\overline{p}) \to \tau \mid [\tau] \mid s \mid \mathbb{Z} \mid \textit{Any} \mid () \\
\textit{path} & r & ::= & e.x \mid e[e] \mid w \\
\textit{value} & v & ::= & \lambda(\overline{x : p}, e) \mid \phi^s \mid [\overline{v}] \mid \textit{box}(v) \mid c \\
\textit{lvalue} & w & ::= & w.x \mid w[c] \mid x
\end{array}
$$

Figure 3 Formal syntax of Swiftlet

Other types include integer (written $\mathbb{Z}$)[11], homogeneous arrays (written $[\tau]$ where τ is the element type), function types (written $(\overline{p}) \to \tau$, where τ is the return type and each p is a parameter type potentially qualified by `inout`), the existential container type (written *Any*), and the unit type (written $()$).

Functions can be recursive (although not hoisted), but we proscribe mutually recursive type declarations. For the sake of simplicity, Swiftlet requires all named declarations (i.e., structures, properties, parameters, and local bindings) to have a unique name. This simplification does not restrict the expressiveness of our language, as name conflicts can always be eliminated via α-conversion. Further, function declarations always feature a capture list, even when it is empty.

Expressions are composed out of array literals, structure instantiations, function calls, conditionals, function declarations, binding declarations, assignments, sequences, casts, values, and paths. The latter are at the heart of mutable value semantics. In broad strokes, a path denotes access to a value or part thereof. It can be the name of a binding or any expression suffixed by either a dotted accessor (e.g., $e.n$) or a bracketed index identifying a specific element in an array (e.g., $e_1[e_2]$).

Borrowing from C parlance, path expressions starting with a name are called *lvalues*, as they may appear on the left hand side of an assignment. Only mutable lvalues can serve as ar-

[10] Named functions are declared in the body of the entry point

[11] We exclude floating-point values from the formal definition.

guments to `inout` parameters. As mentioned in the previous section, immutability applies transitively, meaning that an lvalue is immutable if any component of its path is. Note: a bracketed lvalue can be mutable even when the expression of its index is immutable. Because 0 without enclosing brackets is not a path component, its immutability does affect that of $x[0]$, which is only immutable if x is an immutable binding.

The sequence operator ";" is left associative, that is $a; b; c$ is equivalent to $(a; b); c$. For clarity, the scope of a declaration is introduced explicitly in the formal syntax: binding and function declarations are always trailed by another expression, which represents their scope. For instance, x in the assignment $x = 2$ is bound in $\texttt{var}\ x : \mathbb{Z} = 1\ \texttt{in}\ (f(x); x = 2)$, but it is free in both $x = 2; \texttt{var}\ y : \mathbb{Z} = 1\ \texttt{in}\ f(x)$ and $\texttt{var}\ y : \mathbb{Z} = 1\ \texttt{in}\ f(y); x = 2$. Scopes are used to determine the lifetime of a particular value and reclaim memory.

We bring the reader's attention to a handful of additional differences between Swiftlet's concrete and formal syntax. First, for the sake of concision, the formal syntax does not use argument labels in function calls or structure literals. For instance, a call `f(y: &x)` in the concrete syntax is written $f(\&x)$ formally. Second, the formal syntax lets bindings appear at any position in an expression. For instance, $\texttt{let}\ x = \texttt{let}\ y = 1\ \texttt{in}\ y\ \texttt{in}\ f(x)$ is formally valid, yet it is ill-formed in the concrete syntax. More generally, any expression can appear in a binding's initializer, including assignments. This difference, however, does not raise the expressiveness of the formal syntax above that of its concrete counterpart, as side-effectful expressions can be represented by closures with `inout` arguments.

4.3. Natural semantics

Let X be the set of local names represented by the metasyntactic variable x and V the set of values represented by the metasyntactic variable v. Let $M = \{\texttt{let}, \texttt{var}\}$ be the set of mutability qualifiers. A context μ is a partial function $X \to M \times V$ mapping each local name to its mutability anb value.

Figure 4 presents the natural semantics of Swiftlet with two judgments. The first $(\Delta, \mu \vdash e \Downarrow^R \mu', v)$ reduces an expression e to a value v, where Δ is a set of structure declarations, μ maps the bindings in scope to their respective value and mutability, and μ' records the side effects of the evaluation. The second judgment $(\Delta, \mu \vdash r \Downarrow^L \mu', m\ w)$ operates similarly for reducing a path r to an lvalue w, qualified by a mutability m. The evaluation of a program g starts with a set Δ populated with the structure declarations defined in g and an empty context $\mu = [\bot]$. It either concludes with a final value v, or never terminates, or fails because of a runtime error, such as an invalid cast or an out-of-bound array access.

Because the purpose of this semantics is to understand the high-level behavior of Swiftlet, we purposely do not model all possible typing errors in the natural semantics.[12] Nonetheless, we represent path mutability because that property is leveraged in Section 6 to implement program optimizations.

Bindings Binding declarations are reduced by E-BINDING, which first evaluates the initializer expression e_1. The resulting value is copied and used to extend the evaluation context with a new binding before evaluating the binding's scope expression e_2. The binding is eventually removed from the evaluation context, encoding the fact that the binding goes out of scope.

Copies are expressed explicitly by calls to a helper function *copy*. At an abstract level, *copy* is equivalent to the identity (i.e., $\forall v, copy(v) = v$). In a concrete implementation, however, it models the operations required for cloning a value.[13]

Structures and arrays Structure literals are reduced by E-STRUCTLIT, which starts by evaluating the constructor's arguments. Notice that an argument e_i is reduced in the context μ_{i-1} that results from the reduction of its predecessor, from left to right. Once all arguments have been evaluated, a new instance is built as a partial function ϕ that maps each property to a copy of its corresponding value, using the declaration stored in Δ to determine each property's mutability.

Example 4.1. Let $\Delta = \{\texttt{struct}\ A\{\texttt{var}\ x : \tau, \texttt{let}\ y : \tau\}\}$ and μ be an arbitrary context. The evaluation of $A(3, 4)$ in Δ, μ results in a value $\phi^A = [x \mapsto \texttt{var}\ 3, y \mapsto \texttt{let}\ 4]$.

The rule E-ARRAYLIT operates similarly for array instances. The value of each element is evaluated, from left to right, to build a sequence $[v_1, \ldots, v_k]$ representing the array. Unlike structure properties, array elements do not require separate mutability tracking because an element's mutability is simply that of the array.

Paths Paths can be reduced as either values or lvalues, depending on their position in an expression. The rules E-NAME, E-PROP, and E-ELEM describe reduction as values and are straightforward: E-NAME looks up for a specific binding in the evaluation context, E-PROP reduces the base of the path as structure and looks up a specific property, and E-ELEM reduces both the base and index of the path to select a specific element in an array instance.

Reduction as lvalues is similarly described by the rules P-NAME, P-PROP, and P-ELEM, with one difference. Recall that lvalues are reduced along with their mutability, as defined by the judgment for $\Downarrow^L$, so that our semantics can model immutability violations. The rule P-PROP reads mutability information from the context μ by the means of a helper function *get*:

$$\begin{aligned} get(\mu, x) &= \mu(x) \\ get(\mu, w.x) &= get(\mu, w)(x) \\ get(\mu, w[c]) &= get(\mu, w)_c \end{aligned}$$

The mutability of a property depends on both its declaration and on the mutability of the containing instance. Formally, that is represented by $min(m, m')$ in the premises of the rules, where $\texttt{let} < \texttt{var}$.

Assignments The rule E-ASSIGN first reduces the path on the left hand side as a mutable lvalue, then reduces the right hand side as a value. Since both evaluations can introduce side effects, the context μ is threaded from one to the other.

[12] We present static semantics in the next section.

[13] A formal definition is presented in Section 4.5.

$$\boxed{\Delta, \mu \vdash e \Downarrow^R \mu', v}$$

$$\textsc{E-Name}\quad \frac{\mu(x) = m\ v}{\Delta, \mu \vdash x \Downarrow^R \mu, v}
\qquad
\textsc{E-Prop}\quad \frac{\Delta, \mu \vdash e \Downarrow^R \mu', \phi^s \qquad \phi^s(x) = m\ v}{\Delta, \mu \vdash e.x \Downarrow^R \mu', v}$$

$$\textsc{E-Elem}\quad \frac{\begin{array}{c}\Delta, \mu \vdash e_1 \Downarrow^R \mu', [v_1, \dots, v_k] \\ \Delta, \mu' \vdash e_2 \Downarrow^R \mu'', c \qquad 0 \le c < k\end{array}}{\Delta, \mu \vdash e_1[e_2] \Downarrow^R \mu'', v_{c+1}}
\qquad
\textsc{E-Inout}\quad \frac{\Delta, \mu \vdash r \Downarrow^L \mu', \texttt{var}\ w}{\Delta, \mu \vdash \&r \Downarrow^R \mu', w}$$

$$\textsc{E-Binding}\quad \frac{\Delta, \mu \vdash e_1 \Downarrow^R \mu', v_1 \qquad \Delta, \mu'[x \mapsto m\ copy(v_1)] \vdash e_2 \Downarrow^R \mu'', v_2}{\Delta, \mu \vdash m\ x : \tau = e_1\ \texttt{in}\ e_2 \Downarrow^R \mu''[x \mapsto_? \mu'(x)], v_2}$$

$$\textsc{E-Assign}\quad \frac{\Delta, \mu \vdash e \Downarrow^R \mu', v \qquad \Delta, \mu' \vdash r \Downarrow^L \mu'', \texttt{var}\ w}{\Delta, \mu \vdash r = e \Downarrow^R set(\mu'', w, copy(v)), [\bot]}$$

$$\textsc{E-StructLit}\quad \frac{\begin{array}{c}\overbrace{\Delta, \mu_{i-1} \vdash e_i \Downarrow^R \mu_i, v_i}^{1 \le i \le k} \\ \texttt{struct}\ s\ \{m_1\ x_1 : \tau_1, \dots, m_k\ x_k : \tau_k\} \in \Delta\end{array}}{\Delta, \mu_0 \vdash s(e_1, \dots, e_k) \Downarrow^R \mu_k, [x_i \mapsto m_i\ copy(v_i) \mid 1 \le i \le k]^s}$$

$$\textsc{E-ArrayLit}\quad \frac{\overbrace{\Delta, \mu_{i-1} \vdash e_i \Downarrow^R \mu_i, v_i}^{1 \le i \le k}}{\Delta, \mu_0 \vdash [e_1, \dots, e_k] \Downarrow^R \mu_k, [copy(v_1), \dots, copy(v_k)]}$$

$$\textsc{E-Func}\quad \frac{\begin{array}{c}e_1' = e_1[/\sigma] \qquad \Delta, \mu[x_0 \mapsto \texttt{let}\ \lambda(x_1 : p_1, \dots, x_k : p_k, e_1'[/\sigma'])] \vdash e_2 \Downarrow^R \mu', v \\ \sigma = [y_j \mapsto copy(v_j) \mid (1 \le j \le h) \wedge \mu(y_j) = m_j\ v_j] \qquad \sigma' = [x_0 \mapsto \texttt{func}\ x_0\ (x_1 : p_1, \dots x_k : p_k) \to \tau\ \{[]\ \texttt{in}\ e_1'\}\ \texttt{in}\ x_0]\end{array}}{\Delta, \mu \vdash \texttt{func}\ x_0\ (x_1 : p_1, \dots x_k : p_k) \to \tau\ \{[y_1, \dots, y_h]\ \texttt{in}\ e_1\}\ \texttt{in}\ e_2 \Downarrow^R \mu'[x_0 \mapsto_? \mu(x_0)], v}$$

$$\textsc{E-Call}\quad \frac{\begin{array}{c}\Delta, \mu \vdash e_0 \Downarrow^R \mu_0, \lambda(x_1 : p_1, \dots, x_k : p_k, e_b) \qquad \overbrace{\Delta, \mu_{i-1} \vdash a_i \Downarrow^R \mu_i, v_i}^{1 \le i \le k} \\ \sigma = [x_i \mapsto copy(v_i) \mid 1 \le i \le k] \qquad \Delta, \mu_k \vdash e_b[/\sigma] \Downarrow^R \mu', v\end{array}}{\Delta, \mu \vdash e_0(a_1, \dots, a_k) \Downarrow^R \mu', copy(v)}
\qquad
\textsc{E-Seq}\quad \frac{\Delta, \mu \vdash e_1 \Downarrow^R \mu', v_1 \qquad \Delta, \mu' \vdash e_2 \Downarrow^R \mu'', v_2}{\Delta, \mu \vdash e_1; e_2 \Downarrow^R \mu'', v_2}$$

$$\textsc{E-Cond-T}\quad \frac{\begin{array}{c}\Delta, \mu \vdash e_1 \Downarrow^R \mu', v_1 \qquad v_1 \ne 0 \\ \Delta, \mu' \vdash e_2 \Downarrow^R \mu'', v_2\end{array}}{\Delta, \mu \vdash e_1\ ?\ e_2 : e_3 \Downarrow^R \mu'', v_2}
\qquad
\textsc{E-Cond-F}\quad \frac{\begin{array}{c}\Delta, \mu \vdash e_1 \Downarrow^R \mu', 0 \\ \Delta, \mu' \vdash e_3 \Downarrow^R \mu'', v_3\end{array}}{\Delta, \mu \vdash e_1\ ?\ e_2 : e_3 \Downarrow^R \mu'', v_3}$$

$$\textsc{E-Upcast}\quad \frac{\begin{array}{c}\Delta, \mu \vdash e \Downarrow^R \mu', v \\ typeof(v) \ne Any\end{array}}{\Delta, \mu \vdash e\ \texttt{as}\ Any \Downarrow^R \mu', box(v)}
\qquad
\textsc{E-Downcast}\quad \frac{\begin{array}{c}\Delta, \mu \vdash e \Downarrow^R \mu', box(v) \\ typeof(v) = \tau\end{array}}{\Delta, \mu \vdash e\ \texttt{as}\ \tau \Downarrow^R \mu', v}$$

$$\boxed{\Delta, \mu \vdash r \Downarrow^L \mu', m\ w}$$

$$\textsc{P-Name}\quad \frac{\mu(x) = m\ v}{\Delta, \mu \vdash x \Downarrow^L \mu, m\ x}
\qquad
\textsc{P-Prop}\quad \frac{\begin{array}{c}\Delta, \mu \vdash r \Downarrow^L \mu', m\ w \qquad get(\mu', w) = m_w\ \phi^s \\ \phi^s(x) = m_x\ v_x \qquad m' = min(m_w, m_x)\end{array}}{\Delta, \mu \vdash r.x \Downarrow^L \mu', m'\ w.x}$$

$$\textsc{P-Elem}\quad \frac{\begin{array}{c}\Delta, \mu \vdash r \Downarrow^L \mu', m\ w \qquad get(\mu', w) = m_w\ [v_1, \dots, v_k] \\ \Delta, \mu' \vdash e \Downarrow^R \mu'', c \qquad 0 \le c < k\end{array}}{\Delta, \mu \vdash r[e] \Downarrow^L \mu'', m\ w[c+1]}$$

Figure 4 Natural semantics of Swiftlet

Finally, the mutation is performed by calling a helper function *set*:

$$\begin{aligned}
set(\mu, x, v) &= \mu[x \mapsto v] \\
set(\mu, w.x, v) &= set(\mu, w, get(\mu, w)[x \mapsto v]) \\
set(\mu, w[c], v) &= set(\mu, w, [u_1, \dots, u_{c-1}, v, u_{c+1}, \dots, u_k]) \\
&\quad \text{where } get(\mu, w) = [u_1, \dots, u_k]
\end{aligned}$$

The resemblance of *set* to a functional update suggests that there is a simple mapping from a program using MVS to one that is purely functional. Unlike its rendition after such a purely-functional transformation, our *set* has predictable performance: it can always be implemented as an in-place update without any optimizer heroics, since paths are known to always identify unique and independent values.

Closures Function declarations are handled by E-Func. First, it creates a substitution σ as a table mapping captured bindings to their value in the context μ. These bindings are identified explicitly using the capture list.

As mentioned in Section 3, recall that captures are immutable. Hence, they can be substituted for their values directly, thus encoding the function's environment syntactically.[14]

To support recursive calls, the rule builds another substitution σ' to map the name of the declared function to its own declaration, or more precisely, to an expression that evaluates to the same function. This strategy is reminiscent of the typical formalization of *letrec* in call-by-value λ-calculi (Reynolds 1998b, Chapter 11). In a nutshell, the rule unfolds the recursive definition *once* and rewrites the function's body so that further unfoldings are carried out when a recursive call is evaluated. Notice, however, that the rewritten function has no captures. Any captures in the original are substituted during the first unfolding, so that the declaration cannot accidentally rebind a capture to another value.

The second substitution σ' is applied to the function's body to bind the function's name in the context μ and evaluate the expression e_2, representing the scope in which the function is defined. That binding is finally removed from the context in the rule's conclusion, effectively ending its lifetime.

Example 4.2 (Recursive function)**.** Let a be a term denoting a recursive function declaration:

$$a = \texttt{func}\ f(n : \mathbb{Z}) \to \mathbb{Z}\{[]\ \texttt{in}\ n > 1\ ?\ n * f(n-1)\ :\ 1\}$$

Next, consider the expression $a\ \texttt{in}\ f(2)$, which declares the function f and immediately applies it to an integer argument. That expression is evaluated by E-FUNC, which creates a substitution mapping f to an expression $a\ \texttt{in}\ f$. This substitution is applied to the body of the function, resulting in a closure $\lambda(n : \mathbb{Z}, (n > 1\ ?\ n * (a\ \texttt{in}\ f)(n-1)\ :\ 1))$.

In a call, if n is greater than one, the reduction of the first branch of the conditional will trigger E-FUNC to evaluate $a\ \texttt{in}\ f$, effectively unfolding the recursive declaration one more time. Otherwise, the second branch of the conditional will reduce immediately as the value 1, ending recursion.

Function calls The rule E-CALL describes function calls. The callee is evaluated first and must reduce to a closure of the form $\lambda(\overline{x : p}, e_b)$. Arguments are evaluated next, from left to right, just as in E-STRUCTLIT and E-ARRAYLIT. The value of each argument is then substituted for the corresponding parameter name in the function's body.

The call's `inout` arguments are handled by inlining the lvalue to which they reduce in the function's body. Indeed, notice that E-INOUT evaluates the path following the ampersand as an lvalue rather than a value, using $\Downarrow^L$ rather than $\Downarrow^R$.

Example 4.3. Let $f(\&a[a[0]].b)$ be a function call evaluated by E-CALL, in a context $\mu = [f \mapsto \texttt{let}\ \lambda(x : \texttt{inout}\ \mathbb{Z}, x = 42), a \mapsto \texttt{var}\ [0,1]]$. The rule starts by reducing the callee, by direct application of the E-NAME. Then, the `inout` argument is handled by E-INOUT, triggering the application of P-ELEM that eventually produces a mutable lvalue $\texttt{var}\ a[1]$. The latter is inlined in the closure's body, resulting in an expression $a[1] = 42$ that is evaluated in μ. The call finally concludes with an updated context $\mu' = [f \mapsto \texttt{let}\ \lambda(x : \texttt{inout}\ \mathbb{Z}, x = 42), a \mapsto \texttt{var}\ [0,42]]$.

[14] Alternatively, one could define a closure as a term $\lambda^{\mu}(\overline{x : p}.e)$ where μ would represent the environment. Such a strategy would let us represent mutable (yet copied) captures.

Casts The rules E-UPCAST and E-DOWNCAST describe casts. Upcasting a value wraps it in an existential container (Pierce 2002, Chapter 24), represented formally as a value $box(v)$. Conversely, downcasting unwraps a container to extract its wrapped value.

While upcasting always succeeds, downcasting fails unless the value has the expected dynamic type. The dynamic type is retrieved by a helper *typeof*, which returns the type encoded into the low-level representation of existential containers.

4.4. Static semantics

Figure 5 presents the typing semantics of Swiftlet. Four typing judgments are defined, relating to programs, function arguments, paths and other expressions.

The program judgment $\vdash g : \tau$ initiates type checking, creating a set Δ of structure declarations. The program's entry point is then typed with a judgment $\Delta; \Gamma \vdash e : \tau$, stating that the expression e has type τ in the context of Δ and Γ. Most rules are straightforward except for T-FUNC and T-CALL, which check function declarations and function calls, respectively.

Function declarations Function declarations are typed in two steps. The first creates a new context for type checking the body expression. The second consists of type checking the expression delimiting the scope of the declaration.

The rule T-FUNC starts by mapping each capture to its corresponding type, treating them as immutable regardless of their mutability in the surrounding context. Then, each parameter is mapped onto its type and mutability, resulting in a typing context Γ''. Parameters are immutable unless qualified by `inout`. This translation is expressed by a small helper function:

$$type(x : p) = \begin{cases} \texttt{let}\ \tau & \text{if } p = \tau \\ \texttt{var}\ \tau & \text{if } p = \texttt{inout}\ \tau \end{cases}$$

The context Γ'' is finally extended by mapping the function's name onto its own type before type checking the body e_1 in order to handle recursive calls. The context Γ is extended similarly to type check the expression e_2 in which the newly declared function is visible. In both instances, the binding representing the function is considered immutable.

Function calls In function calls, the type system upholds uniqueness of `inout` parameters by guaranteeing that the same lvalue cannot be dereferenced from two different paths. This test is defined via a relation $\subseteq$ on argument expressions. Intuitively, $a_i \subseteq a_j$ holds if both expressions are `inout` arguments (i.e., paths prefixed by &) whose paths are either identical, or a_i's is a subpath of a_j's. Formally, $\subseteq$ is the minimal reflexive and transitive relation that satisfies the following rules:[15]

[15] The intuition of the operator relates to the size of the path rather than the set of locations that it represents.

$\boxed{\vdash g : \tau}$

$$\textsc{T-Program}\quad \frac{\{d_1,\dots,d_k\},[\bot] \vdash e : \tau}{\vdash d_1;\dots;d_k;e : \tau}$$

$\boxed{\Delta;\Gamma \vdash_{arg} a : p}$

$$\textsc{T-Arg}\quad \frac{\Delta;\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash_{arg} e : \tau} \qquad \textsc{T-Inout}\quad \frac{\Delta;\Gamma \vdash_{path} r : \texttt{var}\ \tau}{\Delta;\Gamma \vdash_{arg} \&r : \texttt{inout}\ \tau}$$

$\boxed{\Delta;\Gamma \vdash e : \tau}$

$$\textsc{T-Const}\quad \frac{}{\Delta;\Gamma \vdash c : \mathbb{Z}} \qquad \textsc{T-Binding}\quad \frac{\begin{gathered}\Delta;\Gamma \vdash e_x : \tau_x \\ \Delta;\Gamma[x \mapsto m\ \tau_x] \vdash e : \tau\end{gathered}}{\Delta;\Gamma \vdash m\ x : \tau_x = e_x\ \texttt{in}\ e : \tau}$$

$$\textsc{T-Read}\quad \frac{\Delta;\Gamma \vdash_{path} r : m\ \tau}{\Delta;\Gamma \vdash r : \tau} \qquad \textsc{T-Assign}\quad \frac{\begin{gathered}\Delta;\Gamma \vdash e_r : \tau \\ \Delta;\Gamma \vdash_{path} r : \texttt{var}\ \tau\end{gathered}}{\Delta;\Gamma \vdash r = e_r : ()} \qquad \textsc{T-Seq}\quad \frac{\begin{gathered}\Delta;\Gamma \vdash e_1 : \tau_1 \\ \Delta;\Gamma \vdash e_2 : \tau_2\end{gathered}}{\Delta;\Gamma \vdash e_1;e_2 : \tau_2}$$

$$\textsc{T-StructLit}\quad \frac{\begin{gathered}\overbrace{\Delta;\Gamma \vdash e_i : \tau_i}^{1\le i\le k} \\ \texttt{struct}\ s\ \{m_1\ x_1 : \tau_1,\dots,m_k\ x_k : \tau_k\}\ \in \Delta\end{gathered}}{\Delta;\Gamma \vdash s(e_1,\dots,e_k) : s} \qquad \textsc{T-ArrayLit}\quad \frac{\overbrace{\Gamma,\Delta \vdash e_i : \tau}^{1\le i\le k}}{\Delta;\Gamma \vdash [e_1,\dots,e_k] : [\tau]}$$

$$\textsc{T-Cond}\quad \frac{\begin{gathered}\Delta;\Gamma \vdash e : \mathbb{Z} \\ \Delta;\Gamma \vdash e_t : \tau \qquad \Delta;\Gamma \vdash e_e : \tau\end{gathered}}{\Delta;\Gamma \vdash e\ ?\ e_t\ :\ e_e : \tau} \qquad \textsc{T-Cast}\quad \frac{\Delta;\Gamma \vdash e : \tau'}{\Delta;\Gamma \vdash e\ \texttt{as}\ \tau : \tau}$$

$$\textsc{T-Func}\quad \frac{\begin{gathered}\Gamma' = [y_j \mapsto \texttt{let}\ \tau_j \mid 1 \le j \le h \wedge \Gamma(y_j) = m_j\ \tau_j] \qquad \Gamma'' = \Gamma'[x_i \mapsto type(p_i) \mid 1 \le i \le k] \\ \Delta,\Gamma''[x \mapsto \texttt{let}\ (p_1,\dots,p_k) \to \tau] \vdash e_1 : \tau_\lambda \qquad \Delta;\Gamma[x \mapsto \texttt{let}\ (p_1,\dots,p_k) \to \tau] \vdash e_2 : \tau\end{gathered}}{\Delta;\Gamma \vdash \texttt{func}\ x(x_1 : p_1,\dots,x_k : p_k) \to \tau_\lambda\ \{[y_1,\dots,y_h]\ \texttt{in}\ e_1\}\ \texttt{in}\ e_2 : \tau}$$

$$\textsc{T-Call}\quad \frac{\Delta;\Gamma \vdash e : (p_1,\dots,p_k) \to \tau \qquad \overbrace{\Delta;\Gamma \vdash_{arg} a_i : p_i}^{1\le i\le k} \qquad \forall_{1\le i\le k}, \forall_{1\le j\le k}, i \ne j \implies a_i \not\subseteq a_j}{\Delta;\Gamma \vdash e(a_1,\dots,a_k) : \tau}$$

$\boxed{\Delta;\Gamma \vdash_{path} r : m\ \tau}$

$$\textsc{T-BindingRef}\quad \frac{\Gamma(x) = m\ x}{\Delta;\Gamma \vdash_{path} x : m\ \tau} \qquad \textsc{T-LetElemRef}\quad \frac{\Delta;\Gamma \vdash e : [\tau] \qquad \Delta;\Gamma \vdash e_c : \mathbb{Z}}{\Delta;\Gamma \vdash_{path} e[e_c] : \texttt{let}\ \tau}$$

$$\textsc{T-VarElemRef}\quad \frac{\Delta;\Gamma \vdash_{path} r : \texttt{var}\ [\tau] \qquad \Delta;\Gamma \vdash e_c : \mathbb{Z}}{\Delta;\Gamma \vdash_{path} r[e_c] : \texttt{var}\ \tau}$$

$$\textsc{T-LetPropRef}\quad \frac{\begin{gathered}\Delta;\Gamma \vdash e : s \\ \texttt{struct}\ s\ \{m_1\ x_1 : \tau_1,\dots,m_k\ x_k : \tau_k\}\ \in \Delta\end{gathered}}{\Delta;\Gamma \vdash_{path} e.x_i : \texttt{let}\ \tau_i} \qquad \textsc{T-VarPropRef}\quad \frac{\begin{gathered}\Delta;\Gamma \vdash_{path} r : \texttt{var}\ s \qquad m_i = \texttt{var} \\ \texttt{struct}\ s\ \{m_1\ x_1 : \tau_1,\dots,m_k\ x_k : \tau_k\}\ \in \Delta\end{gathered}}{\Delta;\Gamma \vdash_{path} r.x_i : \texttt{var}\ \tau_i}$$

Figure 5 Typing semantics.

$$\frac{}{\&r \subseteq \&r} \qquad \frac{\&r \subseteq \&r'}{\&r \subseteq \&r'.n} \qquad \frac{\&r \subseteq \&r'}{\&r \subseteq \&r'[e]}$$

$$\frac{\neg const(e) \vee \neg const(e')}{\&r[e] \subseteq \&r[e']}$$

The last rule applies when the value of an array index is not statically computable. It represents the restriction that fends off cases where two arbitrary expressions would evaluate to the same value, effectively producing two identical paths. For simplicity, we assume here that $const(e)$ holds if and only if e is a constant. This choice is conservative and suffers from obvious false-positives (e.g., $x[0] \subseteq x[0+1]$ while 0 and $0+1$ certainly denote different paths). The *const* predicate is, however, amenable to higher degrees of precision and a more sophisticate definition (e.g., akin to C++11's `constexpr` specifier) could include a wider range of expressions.

Casts Notice that T-Cast does not perform any test to guarantee that the value e is indeed of type τ. Indeed, casts in Swiftlet are completely dynamic and, therefore, errors are handled at runtime. In other words, the correctness of cast a expression is not guaranteed statically and, therefore, is excluded from Swiftlet's definition of (static) type safety.

4.5. Small-step semantics

$$
\begin{array}{rlll}
\textit{integer} & c & & \\
\textit{loc.} & l & & \\
\textit{name} & x, s & & \\
\textit{ptr} & \pi & : & L \rightarrow M \times V \\
\textit{mem} & \eta & : & X \rightarrow V \\
\textit{prog.} & g & ::= & \overline{d}\ e \\
\textit{struct} & d & ::= & \texttt{struct}\ s\ \{\ \overline{b}\ \}; \\
\textit{qual.} & m & ::= & \texttt{let} \mid \texttt{var} \\
\textit{bind.} & b & ::= & m\ x : \tau \\
\textit{arg.} & a & ::= & \&r \mid e \\
\textit{expr.} & e & ::= & e; e \mid b = e\ \texttt{in}\ e \mid r = e \mid [\overline{e}] \mid r \mid v \\
 & & \mid & s(\overline{e}) \mid e(\overline{a}) \mid e\ ?\ e : e \mid e\ \texttt{as}\ \tau \\
 & & \mid & \texttt{func}\ x\ (\overline{x : p}) \rightarrow \tau\ \{[\overline{x}]\ \texttt{in}\ e\}\ \texttt{in}\ e \\
 & & \mid & e; \texttt{pop}\ \overline{l} \\
\textit{param.} & p & ::= & \texttt{inout}\ \tau \mid \tau \\
\textit{type} & \tau & ::= & (\overline{p}) \rightarrow \tau \mid [\tau] \mid s \mid \mathbb{Z} \mid \textit{Any} \mid () \\
\textit{path} & r & ::= & e.x \mid e[e] \mid w \\
\textit{value} & v & ::= & \lambda(\overline{x : p}, \eta, e) \mid [\overline{l}]^{s} \mid [\overline{l}] \mid \textit{box}(l) \\
 & & \mid & c \mid w \\
\textit{lvalue} & w & ::= & w.x \mid w[c] \mid l^{m} \\
\textit{env.} & E\langle\cdot\rangle & ::= & \langle\cdot\rangle; e \mid b = \langle\cdot\rangle\ \texttt{in}\ e \mid l^{m} = \langle\cdot\rangle \\
 & & \mid & [\overline{v}, \langle\cdot\rangle, \overline{e}] \mid s(\overline{v}, \langle\cdot\rangle, \overline{e}) \mid v(\overline{v}, \langle\cdot\rangle, \overline{a}) \\
 & & \mid & \langle\cdot\rangle.x \mid \langle\cdot\rangle[e] \mid v[\langle\cdot\rangle] \mid \langle\cdot\rangle; \texttt{pop}\ \overline{l} \\
 & & \mid & \langle\cdot\rangle\ \texttt{as}\ \tau \mid \langle\cdot\rangle
\end{array}
$$

Figure 6 Formal syntax of Swiftlet's small-step semantics

As mentioned earlier, the natural semantics we presented in Section 4.3 is intended as a formal framework to understand Swiftlet's user model at a high level. We now introduce a second semantics to study the language's low-level operational details and evaluate the soundness of its static semantics.

Figure 6 presents the formal syntax of Swiftlet's small-step semantics. That syntax naturally resembles the one shown in Figure 3. The main differences relate to data representation.

In the small-step semantics, evaluation contexts are split into two partial functions π and η. The first maps memory locations to values while the second maps bindings to memory locations and their mutability. In other words, the small-step semantics models memory cells, whereas that detail is kept abstract in the natural semantics. Hence, an evaluation context μ such that $\mu(x) = m\ v$ in the natural semantics is represented in the small-step semantics by a pair of functions π, η such that $\eta(x) = l$ and $\pi(l) = m\ v$.

Swiftlet's operational semantics appears in Figure 7. We define two evaluation operators. The first ($\longrightarrow$) reduces expression to values, while the second ($\longrightarrow_{lv}$) reduces path expressions to memory locations. Both are defined in a context Δ and map a program state onto its successor. A program state is a triple that consists of a pointer store π, a stack of frames $\overline{\eta}$ and an expression e. Together, π and $\overline{\eta}$ encode the program's memory state, keeping track of the value of each accessible binding in a given frame. For simplicity, we abstract over the size of a value and assume an unbounded set of memory locations.

Example 4.4 (Memory state)**.** Let π and $\overline{\eta}$ represent the memory state of a program such that $\pi = [l_1 \mapsto \texttt{var}\ 8, l_2 \mapsto \texttt{var}\ 6]$ and $\overline{\eta} = [x \mapsto l_1], [x \mapsto l_2, y \mapsto l_2]$. In this state, only x is accessible, as y does not appear in the youngest frame. Furthermore, x is known to be bound to a mutable memory location l_1, at which the value 8 is stored.

We borrow the notion of *evaluation environment* (a.k.a. evaluation context (Felleisen et al. 1987)) to specify the evaluation order of an expression concisely. An environment $E\langle\cdot\rangle$ is a meta-term representing a family of expressions where $\langle\cdot\rangle$ is a "hole" denoting the sub-expression that must be evaluated next. For instance, $b = \langle\cdot\rangle; e$ denotes a family of binding declarations where the value to bind is being evaluated. Then, we write $E\langle e\rangle$ for the substitution of the environment's hole for the expression e and use ESS-CONTEXT to evaluate environments.

Value creation Structures and arrays are handled similarly. Once the arguments have been evaluated, we create a sequence of fresh memory locations that correspond to the "internal" representation of the structure or array value. For a structure (rule ESS-STRUCTLIT), each field's mutability depends on the qualifier of the corresponding property declaration. For instance, given a program with a declaration $\texttt{struct}\ A\{\texttt{var}\ x : \tau, \texttt{let}\ y : \tau\}$, evaluating an expression with type A results in a value $[l_1, l_2]^{A}$ in a memory state π, $\overline{\eta}$ such that $\pi(l_1) = \texttt{var}\ v_1$ and $\pi(l_2) = \texttt{let}\ v_2$. Hence, the pointer store can keep track of each field's mutability and detect illegal write accesses to immutable values later on. On the other hand, memory locations corresponding to array elements (rule ESS-ARRAYLIT) are always qualified by `var`.

Closures are represented differently than in the natural semantics. Specifically, rather than encoding environments syntactically, by substituting captures for their values in the body of the function, we encode them in the form of a frame η_{λ} that is part of the closure's value. Capture-by-copy is carried out by helper function *mkenv* that creates new entries in the pointer store for each captured symbol:

$\boxed{\Delta \vdash \pi;\overline{\eta};e \longrightarrow \pi';\overline{\eta}';e'}$

ESS-CONTEXT
$$\frac{\Delta \vdash \pi;\overline{\eta};e \longrightarrow \pi';\overline{\eta}';e'}{\Delta \vdash \pi;\overline{\eta};E\langle e\rangle \longrightarrow \pi';\overline{\eta}';E\langle e'\rangle}$$

ESS-BINDING
$$\frac{\begin{array}{c} l \notin \mathit{dom}(\pi) \\ \pi' = \pi[l \mapsto m\ v] \qquad \eta' = \eta_1[x \mapsto l] \end{array}}{\Delta \vdash \pi;\overline{\eta};m\ x : \tau = v\ \texttt{in}\ e \longrightarrow \pi';\eta',\overline{\eta};e;\texttt{pop}\ l}$$

ESS-SEQ
$$\frac{\pi' = \mathit{drop}(\pi, v)}{\Delta \vdash \pi;\overline{\eta};v;e \longrightarrow \pi';\overline{\eta};e}$$

ESS-NAME
$$\frac{\begin{array}{c} \pi(\eta_1(x)) = m\ v \\ \pi', v' = \mathit{copy}(\pi, v) \end{array}}{\Delta \vdash \pi;\overline{\eta};x \longrightarrow \pi';\overline{\eta};v'}$$

ESS-PROP
$$\frac{\begin{array}{c} \pi(l_i) = m\ v \qquad \pi', v' = \mathit{copy}(\pi, v) \\ \texttt{struct}\ s\ \{m_1\ x_1 : \tau_1, \dots, m_k\ x_k : \tau_k\} \ \in \Delta \end{array}}{\Delta \vdash \pi;\overline{\eta};[l_1, \dots, l_k]^s.x_i \longrightarrow \pi',\overline{\eta};v'}$$

ESS-ELEM
$$\frac{\begin{array}{c} \pi(l_{c+1}) = m\ v \\ \pi', v' = \mathit{copy}(\pi, v) \qquad 0 \le c < k \end{array}}{\Delta \vdash \pi;\overline{\eta};[l_1, \dots, l_k][c] \longrightarrow \pi';\overline{\eta};v'}$$

ESS-STRUCTLIT
$$\frac{\begin{array}{c} l_1, \dots, l_k \notin \mathit{dom}(\pi) \\ \pi' = \pi[l_i \mapsto m_i\ v_i \mid 1 \le i \le k] \qquad \texttt{struct}\ s\ \{m_1\ x_1 : \tau_1, \dots, m_k\ x_k : \tau_k\} \ \in \Delta \end{array}}{\Delta \vdash \pi;\overline{\eta};s(v_1, \dots, v_k) \longrightarrow \pi';\overline{\eta};[l_i, \dots, l_k]^s}$$

ESS-ARRAYLIT
$$\frac{\begin{array}{c} l_1, \dots, l_k \notin \mathit{dom}(\pi) \\ \pi' = \pi[l_i \mapsto \texttt{var}\ v_i \mid 1 \le i \le k] \end{array}}{\Delta \vdash \pi;\overline{\eta};[v_1, \dots, v_k] \longrightarrow \pi';\overline{\eta};[l_1, \dots, l_k]}$$

ESS-FUNC
$$\frac{\begin{array}{c} l \notin \mathit{dom}(\pi') \qquad \eta' = \eta_1[x_0 \mapsto l] \\ \pi', \eta_\lambda = \mathit{mkenv}(\pi, y_1 : \eta_1(y_i), \dots, y_h : \eta_1(y_h)) \qquad \pi'' = \pi'[l \mapsto \texttt{let}\ \lambda(\overline{x : p}, \eta_\lambda, e_1)] \end{array}}{\Delta \vdash \pi;\overline{\eta};\texttt{func}\ x_0(\overline{x : p}) \to \tau_\lambda\ \{[y_1, \dots, y_h]\ \texttt{in}\ e_1\}\ \texttt{in}\ e_2 \longrightarrow \pi'';\eta',\overline{\eta};e_2;\texttt{pop}\ l}$$

ESS-COND-T
$$\frac{c \neq 0}{\Delta \vdash \pi;\overline{\eta};c\ ?\ e_t\ :\ e_e \longrightarrow \pi';\overline{\eta};e_t}$$

ESS-CALL
$$\frac{\begin{array}{c} I_{cpy} = \{i \mid 1 \le i \le k \land p_i = \tau_i\} \qquad I_{ref} = \{i \mid 1 \le i \le k \land p_i = \texttt{inout}\ \tau_i\} \\ \{l_i \mid i \in I_{cpy}\} \cap \mathit{dom}(\pi) = \varnothing \qquad \forall i, j \in I_{ref}, i \neq j \implies \mathit{acc}(\pi, v_i) \cap \mathit{acc}(\pi, v_j) = \varnothing \\ \pi' = \pi[l_i \mapsto \texttt{let}\ v_i \mid i \in I_{cpy}] \qquad \eta' = \eta_\lambda[x_i \mapsto l_i \mid i \in I_{cpy}][x_i \mapsto v_i \mid i \in I_{ref}] \end{array}}{\Delta \vdash \pi;\overline{\eta};\lambda(x_1 : p_1, \dots, x_k : p_k, \eta_\lambda, e)(v_1, \dots, v_k) \longrightarrow \pi';\eta',\overline{\eta};e;\texttt{pop}\ \{l_i \mid i \in I_{cpy}\}}$$

ESS-COND-F
$$\frac{}{\Delta \vdash \pi;\overline{\eta};0\ ?\ e_t\ :\ e_e \longrightarrow \pi';\overline{\eta};e_e}$$

ESS-POP
$$\frac{\begin{array}{c} \overbrace{\pi_{i-1}(l_i) = m_i\ v_i}^{1 \le i \le k} \qquad \overbrace{\pi_i = \mathit{drop}(\pi_{i-1}, v_i)}^{1 \le i \le k} \\ \pi' = \pi_k[l_i \mapsto \bot \mid 1 \le i \le k] \end{array}}{\Delta \vdash \pi_0;\eta,\overline{\eta};v;\texttt{pop}\ l_1, \dots, l_k \longrightarrow \pi';\overline{\eta};v}$$

ESS-INOUT-PATH
$$\frac{\Delta \vdash \pi;\overline{\eta};\&r \longrightarrow_{lv} \pi',\overline{\eta}';r'}{\Delta \vdash \pi;\overline{\eta};\&r \longrightarrow \pi',\overline{\eta}';r'}$$

ESS-INOUT
$$\frac{}{\Delta \vdash \pi;\overline{\eta};\&l^{\texttt{var}} \longrightarrow \pi,\overline{\eta};l}$$

ESS-ASSIGN-PATH
$$\frac{\Delta \vdash \pi;\overline{\eta};r \longrightarrow_{lv} \pi';\overline{\eta}';r'}{\Delta \vdash \pi;\overline{\eta};r = e_1;e_2 \longrightarrow \pi';\overline{\eta}';r' = e_1;e_2}$$

ESS-ASSIGN
$$\frac{\pi(l) = m_0\ v_0 \qquad \pi' = \mathit{drop}(\pi, v_0) \qquad \pi'' = \pi'[l \mapsto \texttt{var}\ v]}{\Delta \vdash \pi;\overline{\eta};l^{\texttt{var}} = v;e \longrightarrow \pi'';\overline{\eta};e}$$

ESS-UPCAST
$$\frac{l \notin \mathit{dom}(\pi) \qquad \pi' = \pi[l \mapsto \texttt{let}\ v]}{\Delta \vdash \pi;\overline{\eta};v\ \texttt{as}\ \mathit{Any} \longrightarrow \pi';\overline{\eta};\mathit{box}(l)}$$

ESS-DOWNCAST
$$\frac{\tau \neq \mathit{Any} \qquad \mathit{typeof}(v) = \tau \qquad \pi(l) = \texttt{let}\ v \qquad \pi', v' = \mathit{copy}(\pi, v)}{\Delta \vdash \pi;\overline{\eta};\mathit{box}(l)\ \texttt{as}\ \tau \longrightarrow \pi';\overline{\eta};v'}$$

$\boxed{\Delta \vdash \pi;\overline{\eta};r \longrightarrow_{lv} \pi';\overline{\eta}';r'}$

PSS-NAME
$$\frac{\eta_1(x) = l \qquad \pi(l) = m\ v}{\Delta \vdash \pi;\overline{\eta};x \longrightarrow_{lv} \pi;\overline{\eta};l^m}$$

PSS-STRUCT
$$\frac{\Delta \vdash \pi;\overline{\eta};r \longrightarrow_{lv} \pi';\overline{\eta}';r'}{\Delta \vdash \pi;\overline{\eta};r.x \longrightarrow_{lv} \pi';\overline{\eta}';r'.x}$$

PSS-PROP
$$\frac{\begin{array}{c} m' = \min(m, m_i) \qquad \pi(l) = m\ [l_1, \dots, l_k]^s \\ \texttt{struct}\ s\ \{m_1\ x_1 : \tau_1, \dots, m_k\ x_k : \tau_k\} \ \in \Delta \end{array}}{\Delta \vdash \pi;\overline{\eta};l^m.x_i \longrightarrow_{lv} \pi;\overline{\eta};l_i^{m'}}$$

PSS-ARRAY
$$\frac{\Delta \vdash \pi;\overline{\eta};r \longrightarrow_{lv} \pi';\overline{\eta}';r'}{\Delta \vdash \pi;\overline{\eta};r[e] \longrightarrow_{lv} \pi';\overline{\eta}';r'[e]}$$

PSS-INDEX
$$\frac{\Delta \vdash \pi;\overline{\eta};e \longrightarrow \pi';\overline{\eta}';e'}{\Delta \vdash \pi;\overline{\eta};l^m[e] \longrightarrow_{lv} \pi';\overline{\eta}';l^m[e']}$$

PSS-ELEM
$$\frac{0 \le c < k \qquad \pi(l) = m\ [l_1, \dots, l_k]}{\Delta \vdash \pi;\overline{\eta};l^m[c] \longrightarrow_{lv} \pi;\overline{\eta};l^m_{c+1}}$$

Figure 7 Small-step semantics of Swiftlet

$$\frac{}{mkenv(\pi, \varepsilon) = \pi, [\bot]}$$

$$\frac{\pi(l) = m\ v \qquad \pi', v' = copy(\pi, v) \qquad l' \notin dom(\pi') \qquad mkenv(\pi'[l' \mapsto m\ v'], \overline{x : l}) = \pi'', \eta''}{mkenv(\pi, x : l, \overline{x : l}) = \pi'', \eta''[x \mapsto l']}$$

The rule ESS-FUNC applies *mkenv* on the current frame and the bindings declared in the function's capture list. It concludes with a value of the form $\lambda(\overline{x : p}, \eta_\lambda, e)$, where η_λ is the captured environment.

Value copy A number of rules formally describe situations in which copying takes place. All rely on a helper function *copy* to clone a value v in a store π.

We mentioned in Section 4.3 that, at an abstract level, copying was equivalent to the identity. At a lower-level, however, copying structures, closures, arrays and existential containers involve memory allocations that require more attention.

To preserve value independence, *copy* must not clone locations that are part of these values' runtime representation; it must allocate new memory and duplicate contents. Fortunately, as values always form topological trees, the helper can be implemented as a recursive traversal of their representation. For instance, copying structures is defined as follows:

$$\frac{\overbrace{\pi_0(l_i) = m_i\ v_i}^{1 \le i \le k} \qquad \overbrace{\pi_i, v'_i = copy(\pi_{i-1}, \pi_0(l_i))}^{1 \le i \le k} \qquad l'_1, \ldots, l'_k \notin dom(\pi_k) \qquad \pi' = \pi_k[l'_i \mapsto m_i\ v'_i \mid 1 \le i \le k]}{\pi', [l'_1, \ldots, l'_k]^s = copy(\pi_0, [l_1, \ldots, l_k]^s)}$$

Garbage collection Rules ESS-BINDING, ESS-FUNC, and ESS-CALL relate to expressions that delimit a scope. The three of them append an expression of the form `pop` $\bar{l}$ in their conclusions, where the sequence $\bar{l}$ represents the memory locations at which scoped values where allocated.

Pop expressions delimit the end of a scope. They are evaluated by ESS-POP, which removes the last frame and destroy the values stored at the locations $\bar{l}$, reclaiming the memory of the values that are no longer accessible.

Memory collection is carried out by a helper function *drop* that destroys a value by freeing the locations that are part of its representation. Just as copying, this process is implemented as a recursive traversal of the value's representation. For instance, destroying structures is defined as follows:

$$\frac{\overbrace{\pi_0(l_i) = m_i\ v_i}^{1 \le i \le k} \qquad \overbrace{\pi_i = drop(\pi_{i-1}, v_i)}^{1 \le i \le k} \qquad \pi' = \pi_k[l'_i \mapsto \bot \mid 1 \le i \le k]}{\pi' = drop(\pi_0, [l_1, \ldots, l_k]^s)}$$

Assigning a value to a binding ends the lifetime of its current value. Hence, *drop* also appears in the premise of ESS-ASSIGN to collect the memory of the binding's current value. Finally, ESS-SEQ ends the lifetime of the value resulting from the evaluation of the first expression, as it is known to be irrelevant for the remainder of the execution.

Function calls The rules ESS-CALL describes function calls. It prepares a new frame initialized with the closure's environment and extended with new bindings for each argument. This frame is then pushed on the stack, and used to evaluate the function's body. Hence, the callee cannot access the caller's binding, unless they are passed explicitly as `inout` arguments.

Frame preparation requires ESS-CALL to distinguish between *direct* arguments and `inout` arguments. It does so by collecting argument indices in two different sets, namely I_{cpy}, for direct arguments, and I_{ref}, for `inout` arguments. The value of each direct argument is inserted into the new frame, while the *location* of each `inout` argument is inserted into the new frame, without involving any copy. These locations are obtained by applying ESS-INOUT on paths prefixed with an ampersand (&). The rule prescribes that the path be mutable, guaranteeing that immutable bindings cannot serve as `inout` arguments. In addition, the rule requires that `inout` arguments denote disjoint parts of the pointer store.

The helper function *acc* builds the set of memory locations *reachable* from a given root location. Since composition denotes whole/part relationships, this set corresponds to the locations that constitute a value's in-memory representation, together with the locations that constitute parts of this value. For instance, the locations reachable from a location at which a structure instance is stored are obtained as follows:

$$\frac{\pi(l_0) = m\ [l_1, \ldots, l_k]^s}{acc(\pi, l_0) = \bigcup_{1 \le i \le k} acc(\pi, l_i) \cup \{l_0, l_1, \ldots, l_k\}}$$

4.6. Soundness

We now discuss how Swiftlet's operational semantics relates to its typing semantics. Specifically, our typing rules guarantee that well-typed programs can either be reduced to a value, or never terminate, or fail because of a runtime error, such as an invalid cast or a out-of-bound array access. Hence, crucially, they cannot violate immutability restrictions and local reasoning.

We first establish these guarantees on the small-step semantics from Section 4.5.

Definition 4.1 (Well-formed memory state)**.** A memory state $\pi; \overline{\eta}$ is well-formed if $|\overline{\eta}| \ge 1$ and for any pair of bindings $x, x' \in dom(\eta_1)$ such that $x \ne x'$, and $l = \eta_1(x)$, and $l' = \eta_1(x')$, we have $acc(\pi, l) \cap acc(\pi, l') = \varnothing$.

Definition 4.2 (Well-typed memory state)**.** A well-formed memory state $\pi; \overline{\eta}$ is well-typed in a typing context Γ, written $\Delta \vdash \pi; \overline{\eta} : \Gamma$, if and only if for all frames η_i in $\overline{\eta}$ we have $\forall m\ x : \tau \in \Gamma, \pi(\eta_i(x)) = m\ v \wedge \Delta; \Gamma; \pi \vdash v : \tau$.

In plain English, Definition 4.1 states that a memory state is *well-formed* if all its local bindings denote disjoint regions of the memory. Definition 4.2 states that a memory state is *well-typed* in the context of Γ if it matches the latter's typing assumptions. Remark the overloaded typing judgment $\Delta; \Gamma; \pi \vdash e : \tau$, which also involves the pointer store in order to type values whose representation involve memory allocations.

We state the soundness theorem in the classical syntactic style of Wright & Felleisen (1994). Full proofs appear in Appendix A.

Lemma 4.1 (Progress)**.** Given $\pi;\overline{\eta}$ such that $\Delta;\Gamma;\pi \vdash e : \tau$ and $\Delta \vdash \pi;\overline{\eta} : \Gamma$, either e is a value, or there exist $\pi',\overline{\eta}',e'$ such that $\Delta \vdash \pi;\overline{\eta};e \longrightarrow \pi';\overline{\eta}';e'$, or the program is stuck due to a runtime error.

Lemma 4.2 (Preservation)**.** Given $\pi;\overline{\eta}$ such that $\Delta;\Gamma;\pi \vdash e : \tau$, and $\Delta \vdash \pi;\overline{\eta} : \Gamma$, and $\Delta;\pi;\overline{\eta};e \longrightarrow \pi';\overline{\eta}';e'$, there exists Γ' such that $\Delta;\Gamma';\pi' \vdash e' : \tau$ and $\Delta \vdash \pi';\overline{\eta}' : \Gamma'$.

Given a well-typed memory state, Lemma 4.1 states that the evaluation of an expression e either steps due to an invalid array subscript (e.g., $a[2]$ where a is an empty array), or a invalid cast operation. Lemma 4.2 states that the evaluation of a step preserves well-formedness and well-typedness. Type soundness follows trivially.

Theorem 4.1 (Type soundness)**.** If $\Delta;\varnothing;[\bot] \vdash e : \tau$ and $\Delta \vdash [\bot];[\bot];e \longrightarrow^* \pi';\overline{\eta}';e'$, then either e' is a value or the program is stuck due to a runtime error.

$$[\![c]\!]^{\pi}_{\Delta} = c$$

$$[\![box(l)]\!]^{\pi}_{\Delta} = box([\![v]\!]^{\pi}_{\Delta}) \quad \text{where } \pi(l) = m\ v$$

$$[\![box(l)]\!]^{\pi}_{\Delta} = box([\![v]\!]^{\pi}_{\Delta}) \text{ if } \pi(l) = m\ v$$

$$[\![[l_1,\ldots,l_k]]\!]^{\pi}_{\Delta} = [[\![l_1]\!]^{\pi}_{\Delta},\ldots,[\![l_k]\!]^{\pi}_{\Delta}]$$

$$[\![[l_1,\ldots,l_k]^s]\!]^{\pi}_{\Delta} = [x_i \mapsto m_i\ [\![l_i]\!]^{\pi}_{\Delta} \mid 1 \le i \le k]^s \quad \text{where } \texttt{struct}\ s\ \{m_1\ x_1 : \tau_1,\ldots,m_k\ x_k : \tau_k\} \in \Delta$$

$$[\![\lambda(\overline{x : p},\eta_\lambda,e)]\!]^{\pi}_{\Delta} = \lambda(\overline{x_p},e[/\sigma]) \quad \text{where } \sigma = [x_i \mapsto [\![v_i]\!]^{\pi}_{\Delta} \mid x_i \in dom(\eta_\lambda) \wedge \eta_\lambda(x_i) = l_i \wedge \pi(l_i) = m_i\ v_i]$$

Figure 8 Correspondence between value representations

To convince ourselves that the natural semantics from Section 4.3 is an appropriate abstraction of the small-step semantics, can establish an equivalence relation between the two.

Theorem 4.2 (Semantics equivalence)**.** Given an expression e expressible in the natural semantics and a set of structure declarations Δ, if $\Delta \vdash [\bot];[\bot]; \longrightarrow^n \pi;\overline{\eta};v$, then there exists μ',v' such that $\Delta,[\bot] \vdash e \Downarrow \mu',v'$, then such that $[\![v]\!]^{\pi}_{\Delta} = v'$.

Proof sketch. The proof is by induction on the length n of the evaluation sequence $\Delta \vdash [\bot];[\bot]; \longrightarrow^n \pi;\overline{\eta};v$. If $n = 0$, then e is a value and the property holds trivially. Otherwise, assume the property holds for $n' \le n$ and prove it for a sequence of length $n'+1$. □

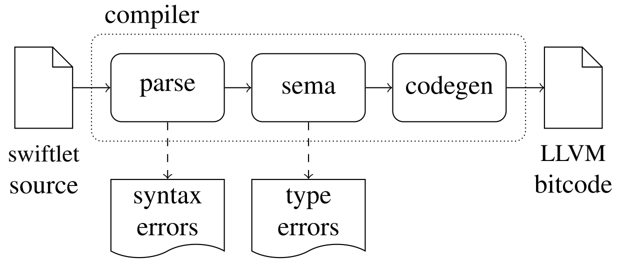


Figure 9 Swiftlet's compiler architecture

Equivalence between memory representations is determined by the means of an operator det $\cdot^{\pi}_{\Delta}$, defined in Figure 8. The operator translates a value from the small-step semantics into their representation in the natural semantics, given a set of structure declarations Δ and a pointer map π.

5. Generating native code with LLVM

This section describes the implementation of a compiler for Swiftlet. That compiler is written in Swift, in the style of MVS, and distributed as an open-source project hosted on GitHub: https://github.com/kyouko-taiga/mvs-calculus.

Figure 9 gives an overview of the compiler's architecture. The "parse" module implements a recursive-descent parser using combinators (Hutton & Meijer 1998) that transforms textual sources to an abstract syntax tree (AST). That AST is passed to the "sema" module (for semantic analysis), which is essentially a type checker. It verifies that expressions are well-typed (e.g., variables of type `Int` are only assigned to integer values), that mutability constraints are satisfied (e.g., constants are never mutated), and guarantees path uniqueness for all `inout` arguments. Finally, the "codegen" module translates the AST to LLVM's intermediate representation, optionally applying a handful of optimizations. Note that code generation always succeed, as ASTs that passed semantic analysis are guaranteed to be well-formed.

LLVM (Lattner & Adve 2004) is a popular middleware in numerous compilers, including Clang (C/C++ and Objective-C), rustc (Rust) and even swiftc (Swift). The framework is centered around an SSA-style (Cytron et al. 1991) intermediate representation, called LLVM IR, that serves as a front-end agnostic language to apply code optimizations, and generate machine code. Hence, LLVM IR dramatically reduces the engineering effort required to build a compiler. However, translating language features into this common representation—a process often referred to as *lowering*—comes with its own challenges.

5.1. Memory representation

There are four built-in data types in Swiftlet: `Int` for signed integer values, `Double` for double-precision floating-point numbers, a generic type `[T]` for arrays of type `T`, and `Any` for containers of arbitrary values. In addition, the language supports two kinds of user-defined types: functions and structures.

5.1.1. Scalars and structures

`Int` and `Double` have a 1-to-1 correspondence with machine types and are represented as `i64` and `double` in LLVM, respectively. Since `struct` declarations cannot be mutually recursive, all values of a structure have a finite memory representation (more on that later) and can be represented as a passive data structure (PDS), where each field is laid out contiguously with possible padding for alignment.

5.1.2. Arrays Arrays require dynamic allocation, as the compiler is in general incapable of determining their size statically. An array is represented by a pointer ϕ to a contiguous block of heap-allocated memory. That block is structured as a tuple $\langle r, n, k, \overline{e} \rangle$ where r is a reference counter, n denotes the number of elements in the array, k denotes the capacity of the array's payload (i.e., the size of its actual contents in bytes) and $\overline{e}$ is a payload of k bytes, containing n elements. The counter r serves to implement *copy-on-write* (see Section 6.3).

Figure 10 depicts the in-memory representation of an array assigned to a local variable. The square on the left of the dashed line represents the single memory cell allocated on the stack, containing the pointer ϕ. The squares on the right represent cells allocated in the heap. Each cell e_i is a single independent element that may itself contain pointers to other heap-allocated memory blocks (e.g., for an array of arrays).

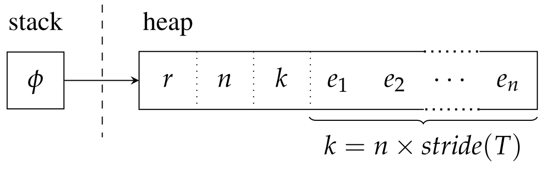


Figure 10 In-memory representation of an array of T

The capacity k of an array typically differs from the number of its elements n. The former depends on the size of an element in memory, or more precisely, its *stride*. The stride of a type denotes the number of bytes between two consecutive instances stored in contiguous memory, which depends on the size and memory alignment of a type. Both information depend on the target ABI and are left for LLVM to figure out.

Example 5.1. An array of two 16-bit integers $[42, 1337]$ on a 64-bit little-endian machine is represented by the byte sequence $\langle 1, 0, 0, 0, 2, 0, 0, 0, 4, 0, 0, 0, 42, 0, 5, 57 \rangle$. It contains two elements, thus $n = 2$, yet its capacity $k = 4$ since each element has a stride of two bytes.

5.1.3. Closures Just like arrays, closures require dynamic allocation because the size of their environment cannot be determined statically. A closure is represented as a triple $\langle \phi, \epsilon, \nu \rangle$ where ϕ is a pointer to a function implementing the closure, ϵ is a pointer to the closure's environment (potentially null if the closure has no captures), and ν is a pointer to the value witness of the closure (see Section 5.2).

Figure 11 depicts the in-memory representation of a closure graphically. The cells $\overline{e}$ represent the contents of the closure's environment, laid out contiguously. Just like in the case of arrays, each cell is an independent value.

The function pointed by ϕ is obtained by *defunctionalization* (Reynolds 1998a). This process transforms a closure into a

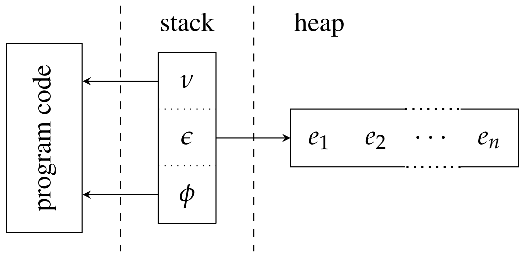


Figure 11 In-memory representation of a closure

global function in which all captured identifiers are lifted into an additional parameter for the closure's environment.

5.1.4. Existential containers Containers of type `Any` are implemented via *value boxing* (Henglein & Jørgensen 1994). Boxing consists of a storing a value inside of a heap-allocated area, so that it can be represented by a fixed-sized pointer to that area. Unfortunately, this approach suffers a performance penalty incurred by heap allocation and collection, which may prove particularly expensive in a programming language with pervasive copying.

We leverage a technique called *small-object optimization* to mitigate that cost. Instead of systematically representing a container as a pointer to a heap-allocated area, we use a small buffer that is large enough to fit small objects inline. Values are boxed in the heap only when they are too large to fit inside of the buffer. In this case, the latter is used to store a pointer to out-of-line storage. Otherwise, we can avoid the cost of allocating and freeing memory in the heap and eliminate the indirection overhead typically caused by value boxing.

A container is represented as a tuple $\langle s, \nu \rangle$ where s is a small inline buffer and ν is a pointer to the value witness of the wrapped value (see Section 5.2). Choosing the size of s is a trade-off between minimizing heap allocation and minimizing the space that is wasted when the wrapped value is smaller than the buffer, or when it must be allocated out-of-line nonetheless. In our implementation, that space is large enough to fit three 64-bit integer values, which is sufficient to store numeric values, arrays, and closures.

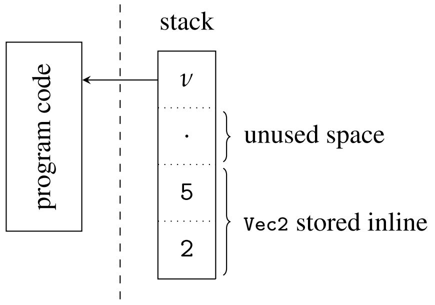


Figure 12 In-memory representation of an existential container using inline storage to hold a vector `Vec2(x: 2, y: 5)`

Figure 12 shows an example of the in-memory representation of a container that stores a 2-dimensional vector (i.e., the result of an expression `Vec2(x: 2, y: 5) as Any`). Here, the vector's value is small enough to be stored directly inside of the container's buffer, leaving some unused space.

5.2. Value witnesses

A value's lifetime corresponds to the span of time from its initialization to its destruction. In the absence of first-class references, that information can be determined statically. Initialization occurs when a value is assigned to a variable while two events can trigger destruction: reassignment and exit from the variable's scope. Following this observation, memory management can be automated during code generation.

Swift allows the declaration of a variable to be separated from its initialization. It then relies on definite assignment analysis (Fruja 2004) for guaranteeing initialization before use, possibly inserting dynamic checks in situations where that property cannot be determined statically (e.g., when variables are initialized conditionally). In contrast, Swiftlet requires that all local bindings be initialized at the point of their declaration. This restriction conveniently implies that the compiler can always distinguish between initialization and assignment.

We say that a type is *trivial* if it denotes a numeric value or a composition of trivial types in a structure (e.g., a pair of `Int`s). Conversely, types requiring dynamic allocations are *non-trivial*. That includes array types, function types, existential containers, and structures containing at least one non-trivial property. In other words, the notional value of a trivial type is represented exactly by the contents of its inline storage, whereas the notional value of a non-trivial type may include out-of-line storage.

Since trivial types do not involve any out-of-line storage, copying or deinitializing a value does not necessitate any particular operation. Hence, assigning a variable of a trivial type boils down to a byte-wise copy of the right operand.

The situation is a bit more delicate for non-trivial types. For arrays, a first issue is that the size of the heap-allocated storage cannot be determined statically. Instead, it depends on the value of k in the tuple representing the array. A second issue is that copying may involve additional operations if the elements contained in the array are dynamically sized as well. In this case, a byte-wise copy of the array's payload would create unintended aliases, breaking value independence. Instead, each non-trivial element should be copied individually. One solution is to synthesize a function for each data type that is applied whenever a copy should occur.

5.2.1. *Synthesizing copy and destruction*

If the type is trivial (i.e., it does not involve any dynamic allocation), its copy function is equivalent to a byte-wise copy. Otherwise, it implements the appropriate logic, calling the copy function of each contained element. Similarly, the logic implementing the destruction of a value can be synthesized into a destructor. If the type is trivial, then this destructor is a no-op. Otherwise, it recursively calls the destructor of each contained element and frees the memory allocated for all values being destroyed.

We synthesize a copy function and a destructor for every type used in a program. Together, these functions form the *value witness* of a type. Just as the witness type specifies the hidden representation of an existential package in type theory (Pierce 2002, Chapter 24), a value witness species the hidden implementation of a type's value semantics, providing the compiler with a uniform programming interface to interact with values. A call to the copy function is issued every time a value is assigned or crosses function boundaries, while a call to the destructor is issued before lifetime ending events, effectively implementing compile-time garbage collection.

All values of a particular type share the same value witness, except closures. Indeed, the environment captured by a closure of type `(T) -> U` might differ from that of another closure with the same type. Hence, a different witness must be synthesized for each function declaration. Incidentally, that explains why closure tuples contain pointers to their copy function and destructor.

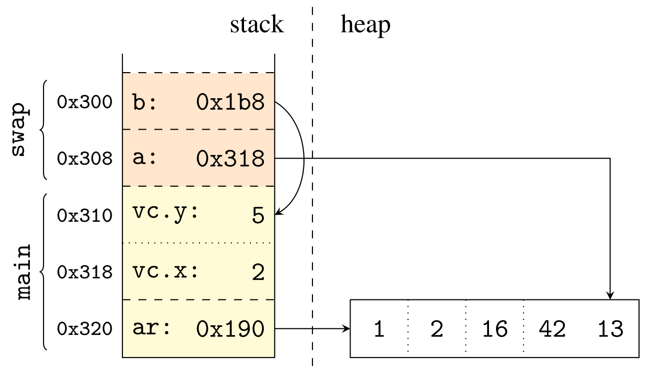


Figure 13 In-memory representation of `inout` arguments

5.2.2. *Synthesizing equality*

As observed in Section 2.4, the ability of MVS to express whole/part relationships allows us synthesize operations based on notional values, such as hashing and equality.

Swiftlet synthesizes an equality function for all types used in the program.[16] For integer and floating-point values, equality corresponds directly to LLVM's `icmp` and `fcmp` instructions, respectively. For other types, the compiler builds a function that recursively checks for equality on each part of the value.

5.3. Inout arguments

At function boundaries, structures are exploded into scalar arguments and passed through registers, provided the machine has enough of them. If the structure is too large, it is passed as a pointer to a stack cell in the caller's context, in which a copy of the argument is stored before the call.

`inout` arguments are passed as (possibly interior) pointers. If the argument refers to a local variable or one of its fields, then it is passed as a pointer to the stack. If it refers to the element of an array, then it is passed as a pointer to the array's storage, offset by the element's index.

Example 5.2. Consider the following program:

```
struct Vec2 { var x: Int; var y: Int; };
```

[16] In Swift, synthesizing equality (and hashing) is provided as an opt-in mechanism by conforming to the protocol `Equatable` (and `Hashable`).

```
func swap(a: inout Int, b: inout Int)
  -> Void { ... };
var ar = [42, 13];
var vc = Vec2(x: 2, y: 5);
swap(a: &ar[1], b: &vc.y)
```

Figure 13 shows the contents of the program's memory during the call to `swap`. As discussed in Section 5.1, the array is allocated out-of-line whereas the 2-dimensional vector is laid out on the stack contiguously. Both `inout` arguments are passed as pointers. The first refers to the second element of the array, stored in the heap. The second refers to the second property of the vector, stored in the stack.

The compiler can guarantee that the pointee can never be outlived, because the language disallows the pointer to escape in any way. In fact, the value of the pointer itself is not accessible. The code generator sees that the callee can only dereference it, either to store or load a value. In our formal semantics, that behavior is modeled by the fact that the path of a value passed as `inout` argument is substituted for the corresponding parameter in the callee (Figure 4, rule E-CALL). Further, recall that the type system guarantees exclusive mutable access to any memory location. Hence, pointers representing `inout` arguments are known to be unique.

6. Optimizations

The language implementation we have described in Section 5 generates a fair amount of memory traffic, because copies are created every time a value is assigned to a variable or passed as an argument. Much of this traffic is unnecessary, though, because most original values are likely to be destroyed immediately after being copied, or because copied values might never be mutated and could have been shared.

This section discusses a handful of techniques included in Swiftlet's compiler to eliminate these unnecessary copies.

6.1. Move semantics

A recurring pattern is to assign values just after they have been created. For example, consider the expression `let x = [1, 2]; f(x)`. The value of the array is assigned directly after its creation, triggering a copy, and it is destroyed immediately afterwards. In other words, a naive implementation will evaluate the right operand, resulting in the creation of a new array value, copy this value to assign `x` and then destroy the original.

To study this inefficiency more formally, we can observe E-BINDING. The unnecessary copy is modeled by $copy(v_1)$ in the premise of the rule. In the concrete operational semantics, *copy* corresponds to the copy function of a value witness, which may involve expensive memory allocation.

Clearly, the copy is useless in the above-mentioned example, since the value of the right operand will never be used after the binding's initialization. Therefore, that value could be *moved* rather than copied. Formally, such an optimization could be expressed by the following variant of E-BINDING:

E-BINDING-MOVE

$$\frac{\begin{array}{c}\Delta, \mu \vdash e_1 \Downarrow^R \mu', v_1 \\ is\_temporary(e_1) \qquad \Delta, \mu'[x \mapsto move(v_1)] \vdash e_2 \Downarrow^R \mu'', v_2\end{array}}{\Delta, \mu \vdash m\ x : \tau = e_1; e_2 \Downarrow^R \mu''[x \mapsto_? \mu'(x)], v}$$

The predicate $is\_temporary(e_1)$ holds when e_1 denotes a temporary value, such as an array literal, or the result of a function call. In that case, the value v_1 can be moved. At an abstract level, the function *move* is equivalent to the identity, just like *copy*. In the concrete operational semantics, however, *move* is a lifetime ending operation that transfers the byte representation of a value, meaning that destructors are not called on moved values. At the machine level, that transfer boils down to a byte-wise copy. Incidentally, *copy* and *move* are equivalent for trivial types but describe different behaviors for non-trivial types.

Earlier, we said that byte-wise copies of non-trivial types may create unintended aliases, threatening value independence. Nonetheless, since *move* is a lifetime ending operation, such aliases are in fact immediately destroyed.

Note that the same optimization could be applied on the last use of a value. In the expression `let x = [1, 2]; [x]`, the constant `x` is used only once, to initialize an array of arrays of `Int`s. As formally described by E-ARRAYLIT, `x` must be copied to create a new array instance. Nonetheless, since that use is the last occurrence of `x` in the entire expression, the value could be moved rather than copied.

6.2. Substituting aliases for copies

Function parameters are considered immutable unless they are annotated with `inout`. Moreover, recall that arguments in Swiftlet are passed by value. Therefore, in terms of a concrete operational semantics, arguments can be passed as aliases to values stored at the call site rather than being copied.

The soundness of this optimization relies on an important additional assumption. The lifetime of the alias must not exceed that of the aliased value. Fortunately, such an assumption can be verified by preventing aliased values from being destroyed, and aliases from escaping the callee. Remark that this idea is akin to Rust-like immutable borrows (Naden et al. 2012).

We can express the substitution of aliases for copies formally in a variant of E-CALL, which no longer copies argument values to build the substitution σ:

E-CALL-ALIAS

$$\frac{\begin{array}{c}\overbrace{\Delta, \mu_{i-1} \vdash a_i \Downarrow^R \mu_i, v_i}^{1 \le i \le k} \\ \Delta, \mu \vdash e_0 \Downarrow^R \mu_0, \lambda(x_1 : p_1, \ldots, x_k : pk, e_b) \\ \sigma = [x_i \mapsto v_i \mid 1 \le i \le k] \qquad \Delta, \mu_i \vdash e_b[/\sigma] \Downarrow^R \mu', v\end{array}}{\Delta, \mu \vdash e_0(a_1, \ldots, a_k) \Downarrow^R \mu', copy(v)}$$

The aforementioned assumption implies a kind of contract between the caller and the callee: both guarantee that the lifetime of a borrowed argument does not end before the end of the call. Hence, the callee is no longer responsible for ending the lifetime of borrowed parameters. In fact, that operation must even be prohibited. At a concrete operation level, it means that destructors are not called on borrowed parameters.

A second part of the contract stipulates that borrowed parameters may not escape. That clause is guaranteed by the application of *copy* in the conclusion of the rule, which produces a new value whose lifetime is independent from that of any borrowed parameter.

Following the same rationale, initialization of immutable bindings from immutable values can be substituted by aliases as well. Consider the following expression: `let x = [[1, 2], [3, 4]]; let y = x[0]; f(y)`. The constant `y` is initialized from another constant value. Moreover, the lifetime of `y` is lexically shorter than `x`'s. Therefore, `y` can simply alias `x`'s first element rather than copying it. Formally, this optimization can be described by another variant of E-BINDING:

E-BINDING-ALIAS

$$\frac{\Delta, \mu \vdash r \Downarrow^{L} \mu', \mathtt{let}\, w \qquad \sigma = [x \mapsto w] \qquad \Delta, \mu' \vdash e[/\sigma] \Downarrow^{R} \mu'', v}{\Delta, \mu \vdash \mathtt{let}\; x : \tau = r; e \Downarrow^{R} \mu''[x \mapsto_{?} \mu'(x)], v}$$

The rule only applies to binding declarations of the form $\mathtt{let}\; x : \tau = r; e$, where the binding is declared constant and initialized by a path expression. Notice that the path expression is evaluated with $\Downarrow^{L}$ rather than $\Downarrow^{R}$, producing an lvalue rather than a value. The rule additionally checks that this lvalue is immutable before substituting it for the declared binding in the expression e.

6.3. Copy-on-write

The optimization strategies we discussed in Section 6.2 are not applicable in the presence of mutation. Any assignment involving a mutable binding, on the left- or right-hand side typically requires a copy, because the value might be mutated later. Similarly, assigning a mutable value to a mutable binding also requires a copy.

Nonetheless, it is possible that neither the original nor the copy end up being actually mutated, perhaps because the mutation depends on a condition that is evaluated at runtime. In this case, unfortunately, the compiler must conservatively assume that a mutation will occur and perform a copy to preserve value independence.

One simple mechanism can be used to work around this apparent shortcoming: *copy-on-write*. Copy-on-write leverages runtime knowledge to delay copies until they are actually needed. Heap-allocated storage is associated with a counter that keeps track of the number of references to that storage. Every time a value is copied, an alias is created and the counter is incremented. The value of this counter is checked before mutation actually occurs, at runtime, to determine uniqueness. If the storage is shared, the counter is decremented, the storage is duplicated and the mutation is performed on a copy. Otherwise, the mutation is performed on the original.

Example 6.1. Consider the following program:

```
1 func sort(array: inout [Int]) -> Void { ... };
2 var a0 = [1, 2];
3 a0[1] = 3;
4 var a1 = a0;
5 sort(array: &a1)
6   // a0 and a1 are the same array
```

We assume the existence of a function that sorts an array in-place. Then, we declare an array `a0`, which is mutated at line 3. At that point, the value of `a0`'s internal reference counter is 1, so the mutation is performed on its storage directly.

Line 4 declares another array `a1`, initialized to `a0`. With copy-on-write, the value of `a0` is not copied right away. Instead, the reference counter of its internal storage is incremented, meaning that `a1` is actually an alias by the time it is passed as an argument to `sort`, at line 5. Should elements not be in order, the first mutation that `sort` will attempt will trigger a copy. In the present case, however, no copy will occur and `a0` and `a1` will continue to share state after line 5.

Of course, copy-on-write prevents purely static garbage collection. Indeed, because of potential sharing, the lifetime of heap-allocated storage can no longer be determined at compile-time. Nonetheless, garbage collection can still be automated with predictible performance. The reference counter is decreased whenever the destructor of a value referring to the associated storage is called. If it reaches zero, then the contents of the storage are destroyed and deallocated.

Swiftlet applies copy-on-write on arrays only, as structures are allocated inline, enabling a different set of optimizations to elimitate unnecessary copies. One limitation of our approach, though, stems from its interaction with the implementation of `inout` arguments (Section 5). Recall that an `inout` argument is passed as a (possibly interior) pointer. Hence, the callee has no way to determine whether or not that pointer refers to a value inside of a shared buffer. As a result, the caller is compelled to copy non-unique storage defensively.

Example 6.2. Consider the following program:

```
1 func sort(array: inout [Int]) > Void { ... };
2 var a0 = [[1, 2], [3, 4]];
3 let a1 = a1;
4 sort(array: &a0[1])
```

The variable `a0` is declared as an array of arrays of `Int`s. With copy-on-write, it shares state with the variable `a1` by the time `sort` is called at line 4. The caller is compelled to copy the outer array `a0` because there is no way for the callee to determine that inner array `a0[1]` is stored inside of shared storage.

Nonetheless, note that `a0`'s copy will not trigger the copy of its inner arrays, applying copy-on-write instead. Hence, a copy of the inner array `a0[0]` will occur if and only if `sort` must perform a mutation. Meanwhile, `a0[1]` will share state with `a1[1]` after the call at line 4.

6.4. Leveraging local reasoning

We cited O'Hearn et al. (2001) in the introduction to emphasize the importance of local reasoning for human developers and compilers alike. In particular, one can easily identify and discard mutations, whose results cannot be observed elsewhere.

```
struct Vec2 { ... };
// ...
```

```
3 func f(v0: Vec2) -> Vec2 {
4   var v1 = v0;
5   let v2 = v1;
6   v1.x = 8;
7   Vec2(x: v2.x, y: v1.y)
8 }
```

Consider the above program. Thanks to local reasoning, an optimizer can safely discard the assignment to `v1.x` at line 6, because its effect can never be observed. As all values are independent, neither the parameter `v0` nor the local variable `v1` can share mutable state with any other variable in the program, no matter what we write in place of the ellipsis at line 2. Furthermore, without the assignment at line 6, it becomes evident that `v0`, `v1`, and `v2` denote the exact same value. Therefore, constant propagation will eventually deduce that the function simply returns the value of its argument.

Freedom from aliasing also simplifies scalar replacement of aggregates (SROA) (Jambor 2010). This optimization consists of substituting the parts of an aggregate with local scalar variables. The goals of this substitution are twofold: avoid heap allocation by inlining aggregates on the stack, and unlock additional optimization opportunities based on scalar values, such as dead store elimination and constant propagation.

Our implementation substitutes array literals on the stack when it detects that their value can never escape. That detection is performed as a simple AST traversal that looks for uses of the array as a function argument or a return value. Then, it relies on LLVM's SROA pass to transform stack-allocated arrays to scalar SSA values.

7. Performance Evaluation

This section evaluates the performance of MVS, implemented with the strategies and optimizations we discussed in Sections 5 and 6, with the aim to answer the two following questions:

– Does MVS suffer prohibitive performance overhead due to the use of copy-on-write?
– Does MVS offer performance gains in comparison to functional updates?

7.1. Environment

The benchmarks were run on a workstation equipped with an Intel Xeon Gold 6154 CPU clocked at 3.00GHz and 192GB of RAM. The machine runs a Debian-based distribution that uses Linux 5.10.40 kernel.

We compiled benchmarks with Swift 5.5.2, Scala 2.12 compiled with Scala Native 0.4.2 (Shabalin 2020), Clang 11 for C++, and our own compiler for Swiftlet.[17] The choice of these compilers is deliberate. All implementations target LLVM IR, allowing us to explore trade-offs of the programming model on each language, rather than the differences of lower-level code generation.

Benchmarks were compiled with the most aggressive optimization level: `-O3` for C++, `-Ounchecked` for Swift, and `release -full` for Scala Native.

[17] https://github.com/kyouko-taiga/mvs-calculus

```
struct s0 {
  var p0: [[Double]]
  var p1: [[Double]]
}
@inline(never)
func f0(_ v0: s0, _ v1: Double) -> Double {
  var v4: s0 = v0
  let v8: [[Double]] = v4.p1
  let v14: [Double] = v8[0]
  let v13: Double = v14[0]
  v4.p0 = v8
  var v18: Double = v13
  v4.p0 = v8
  let v27: Double = v14[0]
  let v73: Double = v18 - v27
  v4.p0 = v8
  let v175: Double = v73 + v27
  return v175
}
```

Listing 3 A randomly generated benchmark (Swift).

7.2. Benchmarks

We evaluate the aforementioned languages and compilers on two groups of benchmarks:

Synthetic benchmarks. Our primary set of benchmarks is composed of randomly generated programs with varying lines of codes and number of mutating instructions. These programs are produced by a fuzzer that builds correct-by-construction programs as language-agnostic ASTs that are eventually transformed to source code by language-specific translators.

The fuzzer is able to build programs that use all the constructs present in Swiftlet except higher-order functions and existential containers, while the translators produce idiomatic code in each target language. For the purpose of our evaluation, we implemented translators for Swift; Scala, a functional language; and C++, an imperative language. Arrays are mapped to `scala.collection.immutable.Vector` in Scala and `std::vector` in C++. Structures are mapped to structs in C++ and to immutable case classes in Scala.

Generated programs are free from unintended sharing of mutable state. That guarantee is trivial to establish in Swift, Swiftlet, and Scala, thanks to MVS and pure functional programming. In C++, mutable references across function boundaries are prohibited. For instance, a function `void f(std::vector<T>& ar, T& el)` is considered illegal, as `el` may alias an element in `ar`.

Each benchmark is built as a variation of the same program archetype. First, it generates a large data structure composed of structs and arrays at different depths, whose leaves are floating-point numbers. Then, it traverses this data structure, mutating the nodes and performing arithmetic operations on the leaves. The traversal is directed by calls to non-recursive functions that operate on a specific part of the whole data structure. Some of these functions are exempt from inlining to ensure some call overhead is factored into our results. Listing 3 shows a program generated in Swift.

The overall complexity of a randomly generated program is approximated by counting the number of dynamically executed instructions. We use that heuristic to exclude benchmarks exceeding a threshold and keep program inputs that can terminate in a reasonable time. Finally, the number of mutating instructions can be adjusted by modifying the weights used by the fuzzer to select the constructs that it generates.

Well-known micro-benchmarks. Additionally, we also port 5 well-known benchmarks by Marr et al. (2016): Bounce, Mandelbrot, NBody, Permute and Queens. This choice is dictated by the limited set of features in Swiftlet. The benchmarks we picked are implemented using imperative language constructs such as arrays in Swift/Swiftlet/Scala and `std::vector` in C++.

7.3. Results of synthetic benchmarks

We report results for a data set measuring the execution time of 1344 randomly generated programs across 20 different independent runs. We normalize execution times by the fastest implementation per benchmark and report aggregated scores (Figure 14). We also report 50 percentile normalized time across all benchmarks classified by the number of mutations as fraction of all total memory accesses (Figure 15).

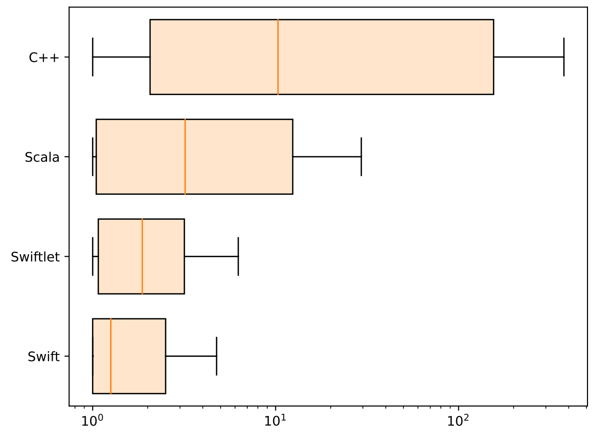


Figure 14 Running times (x-axis) normalized by the fastest implementation across all synthetic benchmarks.

Results show that C++ performs relatively poorly in read-heavy benchmarks, due to the cost of copying large data structures. Indeed, each read of `std::vector` copies an entire element, potentially causing repeated copies in deeply nested data structures. On the other hand, C++ is usually the fastest implementation in programs that are dominated by writes.

In contrast, Swift is the fastest language in the overwhelming majority of the benchmarks. Just as C++, Swiftlet allows in-place updates while the use of copy-on-write mitigates the cost of copying large data structures. Although our own implementation does not match Swift's performance (we explain the gap below), remark that Swiftlet also outperforms C++.

Scala relies on persistent data structures (Odersky & Moors 2009; Stucki et al. 2015) to model updates to immutable reference types. Overall, it performs remarkably well compared to C++, only outperformed on programs consisting of a significant number of mutating operations (> 70%). Those benchmarks

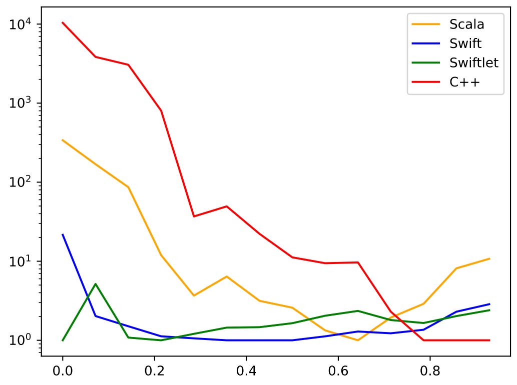


Figure 15 Normalized running times (y-axis) relative to ratio of writes as fraction of all memory accesses (x-axis) across all synthetic benchmarks.

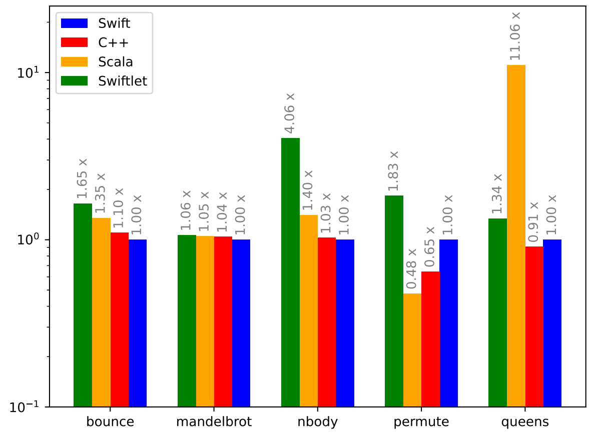


Figure 16 Normalized running times (y-axis) on micro-benchmarks.

underline the advantage of in-place updates in Swift, Swiftlet, and C++. Further, read-heavy benchmarks reveal additional overhead for traversing non-contiguously allocated storage.

In summary, those results provide satisfying answers to our earlier questions. Swift proved slower than C++ only for programs with extremely large number of mutating operations (> 90%), providing compelling empirical evidence in favor of copy-on-write, in combination with other optimizations facilitated by the use of Swiftlet. Meanwhile, Swift outperformed Scala in overwhelming majority of benchmarks, confirming the relevance of Swiftlet to sidestep the performance overhead introduced by functional updates.

7.4. Results of micro-benchmarks

Micro-benchmarks provide another perspective on the performance differences between the languages. We record 1000 in-process iterations across 10 independent runs. Each run discards first 900 measurements as warm-up. Finally, we report 50 percentile time across last 100 iterations across all runs (Figure

16).

Overall, both Swift and C++ perform the best, presenting extremely similar performance profiles. Since the benchmarks do not pass large data structures by value, they avoid the pathological slow-down that we have observed on synthetic benchmarks in C++.

Even though we wrote benchmarks in an imperative style, Scala does not consistently match the performance of Swift and C++. In particular, Scala lacks support for composite value types and, leading to a sub-optimal memory layout that requires additional pointer indirections (Bounce, NBody). Queens highlights a pathalogical case of early returns being compiled inefficiently (which are not idiomatic in an expression-oriented language).

Swiftlet lags behind Swift on benchmarks that depend on heavy mutation of arrays (Bounce, NBody, Permute). Although we implemented an extremely similar approach to compilation, Swiftlet's compiler lacks optimization passes that can reason about uniqueness of reference counts, compelling execution to run more runtime checks on reference counters.

7.5. Closing the gap between Swift and Swiftlet

Although Swiftlet shows competitive results, it is significantly slower than Swift in most benchmarks. This shortcoming can be explained in large part by a key missing optimization.

In spite of its advantages, copy-on-write comes at the cost of runtime checks preceding every mutation, thus impacting performance negatively. Fortunately, some of these checks can be removed with static reasoning. For instance, the compiler can prove the uniqueness of a particular value once and for all in a sequence of assignments without control flow, eliminating the need to check for uniqueness after the first mutating operation.

More generally, optimizers can reason statically about the value of a reference counter by tracking pairs of increment and decrement in a control flow graph. Additionally, compilers may generate faster code paths for uniquely referred data structures along with a default slow path, enabling further downstream optimization in the former case (Ullrich & de Moura 2019).

In Swift, these techniques are implemented in a dedicated optimizer. Manual modification of the LLVM bitcode generated by our compiler reveals that unnecessary reference counter checks account for most of the performance loss that we observed. The remainder of the gap is due to more aggressive use of SROA, thanks to a cleverer escape analysis, as well as the static allocation of constant arrays.

8. Related work

There exists a vast body of work dedicated to the enforcement of local reasoning and its impact on software performance. This section reviews some of the most related work in these areas.

Type-based approaches aimed at taming reference aliasing have received a lot of attention in recent years. A significant part of the research effort, however, is set in the context of languages based primarily on reference semantics. Ownership types (Clarke et al. 2013), for instance, bake aliasing restrictions into references by attaching them to *ownership contexts*, typically represented by other references. Other influential proposals, such as Gordon et al. (2012)'s type system for uniqueness and immutability or Naden et al. (2012)'s type system for borrowing, are also built on top of reference semantics. Further, although the concept of independent values appear in early work on aliasing protection mechanisms (Noble et al. 1998), value semantics is typically confined to fully immutable types.

Nonetheless, we observe that MVS share one key insight with these approaches: it dissociates the knowledge of a location from the permission to access it. In capability-based systems (Smith et al. 2000), that permission and its extent (read or write) is granted by a "token" usually treated as a linear resource at the type-system level and erased at runtime. In Swiftlet, that permission is tied to a path and its mutability.

The benefits of MVS with respect to correctness, safety, and performance can be traced back to Baker (1992)'s Linear Lisp. Building on Wadler (1990)'s linear types, Linear Lisp offered static garbage collection, promised freedom from data race and addressed concerns of efficiency by the means of a constant pool (Baker 1994). Despite their advantages, however, linear types impose a relatively constraining programming style in which every value must be used *exactly once*, requiring particular care to deal with conditional code. In response, Tov & Pucella (2011) proposes to relax the linearity constraint with affine resources, which can be used *at most once*.

Rust (Matsakis & Klock 2014) is certainly a heir of ownership types, affine types and capability-based systems, unifying these ideas into a coherent, high performance programming language. Ultimately, Rust's insistence on the uniqueness of mutating references pursues the same goal as MVS: local reasoning. Hence, it is no wonder that attempts to formalize its semantics (Weiss et al. 2019; Jung et al. 2018) typically draw inspiration from separation logic (Reynolds 2002). The similarity does not end here. As we already mentioned, Rust's borrowing is similar in nature to the way arguments can be passed in Swiftlet. Mutable borrows are operationally identical to `inout` arguments, and immutable borrows correspond to the way we optimize pass-by-value semantics. The difference lies in the way Rust and Swiftlet guarantee soundness. The latter allows syntactic enforcement of frame-based reasoning, as references are not surfaced in the user's programming model, while the former leverages a more sophisticated type system for the sake of expressiveness.

Project Valhalla (Simms 2019) is an ongoing effort to bring *inline types* into Java. Inline types do not have a default identity, as opposed to regular objects in Java, and their composition describes whole/part relationships. Just as our structs, they are allocated inline (hence the name) to better match the hardware's memory model and unlock more aggressive optimizations. Inline types are, however, immutable. Similar features exist in a number of "reference-oriented" languages, such as Python, Kotlin, or Scala, to cite a few.

The C# programming language (Microsoft Corporation 2021) supports mutable value types alongside with reference types. Their in-memory representation is comparable to the approach we developed in this paper. One important difference relates to the interaction between C#'s value types and interfaces, the latter being tied to reference semantics, sometimes leading to counter-intuitive situations (Steimann 2021). Swift addresses that issue with value witnesses, implementing different copy behaviors for reference types and value types.

Significant effort has been poured into techniques that optimize functional updates. One well-established approach is fusion (Johann 2003), a process aimed at eliminating intermediate data structures from expressions written as compositions of functions. Fusion, however, cannot eliminate all intermediate structures, in particular when they are accessed by multiple consumers. In that case, allocating and reclaiming temporary space may incur a significant overhead. Shaikhha et al. (2017) propose to address this shortcoming by rewriting programs in a *destination passing style* to guarantee efficient downstream stack-like allocation and compile-time garbage collection. In Swiftlet, intermediate structures can be removed altogether using `inout` to perform in-place part-wise mutation, or by relying on optimizations to substitute aliases for copies (Section 6).

Reinking et al. (2021) advocate for the use of reference counting as an automatic garbage collection mechanism to allow efficient in-place updates of unique data structures, using borrowed references to reduce reference counter updates (Ullrich & de Moura 2019). Unlike our naive implementation of copy-on-write, their framework is able to generate faster code paths when reference counts can be tracked statically.

9. Future work

This paper focuses on a single threaded execution model, yet concurrent and parallel applications have become ubiquitous. Fortunately, MVS offers promising prospects in that area. Specifically, MVS is immune to data races—a condition in which two or more threads access the same memory location concurrently—and provides a simple yet powerful framework to reason locally about concurrent programs, akin to concurrent separation logic (Brookes & O'Hearn 2016). One future direction is, therefore, to explore implementation strategies that leverage MVS to support efficient parallelization.

Part of Swift's concurrency model is based on actors (Agha 1990) with reference semantics. This choice seems appropriate in light of the vast literature on actor-based concurrency, while languages such as Pony (Clebsch et al. 2015) or Encore (Brandauer et al. 2015) already present compelling arguments in favor of type-based aliasing restrictions for memory safety. Nonetheless, revisiting concurrency without compromising on the constraints of MVS with respect to first-class references is an exciting challenge.

For the sake of conciseness, Swiftlet leaves out *protocols*, the construct that Swift uses to define constraints on generic types (Racordon & Buchs 2020). Protocols, however, present a number of interesting issues to generate efficient code. One challenge, in particular, is to choose between monomorphisation and type erasure (Griesemer et al. 2020). The former approach involves generating multiple variants of the same generic code, specialized for different concrete types. The latter involves settling for a common representation, typically by introducing indirections (e.g., boxing).

The best solution is likely a clever combination of both approaches. Hence, extending Swiftlet to study these aspects is another interesting direction for future work.

10. Conclusion

We discuss implementation strategies to compile programming languages featuring mutable value semantics, a paradigm that supports local reasoning by upholding the notion of value and excluding references from the user's programming model. These strategies are inspired by the Swift programming language, which leverages the benefits of MVS for safety, correctness and efficiency. To illustrate the details of our implementation, we introduce Swiftlet, a subset of Swift that focus on features essential for MVS, through a series of informal examples as well as a formal operational semantics. Swiftlet supports compounds of heterogeneous data, dynamically sized lists, type-erased containers, and closures.

We discuss a handful of simple yet efficient static and dynamic optimization techniques to eliminate unnecessary copies. Finally, we evaluate the performance of MVS on a large set of randomly generated programs with varying numbers of mutating operations, comparing Swift, Swiftlet, Scala and C++. Our results provide empirical evidence in favor of MVS. Specifically, they show that copy-on-write offers compelling performance gain and they highlight the benefits of in-place, part-wise mutation over functional updates in programs with large number of writes.

Acknowledgments

The authors would like to thank the reviewers for their helpful comments and suggestions.

References

Agha, G. A. (1990). *Actors - A model of concurrent computation in distributed systems.* MIT Press.

Apple Inc. (2021). *The swift programming language.* https://docs.swift.org/swift-book/. (Retrieved September 20, 2021)

Baker, H. G. (1992). Lively linear lisp: "look ma, no garbage!". *ACM SIGPLAN Notices*, *27*(8), 89–98. Retrieved from https://doi.org/10.1145/142137.142162 doi: 10.1145/142137.142162

Baker, H. G. (1994). Linear logic and permutation stacks - the forth shall be first. *SIGARCH Comput. Archit. News*, *22*(1), 34–43. Retrieved from https://doi.org/10.1145/181993.181999 doi: 10.1145/181993.181999

Bierema, N. (2022). *Immutable.js.* https://github.com/immutable-js/immutable-js. (Retrieved January 10, 2022)

Brandauer, S., Castegren, E., Clarke, D., Fernandez-Reyes, K., Johnsen, E. B., Pun, K. I., ... Yang, A. M. (2015). Parallel objects for multicores: A glimpse at the parallel language encore. In M. Bernardo & E. B. Johnsen (Eds.), *Formal methods for multicore programming* (Vol. 9104, pp. 1–56). New York, NY: Springer. Retrieved from https://doi.org/10.1007/978-3-319-18941-3_1 doi: 10.1007/978-3-319-18941-3_1

Brookes, S., & O'Hearn, P. W. (2016). Concurrent separation logic. *ACM SIGLOG News*, *3*(3), 47–65. Retrieved from https://dl.acm.org/citation.cfm?id=2984457

Clarke, D., Östlund, J., Sergey, I., & Wrigstad, T. (2013). Ownership types: A survey. In D. Clarke, J. Noble, & T. Wrigstad (Eds.), *Aliasing in object-oriented programming. types, analysis and verification* (Vol. 7850, pp. 15–58). New York, NY: Springer. Retrieved from https://doi.org/10.1007/978-3-642-36946-9_3 doi: 10.1007/978-3-642-36946-9_3

Clebsch, S., Drossopoulou, S., Blessing, S., & McNeil, A. (2015). Deny capabilities for safe, fast actors. In E. G. Boix, P. Haller, A. Ricci, & C. Varela (Eds.), *Programming based on actors, agents, and decentralized control* (pp. 1–12). New York, NY: ACM. Retrieved from https://doi.org/10.1145/2824815.2824816 doi: 10.1145/2824815.2824816

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., & Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, *13*(4), 451–490. Retrieved from https://doi.org/10.1145/115372.115320 doi: 10.1145/115372.115320

Felleisen, M., Friedman, D. P., Kohlbecker, E. E., & Duba, B. F. (1987). A syntactic theory of sequential control. *Theoretical Computer Science*, *52*, 205–237. Retrieved from https://doi.org/10.1016/0304-3975(87)90109-5 doi: 10.1016/0304-3975(87)90109-5

Fruja, N. G. (2004). The correctness of the definite assignment analysis in c#. *J. Object Technol.*, *3*(9), 29–52. Retrieved from https://doi.org/10.5381/jot.2004.3.9.a2 doi: 10.5381/jot.2004.3.9.a2

Gordon, C. S., Parkinson, M. J., Parsons, J., Bromfield, A., & Duffy, J. (2012). Uniqueness and reference immutability for safe parallelism. In G. T. Leavens & M. B. Dwyer (Eds.), *Object-oriented programming, systems, languages, and applications* (pp. 21–40). New York, NY: ACM. Retrieved from https://doi.org/10.1145/2384616.2384619 doi: 10.1145/2384616.2384619

Griesemer, R., Hu, R., Kokke, W., Lange, J., Taylor, I. L., Toninho, B., ... Yoshida, N. (2020). Featherweight go. *Proc. ACM Program. Lang.*, *4*(OOPSLA), 149:1–149:29. Retrieved from https://doi.org/10.1145/3428217 doi: 10.1145/3428217

Haller, P., & Odersky, M. (2010). Capabilities for uniqueness and borrowing. In T. D'Hondt (Ed.), *European conference on object-oriented programming* (Vol. 6183, pp. 354–378). New York, NY: Springer. Retrieved from https://doi.org/10.1007/978-3-642-14107-2_17 doi: 10.1007/978-3-642-14107-2_17

Henglein, F., & Jørgensen, J. (1994). Formally optimal boxing. In H. Boehm, B. Lang, & D. M. Yellin (Eds.), *Conference record of popl'94: 21st ACM SIGPLAN-SIGACT symposium on principles of programming languages, portland, oregon, usa, january 17-21, 1994* (pp. 213–226). ACM Press. Retrieved from https://doi.org/10.1145/174675.177874 doi: 10.1145/174675.177874

Hutton, G., & Meijer, E. (1998). Monadic parsing in haskell. *J. Funct. Program.*, *8*(4), 437–444. Retrieved from http://journals.cambridge.org/action/displayAbstract?aid=44175

Jambor, M. (2010). The new intraprocedural scalar replacement of aggregates. In *Proceedings of the gcc developers' summit* (pp. 47–54).

Johann, P. (2003). Short cut fusion is correct. *J. Funct. Program.*, *13*(4), 797–814. Retrieved from https://doi.org/10.1017/S0956796802004409 doi: 10.1017/S0956796802004409

Jung, R., Jourdan, J., Krebbers, R., & Dreyer, D. (2018). Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, *2*(POPL), 66:1–66:34. Retrieved from https://doi.org/10.1145/3158154 doi: 10.1145/3158154

Lattner, C., & Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization* (pp. 75–88). San Jose, CA, USA: IEEE. Retrieved from https://doi.org/10.1109/CGO.2004.1281665 doi: 10.1109/CGO.2004.1281665

Leroy, X., & Grall, H. (2009). Coinductive big-step operational semantics. *Inf. Comput.*, *207*(2), 284–304. Retrieved from https://doi.org/10.1016/j.ic.2007.12.004 doi: 10.1016/j.ic.2007.12.004

Marr, S., Daloze, B., & Mössenböck, H. (2016). Cross-language compiler benchmarking: are we fast yet? *ACM SIGPLAN Notices*, *52*(2), 120–131.

Matsakis, N. D., & Klock, F. S. (2014). The rust language. In M. Feldman & S. T. Taft (Eds.), *Conference on high integrity language technology, HILT* (pp. 103–104). ACM. Retrieved from https://doi.org/10.1145/2663171.2663188 doi: 10.1145/2663171.2663188

McCall, J. (2017). *Swift ownership manifesto.* https://github.com/apple/swift/blob/main/docs/OwnershipManifesto.md. (Retrieved Jan 9, 2022)

Microsoft Corporation. (2021). *C# documentation.* https://docs.microsoft.com/en-us/dotnet/csharp/. (Retrieved September 20, 2021)

Naden, K., Bocchino, R., Aldrich, J., & Bierhoff, K. (2012). A type system for borrowing permissions. In J. Field & M. Hicks (Eds.), *Principles of programming languages* (pp. 557–570). New York: ACM. Retrieved from https://doi.org/10.1145/2103656.2103722 doi: 10.1145/2103656.2103722

Noble, J., Vitek, J., & Potter, J. (1998). Flexible alias protection. In E. Jul (Ed.), *Ecoop'98 - object-oriented programming, 12th european conference, brussels, belgium, july 20-24, 1998, proceedings* (Vol. 1445, pp. 158–185). Springer. Retrieved from https://doi.org/10.1007/BFb0054091 doi: 10.1007/BFb0054091

Odersky, M., & Moors, A. (2009). Fighting bit rot with types (experience report: Scala collections). In R. Kannan & K. N. Kumar (Eds.), *Foundations of software technology and theoretical computer science* (Vol. 4, pp. 427–451). Saarland, Germany: Schloss Dagstuhl - Leibniz-Zentrum für Informatik. Retrieved from https://doi.org/10.4230/LIPIcs.FSTTCS.2009.2338 doi: 10.4230/LIPIcs.FSTTCS.2009.2338

O'Hearn, P. W., Reynolds, J. C., & Yang, H. (2001). Local reasoning about programs that alter data structures. In L. Fribourg (Ed.), *Computer science logic* (Vol. 2142, pp. 1–19). New York, NY: Springer. Retrieved from https://doi.org/10.1007/3-540-44802-0_1 doi: 10.1007/3-540-44802-0_1

O'Neill, M. E. (2009). The genuine sieve of eratosthenes. *Journal of Functional Programming*, *19*(1), 95–106. Retrieved from https://doi.org/10.1017/S0956796808007004 doi: 10.1017/S0956796808007004

Pierce, B. C. (2002). *Types and programming languages* (1st ed.). The MIT Press.

Potanin, A., Östlund, J., Zibin, Y., & Ernst, M. D. (2013). Immutability. In D. Clarke, J. Noble, & T. Wrigstad (Eds.), *Aliasing in object-oriented programming. types, analysis and verification* (Vol. 7850, pp. 233–269). Berlin: Springer. Retrieved from https://doi.org/10.1007/978-3-642-36946-9_9 doi: 10.1007/978-3-642-36946-9_9

R Core Team. (2020). R: A language and environment for statistical computing [Computer software manual]. Vienna, Austria. Retrieved from https://www.R-project.org/

Racordon, D., & Buchs, D. (2020). Featherweight swift: a core calculus for swift's type system. In R. Lämmel, L. Tratt, & J. de Lara (Eds.), *Proceedings of the 13th ACM SIGPLAN international conference on software language engineering, SLE 2020, virtual event, usa, november 16-17, 2020* (pp. 140–154). ACM. Retrieved from https://doi.org/10.1145/3426425.3426939 doi: 10.1145/3426425.3426939

Reinking, A., Xie, N., de Moura, L., & Leijen, D. (2021). Perceus: garbage free reference counting with reuse. In S. N. Freund & E. Yahav (Eds.), *PLDI '21: 42nd ACM SIGPLAN international conference on programming language design and implementation, virtual event, canada, june 20-25, 20211* (pp. 96–111). ACM. Retrieved from https://doi.org/10.1145/3453483.3454032 doi: 10.1145/3453483.3454032

Reynolds, J. C. (1998a). Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, *11*(4), 363–397. doi: 10.1023/A:1010027404223

Reynolds, J. C. (1998b). *Theories of programming languages*. Cambridge University Press.

Reynolds, J. C. (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* (pp. 55–74). IEEE Computer Society. Retrieved from https://doi.org/10.1109/LICS.2002.1029817 doi: 10.1109/LICS.2002.1029817

Rytz, L., Amin, N., & Odersky, M. (2013). A flow-insensitive, modular effect system for purity. In W. Dietl (Ed.), *Formal techniques for java-like programs* (pp. 4:1–4:7). New York, NY: ACM. Retrieved from https://doi.org/10.1145/2489804.2489808 doi: 10.1145/2489804.2489808

Saeta, B., Shabalin, D., Rasi, M., Larson, B., Wu, X., Schuh, P., ... Wei, R. (2021). *Swift for tensorflow: A portable, flexible platform for deep learning.*

Shabalin, D. (2020). *Just-in-time performance without warm-up* (Tech. Rep.). Lausanne, Swizerland: EPFL.

Shaikhha, A., Fitzgibbon, A. W., Jones, S. P., & Vytiniotis, D. (2017). Destination-passing style for efficient memory management. In P. Trinder & C. E. Oancea (Eds.), *Proceedings of the 6th ACM SIGPLAN international workshop on functional high-performance computing, fhpc@icfp 2017, oxford, uk, september 7, 2017* (pp. 12–23). ACM. Retrieved from https://doi.org/10.1145/3122948.3122949 doi: 10.1145/3122948.3122949

Siek, J., Lee, L.-Q., & Lumsdaine, A. (2002). *The boost graph library: User guide and reference manual*. USA: Addison-Wesley Longman Publishing Co., Inc.

Simms, D. (2019). *Valhalla.* https://wiki.openjdk.java.net/display/valhalla. (Retrieved September 20, 2021)

Smith, F., Walker, D., & Morrisett, G. (2000). Alias types. In G. Smolka (Ed.), *Programming languages and systems, 9th european symposium on programming, ESOP 2000, held as part of the european joint conferences on the theory and practice of software, ETAPS 2000, berlin, germany, march 25 - april 2, 2000, proceedings* (Vol. 1782, pp. 366–381). Berlin: Springer. Retrieved from https://doi.org/10.1007/3-540-46425-5_24 doi: 10.1007/3-540-46425-5_24

Steimann, F. (2021). The kingdoms of objects and values. In *Onward!* ACM.

Stepanov, A., & McJones, P. (2009). *Elements of programming* (1st ed.). Boston, MA: Addison-Wesley Professional.

Stepanov, A., & Rose, D. E. (2014). *From mathematics to generic programming* (1st ed.). Boston, MA: Addison-Wesley Professional.

Strachey, C. S. (2000). Fundamental concepts in programming languages. *High. Order Symb. Comput.*, *13*(1/2), 11–49. Retrieved from https://doi.org/10.1023/A:1010000313106 doi: 10.1023/A:1010000313106

Stucki, N., Rompf, T., Ureche, V., & Bagwell, P. (2015). RRB vector: a practical general purpose immutable sequence. In K. Fisher & J. H. Reppy (Eds.), *Proceedings of the 20th ACM SIGPLAN international conference on functional programming, ICFP 2015, vancouver, bc, canada, september 1-3, 2015* (pp. 342–354). ACM. Retrieved from https://doi.org/10.1145/2784731.2784739 doi: 10.1145/2784731.2784739

Tofte, M., Birkedal, L., Elsman, M., & Hallenberg, N. (2004). A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, *17*(3), 245–265. Retrieved from https://doi.org/10.1023/B:LISP.0000029446.78563.a4 doi: 10.1023/B:LISP.0000029446.78563.a4

Tov, J. A., & Pucella, R. (2011). Practical affine types. In T. Ball & M. Sagiv (Eds.), *Proceedings of the 38th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2011, austin, tx, usa, january 26-28, 2011* (pp. 447–458). ACM. Retrieved from https://doi.org/10.1145/1926385.1926436 doi: 10.1145/1926385.1926436

Turner, J. (2017). *Rust 2017 survey results.* https://blog.rust-lang.org/2017/09/05/Rust-2017-Survey-Results.html. (Retrieved April 8, 2021)

Ullrich, S., & de Moura, L. (2019). Counting immutable beans: reference counting optimized for purely functional programming. In J. Stutterheim & W. Chin (Eds.), *IFL '19: Implementation and application of functional languages, singapore, september 25-27, 2019* (pp. 3:1–3:12). ACM. Retrieved from https://doi.org/10.1145/3412932.3412935 doi: 10.1145/3412932.3412935

Vitek, J., & Bokowski, B. (2001). Confined types in java. *Software: Practice and Experience*, *31*(6), 507–532. Retrieved from https://doi.org/10.1002/spe.369 doi: 10.1002/spe.369

Wadler, P. (1990). Linear types can change the world! In M. Broy & C. B. Jones (Eds.), *Programming concepts and methods* (p. 561). Amsterdam, Netherlands: North-Holland.

Weiss, A., Patterson, D., Matsakis, N. D., & Ahmed, A. (2019). Oxide: The essence of rust. *CoRR*, *abs/1903.00982*. Retrieved from http://arxiv.org/abs/1903.00982

Wright, A. K., & Felleisen, M. (1994). A syntactic approach to type soundness. *Information and Computation*, *115*(1), 38–94. Retrieved from https://doi.org/10.1006/inco.1994.1093 doi: 10.1006/inco.1994.1093

About the authors

Dimitri Racordon is a post-doctoral researcher at the University of Geneva in Switzerland, and Northeastern University in the United States. His current research focuses on type-based approaches to memory safety, as well as language designs for safe and efficient concurrency. You can contact him at dimitri.racordon@unige.ch or visit https://kyouko-taiga.github.io.

Denys Shabalin is a software engineer at Google Research. His current research focuses on applying programming language research techniques to systems and compilers for machine learning. You can contact him at shabalin@google.com.

Daniel Zheng is a software engineer at Google Research. Previously, he worked in programming languages for machine learning on the Swift for TensorFlow project, focusing on differentiable programming for Swift. Currently, he works on machine learning for source code. You can contact him at danielzheng@google.com or visit https://danzheng.me.

Dave Abrahams is a Principal Scientist at Adobe's Software Technology Lab. He is a contributor to the C++ standard, a founding contributor to the Boost C++ library and a key designer of the Swift programming language. Before joining Adobe, he spent two years at Google extending Swift for machine learning with Swift for TensorFlow, and before that, seven years at Apple creating Swift and SwiftUI. You can contact him at dave@boostpro.com.

Brennan Saeta is a software engineer at Google Research. He previously led the Swift for TensorFlow project which explored customizing programming languages for machine learning. Currently, he works on JAX, a library for machine learning. You can contact him at saeta@google.com.

A. Proof of type soundness

We describe a handful of supporting lemma. The first states that if a path a_i does not overlap another path a_j in some typing context Γ, then they cannot denote overlapping memory locations.

Lemma A.1. Let $\pi;\overline{\eta}$ be a well-formed memory state such that $\Delta \vdash \pi;\overline{\eta} : \Gamma$. Let a_1 and a_2 be two well-typed argument expressions such that $\Delta;\Gamma;\pi \vdash a_1 : \tau_1$ and $\Delta;\Gamma;\pi \vdash a_2 : \tau_2$. If $\Delta \vdash \pi;\overline{\eta};a_i \longrightarrow \pi;\overline{\eta};l_i$, and $\Delta \vdash \pi;\overline{\eta};a_j \longrightarrow \pi;\overline{\eta};l_i$, then $a_i \nsubseteq a_j \implies acc(\pi, l_i) \cap acc(\pi, l_j) = \varnothing$.

Proof of Lemma A.1. The proof is obvious from the fact that paths denote roots of tree-shaped data structures, and that well-formed memory state preserve the uniqueness of each binding. Hence, the location denoted by the root x_i of a path cannot overlap with that of another root x_j. It follows that, unless $a_i \subseteq a_j$, a_i cannot denote an ancestor of a_j. □

The next lemmas state that copying a value preserves its type and the well-typedness of a program state.

Lemma A.2. If $\Delta;\Gamma;\pi \vdash v : \tau$ and $\pi', v' = copy(\pi, v)$ then $\Delta;\Gamma;\pi' \vdash v' : \tau$.

Proof of Lemma A.2. By induction on the recursive definition of $copy(\pi, v)$. □

Lemma A.3. If $\Delta;\Gamma;\pi \vdash v : \tau$, and $\Delta \vdash \pi;\overline{\eta} : \Gamma$, and $\pi', v' = copy(\pi, v)$ then $\Delta \vdash \pi';\overline{\eta} : \Gamma$.

Proof of Lemma A.3. By induction on the recursive definition of $copy(\pi, v)$ and the fact that *copy* preserves well-typedness. □

The typing semantics from Section 4.4 does not have any rule for pop expressions, since those are only defined in the small-step semantics. To prove soundness inductively, though, we must define an additional rule:

$$\textsc{T-Pop}\quad \frac{\Delta;\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash e\,\texttt{pop}\;\overline{l} : \tau}$$

Finally, we prove progress and preservation.

Proof of Lemma 4.1. The proof is by induction on the typing derivation of e.

case v: Trivial, v is a value.

case r: The type derivation is governed by T-READ and must have the form

$$\frac{\Delta;\Gamma \vdash_{path} r : m\ \tau}{\Delta;\Gamma \vdash r : \tau}$$

There are three sub-cases to consider:

sub-case x: The typing derivation is governed by T-BINDINGREF. By the fact that $\Delta \vdash \pi;\overline{\eta} : \Gamma$, we know that $x \in dom(\eta_1)$. Then by ESS-NAME we have $\Delta \vdash \pi;\overline{\eta};x \longrightarrow \pi';\overline{\eta};v$.

sub-case $e_1[e_2]$: The typing derivation is governed by T-LETELEMREF or T-VARELEMREF. By the fact that $\Delta \vdash \pi;\overline{\eta} : \Gamma$, we know that $e_1 : [\tau]$ and $e_2 : \mathbb{Z}$. Then, there are four situations to consider:

1. If e_1 and e_2 are values, then e_1 is an array instance of the form $[l_1, \dots, l_k]$. If e_2 is a number c such that $0 \leq c < k$, then by E-ELEM we have $\Delta \vdash \pi; \overline{\eta}; [\bar{l}][c] \longrightarrow \pi'; \overline{\eta}; v$.
2. If e_1 and e_2 are values, but e_2 is not a number, or is not in the range $0 \leq c < k$, then the evaluation is stuck at an invalid array subscript.
3. If e_1 is a value but e_2 is not, then by ESS-CONTEXT we have $\Delta \vdash \pi; \overline{\eta}; v[e_2] \longrightarrow \pi'; \overline{\eta}'; v[e_2']$.
4. If e_1 is not a value, then by ESS-CONTEXT we have $\Delta \vdash \pi; \overline{\eta}; e_1[e_2] \longrightarrow \pi'; \overline{\eta}'; e_1'[e_2]$.

sub-case $e.x$**:** The typing derivation is governed by T-LETPROPREF or T-VARPROPREF. By the fact that $\Delta \vdash \pi; \overline{\eta} : \Gamma$, we know that $e : s$ and s is a field of s.

1. If e_1 is a value, then it is a structure instance of the form $[\bar{l}]^s$ and by ESS-PROP we have $\Delta \vdash \pi; \overline{\eta}; [\bar{l}]^s.x \longrightarrow \pi'; \overline{\eta}'; v$.
2. If e_1 is not a value, then by ESS-CONTEXT we have $\Delta \vdash \pi; \overline{\eta}; e.x \longrightarrow \pi'; \overline{\eta}'; e'.x$.

case $[\bar{e}]$**:** The typing derivation is governed by T-ARRAYLIT.

1. If all elements in $\bar{e}$ are values, then by ESS-ARRAYLIT we have $\Delta \vdash \pi; \overline{\eta}; [\bar{v}] \longrightarrow \pi'; \overline{\eta}'; [\bar{l}]$.
2. If there exists e_i that is not a value, then we know that e_j for all $1 \leq j < i$ are values and by ESS-CONTEXT we have $\Delta \vdash \pi; \overline{\eta}; [v_1, \dots, v_{i-1}, e_i, e_{i+1} \dots e_k] \longrightarrow \pi'; \overline{\eta}'; [v_1, \dots, v_{i-1}, e', e_{i+1} \dots e_k]$

case $s(e_1, \dots, e_k)$**:** The typing derivation is governed by T-STRUCTLIT. By the fact that $\Delta; \Gamma; \pi \vdash s(\bar{e}) : \tau$, we know that $\texttt{struct}\ s\{m_1\ x_1 : \tau_1, \dots, m_k\ x_k : \tau_k\} \in \Delta$.

1. If all elements $e_1, \dots, e_k$ are values, then by ESS-STRUCTLIT we have $\Delta \vdash \pi; \overline{\eta}; s(\bar{v}) \longrightarrow \pi'; \overline{\eta}'; [\bar{l}]^s$.
2. If there exists e_i that is not a value, then we know that e_j for all $1 \leq j < i$ are values, and by ESS-CONTEXT we have $\Delta \vdash \pi; \overline{\eta}; s(v_1, \dots, v_{i-1}, e_i, e_{i+1} \dots e_k) \longrightarrow \pi'; \overline{\eta}'; s(v_1, \dots, v_{i-1}, e_i', e_{i+1} \dots e_k)$

case $\texttt{func}\ x_0(\overline{x : p}) \to \tau_\lambda\ \{[y_1, \dots, y_h]\ \texttt{in}\ e_1\}\ \texttt{in}\ e_2$**:** The typing derivation is governed by T-FUNC. By the fact that $\Delta \vdash \pi; \overline{\eta} : \Gamma$, we know that $\{y_1, \dots, y_h\} \in dom(\eta_))$. Therefore $mkenv(\pi, y_1 : \eta(y_1), \dots, y_h : \eta(y_h))$ is defined. Then by ESS-FUNCLIT we have $\Delta \vdash \pi; \overline{\eta}; \texttt{func}\ x_0(\overline{x : p}) \to \tau_\lambda\ \{[y_1, \dots, y_h]\ \texttt{in}\ e_1\}\ \texttt{in}\ e_2 \longrightarrow \pi''; \mu'', \overline{\eta}; e_2; \texttt{pop}\ l$

case $e(a_1, \dots, a_k)$**:** The typing derivation is governed by T-CALL. We know that $e : (p_1, \dots, p_k) \to \tau$. Further, by Lemma A.1 and the fact that $\Delta; \Gamma; \pi \vdash e(a_1, \dots, a_k) : \tau$, we know that `inout` arguments do not denote overlapping memory locations.

1. If e is a value $\lambda(\overline{x : p}, \eta_\lambda, e_\lambda)$, then by induction hypothesis, for all elements a_i, either a_i is a value, or the program can take a step.
   (a) If all arguments $a_1, \dots, a_k$ are values, then by ESS-CALL we have $\Delta \vdash \pi; \overline{\eta}; v_0(\bar{v}) \longrightarrow \pi'; \eta', \overline{\eta}'; e_\lambda; \texttt{pop}\ \{l_i \mid i \in I_{cpy}\}$.
   (b) If there exists a_i that is not a value, then we know that a_j for all $1 \leq j < i$ are values and the type derivation for a_i must have either of the following form, depending on whether a_i is a regular argument e or an `inout` argument $\&r$.

$$\frac{\Delta; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash_{arg} e : \tau} \qquad \frac{\Delta; \Gamma \vdash_{path} r : \texttt{var}\ \tau}{\Delta; \Gamma \vdash_{arg} \&r : \texttt{inout}\ \tau}$$

   i. If $a_i \equiv e$, then by induction hypothesis e takes a step.
   ii. If $a_i \equiv \&r$, and r is a location, then by ESS-INOUT applies.
   iii. If $a_i \equiv \&r$, and r is not a location, then r takes a step via $\longrightarrow_{lv}$.

   In all cases, by ESS-CONTEXT we have $\Delta \vdash \pi; \overline{\eta}; v_0(v_1, \dots, v_{i-1}, e_i, e_{i+1} \dots e_k) \longrightarrow \pi'; \overline{\eta}'; v_0(v_1, \dots, v_{i-1}, e_i', e_{i+1} \dots e_k)$
2. If e is not a value, then by ESS-CONTEXT we have $\Delta \vdash \pi; \overline{\eta}; e(\bar{a}) \longrightarrow \pi'; \overline{\eta}'; e'(\bar{a})$.

case $b = e_x\ \texttt{in}\ e$**:** The typing derivation is governed by T-BINDING.

1. If e_x is a value, then by ESS-BINDING we have $\Delta \vdash \pi; \overline{\eta}; b = v\ \texttt{in}\ e \longrightarrow \pi'; \overline{\eta}'; e; \texttt{pop}\ l$.
2. If e_x is not a value, then by ESS-CONTEXT we have $\Delta \vdash \pi; \overline{\eta}; b = e_x\ \texttt{in}\ e \longrightarrow \pi'; \overline{\eta}'; b = e_x'; e$.

case $r = e_r$**:** The typing derivation is governed by T-ASSIGN and must have the form

$$\frac{\Delta; \Gamma \vdash e_r : \tau \qquad \Delta; \Gamma \vdash_{path} r : \texttt{var}\ \tau}{\Delta; \Gamma \vdash r = e_r : [\bot]}$$

There are three situations to consider:

1. If e_r is a value and r is a location, then by ESS-ASSIGN we have $\Delta \vdash \pi; \overline{\eta}; \texttt{var}\ l = v \longrightarrow \pi'; \overline{\eta}'()$.
2. If e_r is not a value and r is a location, then by ESS-CONTEXT we have $\Delta \vdash \pi; \overline{\eta}; \texttt{var}\ l = e_r \longrightarrow \pi'; \overline{\eta}'; \texttt{var}\ l = e_r'$.
3. If r is not a location, then we have $\Delta \vdash \pi; \overline{\eta}; r = e_1 \longrightarrow_{lv} \pi'; \overline{\eta}'; r' = e_1$.

case $e_1\ ?\ e_2\ :\ e_3$**:** The typing derivation is governed by T-COND. There three two cases to consider.

1. If e_1 is a value different than 0, then by ESS-COND-T we have $\Delta \vdash \pi; \overline{\eta}; v\ ?\ e_2\ :\ e_3 \longrightarrow \pi; \overline{\eta}; e_2$.
2. If e_1 is the value 0, then by ESS-COND-F we have $\Delta \vdash \pi; \overline{\eta}; 0\ ?\ e_2\ :\ e_3 \longrightarrow \pi; \overline{\eta}; e_3$.
3. If e_1 is not a value, then by ESS-CONTEXT we have $\Delta \vdash \pi; \overline{\eta}; e_1\ ?\ e_2\ :\ e_3 \longrightarrow \pi'; \overline{\eta}'; e_1'\ ?\ e_2\ !\ e_3$.

case $e; \texttt{pop}\ \bar{l}$**:** The typing derivation is governed by T-POP. There are two cases to consider.

1. If e, then ESS-POP we have $\Delta \vdash \pi; \eta, \overline{etav}; \texttt{pop}\ \bar{l} \longrightarrow \pi'; \overline{\eta}; v$.
2. If e is not a value, then ESS-POP we have $\Delta \vdash \pi; \eta, \overline{etae}; \texttt{pop}\ \bar{l} \longrightarrow \pi'; \overline{\eta}'; e'; \texttt{pop}\ \bar{l}$.

case $e_1; e_2$**:** The typing derivation is governed by T-SEQ. There are two situations to consider:

1. If e_1 is a value, then by ESS-SEQ we have $\Delta \vdash \pi; v; e_2 \longrightarrow \pi; \overline{\eta}; e_2$.
2. If e_1 is not a value, then by ESS-CONTEXT we have $\Delta \vdash \pi; e_1; e_2 \longrightarrow \pi'; \overline{\eta}'; e_1'; e_2$.

case $e\ \texttt{as}\ \tau$**:** The typing derivation is governed by T-CAST. There are four situations to consider:

1. If e_1 is a value and $\tau = Any$, then by ESS-UPCAST we have $\Delta \vdash \pi; v\ \texttt{as}\ Any \longrightarrow \pi; \overline{\eta}; box(l)$.
2. If e_1 is a value $box(l)$ and $\tau \neq Any$ and $typeof(v) = \tau$, then by ESS-DOWNCAST we have $\Delta \vdash \pi; v\ \texttt{as}\ \tau \longrightarrow \pi'; \overline{\eta}; v$, where v' is a copy of the value at l.
3. If e_1 is a value $box(l)$ and $\tau \neq Any$ but $typeof(v) \neq \tau$, then evaluation is stuck at an invalid downcast.
4. If e_1 is not a value, then by ESS-CONTEXT we have $\Delta \vdash \pi; e_1\ \texttt{as}\ \tau \longrightarrow \pi'; \overline{\eta}'; e_1'$.

□

Proof of Lemma 4.2. The proof is by induction on the typing derivation of e.

case r**:** The type derivation is governed by T-READ and must have the form

$$\frac{\Delta; \Gamma \vdash_{path} r : m\ \tau}{\Delta; \Gamma \vdash r : \tau}$$

There are three sub-cases to consider:

sub-case x**:** The typing derivation is governed by T-BINDINGREF and we know that e steps with ESS-NAME. We pick $\Gamma' = \Gamma$. Since $\pi; \overline{\eta}$ is well-typed in Γ, we know that $\pi(\eta_1(x)) = m\ v$ and $\Delta; \Gamma; \pi \vdash v : \tau$, then by Lemma A.2 we have $\Delta; \Gamma'; \pi' \vdash v' : \tau$. Furthermore, by Lemma A.3 we have $\Delta \vdash \pi'; \overline{\eta} : \Gamma'$.

sub-case $e_1[e_2]$**:** The typing derivation is governed by T-LETELEMREF or T-VARELEMREF. Both state that we have the derivations $\Delta; \Gamma \vdash e_1 : [\tau]$ and $\Delta; \Gamma \vdash e_2 : \mathbb{Z}$. Since we know that $e_1[e_2]$ steps by assumption, there are only three situations to consider:

1. If e_1 and e_2 are values, then e_1 is an array instance of the form $[l_1, \ldots, l_k]$, and e_2 is a number c such that $0 \leq c < k$, and $e_1[e_2]$ steps with ESS-ELEM. We pick $\Gamma' = \Gamma$. We know that $\pi(l_c) = m\ v$ and $\Delta; \Gamma; \pi \vdash v : \tau$, then by Lemma A.2 we have $\Delta; \Gamma'; \pi' \vdash v' : \tau$. Furthermore, by Lemma A.3 we have $\Delta \vdash \pi'; \overline{\eta} : \Gamma'$.
2. If e_1 is a value but e_2 is not, then by induction hypothesis e_2 takes a step, its type is preserved, and the resulting program state is well-typed.
3. If e_1 is not a value, then by induction hypothesis e_1 takes a step, its type is preserved, and the resulting program state is well-typed.

sub-case $e.x$**:** The typing derivation is governed by T-LETPROPREF or T-VARPROPREF. Both state that we have the derivations $\Delta; \Gamma \vdash e : s$. Further, since $e.x$ is well-typed by assumption, s denotes a definition $\texttt{struct}\ s\{\overline{m\ x : \tau}\}$ and there exists $x_i = x$.

1. If e is a value, then e is an structure instance of the form $[\bar{l}]^s$ and we know that $e.x$ steps with ESS-PROP. We pick $\Gamma' = \Gamma$. We know that $\pi(l_i) = m\ v$ and $\Delta; \Gamma; \pi \vdash v : \tau$, then by Lemma A.2 we have $\Delta; \Gamma'; \pi' \vdash v' : \tau$. Furthermore, by Lemma A.3 we have $\Delta \vdash \pi'; \overline{\eta} : \Gamma'$.
2. If e is not a value, then by induction hypothesis e takes a step and its type is preserved.

case $[\bar{e}]$**:** The typing derivation is governed by T-ARRAYLIT. Since $[\bar{e}]$ is well-typed by assumption, all elements have type τ.

1. If all elements in $\bar{e}$ are values, then we know that $[\bar{e}]$ steps with ESS-ARRAYLIT. We pick $\Gamma' = \Gamma$. We know that all elements in $\bar{e}$ are values of type τ, then $\Delta; \Gamma; \pi' \vdash [\bar{l}] : [\tau]$, as requested. Furthermore, since $\bar{l} \notin dom(\pi)$, $[\bar{l}]$ is an independent value in π' and we have $\Delta \vdash \pi'; \overline{\eta} : \Gamma'$.
2. If there exists e_i that is not a value, then by induction hypothesis e_i takes a step and its type is preserved.

case $s(\bar{e})$**:** The typing derivation is governed by T-STRUCTLIT. Since $s(\bar{e})$ is well-typed by assumption, s denotes a definition $\texttt{struct}\ s\{\overline{m\ x : \tau}\}$ and each argument e_i has type τ_i.

1. If all elements in $\bar{e}$ are values, then we know that $s(\bar{e})$ steps with ESS-STRUCTLIT. We pick $\Gamma' = \Gamma$. We know that all elements in $\bar{e}$ are well-typed with respect to s's definition, then $\Delta; \Gamma; \pi' \vdash [\bar{l}]^s : s$, as requested. Furthermore, since $\bar{l} \notin dom(\pi)$, $[\bar{l}]^s$ is an independent value in π' and we have $\Delta \vdash \pi'; \overline{\eta} : \Gamma'$.
2. If there exists e_i that is not a value, then by induction hypothesis e_i takes a step and its type is preserved.

case $\texttt{func}\ x_0(\overline{x : p}) \rightarrow \tau_\lambda\ \{[y_1, \ldots, y_h]\ \texttt{in}\ e_1\}\ \texttt{in}\ e_2$**:** The typing derivation is governed by T-FUNC and evaluation steps with ESS-FUNC.

1. If e_2 is a value, we pick $\Gamma' = \Gamma[x_0 \mapsto (\bar{p}) \rightarrow \tau_\lambda]$ and we have $\Delta; \Gamma'; \pi'' \vdash v : \tau$, as requested.
2. If e_2 is not a value, then by induction hypothesis e_2 takes a step and its type is preserved.

case $e(\overline{a})$: The typing derivation is governed by T-CALL. Since $e(\overline{a})$ is well-typed by assumption, e has type $(\overline{p}) \to \tau$ and each argument a_i has type p_i.

1. If e is a value, then e is a function object of the form $\lambda(\overline{x : p}, \eta_\lambda, e_\lambda)$. Then the evaluation steps depends on the arguments:
   (a) If all elements in $\overline{a}$ are values, then we know that $e(\overline{a})$ steps with ESS-CALL.
      – We pick Γ' such that each parameter name x_i is mapped onto its type p_i, either mutably if $p_i = \texttt{inout}\ \tau_i$ for some τ_i, or immutably otherwise, and each name in $dom(\eta_\lambda)$ is mapped onto its type and mutability in π.
      – Since e is well-typed by assumption, we know that $\Delta \vdash \pi; \overline{\eta}; e(\overline{a}) \longrightarrow \pi'; \eta', \overline{\eta}; e_\lambda; \texttt{pop}\ \overline{l}$. We assume $\Delta; \Gamma'; \eta' \vdash e_\lambda; \texttt{pop}\ \overline{l}$ holds by induction.
      – Finally we must show that $\Delta \vdash \pi'; \eta_\lambda, \overline{\eta} : \Gamma$. Since e is well-typed, the free in e_λ are arguments in $\overline{x}$, inserted in Γ' using the argument list.

      Furthermore, since ESS-CALL checks for uniqueness of `inout` arguments, we know that parameter name is either mapped onto a new location in π' or its location cannot overlap with another location in π'. Hence, we have $\Delta \vdash \pi'; \eta', \overline{\eta} : \Gamma'$.
   (b) If there exists a_i that is not a value, then we know that a_j for all $1 \leq j < i$ are values and the type derivation for a_i must have either of the following form, depending on whether a_i is a regular argument e or an `inout` argument $\&r$.

$$\frac{\Delta; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash_{arg} e : \tau} \qquad \frac{\Delta; \Gamma \vdash_{path} r : \texttt{var}\ \tau}{\Delta; \Gamma \vdash_{arg} \&r : \texttt{inout}\ \tau}$$

      i. If $a_i \equiv e$, then by induction hypothesis e takes a step and its type is preserved.
      ii. If $a_i \equiv \&r$, and r is a location, then ESS-INOUT applies and r's type is preserved.
      iii. If $a_i \equiv \&r$, and r is not a location, then r takes a step via $\longrightarrow_{lv}$ and r's type is preserved.

      In all cases, $e(\overline{a})$ takes a step and its type is preserved.
2. If e is not a value, then by induction hypothesis e takes a step and its type is preserved.

case $b = e_1\ \texttt{in}\ e_2$: The typing derivation is governed by T-BINDING.

1. If e_1 is a value, then we know that $m\ x : \tau = e_1\ \texttt{in}\ e_2$ steps with ESS-BINDING.
   – We pick $\Gamma' = \Gamma[x \mapsto m\ \tau]$.
   – We assume $\Delta; \Gamma'; \pi' \vdash e_2 : \tau$ by induction.
   – Finally we must show that $\Delta \vdash \pi'; \eta', \overline{\eta} : \Gamma'$. We know that $dom(\pi') \setminus dom(\pi) = \{l\}$, and $\pi'(l) = m\ v_1$. We know that $\Delta \vdash \pi; \eta, \overline{\eta} : \Gamma$ by assumption, therefore we can show that $\Delta \vdash \pi'; \eta', \overline{\eta} : \Gamma'$ holds.
2. If e_1 that is not a value, then by induction hypothesis e_1 takes a step and its type is preserved.

case $r = e_r$**:** The typing derivation is governed by T-ASSIGN and must have the form

$$\frac{\Delta; \Gamma \vdash e_r : \tau \qquad \Delta; \Gamma \vdash_{path} r : \texttt{var}\ \tau}{\Delta; \Gamma \vdash r = e_r : ()}$$

There are three situations to consider:

1. If e_r is a value and r is a location, then we know that $r = e_r$ steps with ESS-ASSIGN. We know that r has type $\texttt{var}\ \tau$ and e has type $m\tau$ by assumption, therefore the typing context does not change. We pick $\Gamma' = \Gamma$ and we have $\Delta; \Gamma; \pi \vdash r = e_r : ()$, as requested.
2. If e_1 is not a value and r is a location, then by induction hypothesis e_1 takes a step and its type is preserved.
3. If r is not a location, then r takes a step via $\longrightarrow_{lv}$ and its type is preserved.

case $e_1\ ?\ e_2\ :\ e_3$**:** The typing derivation is governed by T-COND.

1. If e_1 is a value different than 0, then $e_1\ ?\ e_2\ :\ e_3$ takes a step with ESS-COND-T. We pick $\Gamma' = \Gamma$ and we have $\Delta; \Gamma; \pi \vdash e_2 : \tau$, as requested.
2. If e_1 is the value 0, then $e_1\ ?\ e_2\ :\ e_3$ takes a step with ESS-COND-F. We pick $\Gamma' = \Gamma$ and we have $\Delta; \Gamma; \pi \vdash e_3 : \tau$, as requested.
3. If e_1 is not a value, then by induction hypothesis e_1 takes a step and its type is preserved.

case $e; \texttt{pop}\ \overline{l}$**:** The typing derivation is governed by T-POP. There are two situations to consider:

1. If e is a value, then $e; \texttt{pop}\ \overline{l}$ takes a step with ESS-POP. We pick $\Gamma' = \Gamma$ and we have $\Delta; \Gamma'; \pi \vdash v : \tau$, as requested.
2. If e is not a value, then by induction hypothesis e takes a step and its type is preserved.

case $e_1; e_2$**:** The typing derivation is governed by T-SEQ. There are two situations to consider:

1. If e_1 is a value, then $v; e_2$ takes a step with ESS-SEQ. We pick $\Gamma' = \Gamma$ and we have $\Delta; \Gamma'; \pi \vdash e_2 : \tau$, as requested.
2. If e_1 is not a value, then by induction hypothesis e_1 takes a step and its type is preserved.

case e `as` τ**:** The typing derivation is governed by T-CAST. There are three situations to consider:

1. If v is a value and $\tau = \mathit{Any}$, we pick $\Gamma' = \Gamma$ and we have $\Delta; \Gamma'; \pi \vdash v : \mathit{Any}$, as requested.
2. If v is a value and $\tau \neq \mathit{Any}$ and $\mathit{typeof}(v) = \tau$, we pick $\Gamma' = \Gamma$ and we have $\Delta; \Gamma'; \pi \vdash v : \tau$, as requested.
3. If e is not a value, e takes a step and the type of e `as` τ is preserved.

□