A Formal Definition of Swift's Value Semantics

Racordon, Dimitri

A Formal Definition of Swift's Value Semantics

Work in Progress

Dimitri Racordon
Faculty of Sciences, University of Geneva
Geneva, Switzerland

November 24, 2020

Abstract

Swift is a general-purpose programming language designed as a modern substitute for C-based languages, such as C/C++ and Objective-C. As such, its semantics departs from that of most alternative based on a virtual machine. Specifically, Swift distinguishes between reference and value semantics. The former corresponds to the way most objects in Java-like languages behave, whereas the latter relates more to passive data structures in C/C++. The support of value semantics in concert with common object-orientation patterns gives rise to a number of subtleties with which inexperienced developers may not be familiar. In this work in progress, I attempt to shed light on these aspects with a formal description of Swift's value semantics.

Keywords— swift, value semantics, reference semantics, formal languages

1 Introduction

Swift[1] is a modern general-purpose programming language, developed by Apple, to be a successor to C-based languages, such as C/C++ and Objective-C. It is beingused in a broad spectrum of applications, obviously including graphical apps for iOS and macOS, but also in the context of server-side and machine learning development. This success can be attributed to its relatively rich type system on one front, which allows for clear and expressive APIs, and on another by its efficient memory model and seamless integration into the C ecosystem. Swift distinguishes between two different kinds of data types: references and values. The formers adopt the semantics traditionally associated with objects in Java-like languages, while the latters share more similarities with passive data structures (PDSs) in C/C++.

Consider for instance the following program:

```
1  struct Pair {
2    var fst: Int
3    var snd: Int
4  }
5
6  var x = Pair(fst: 2, snd: 3)
7  var y = x
8  y.fst = 4
9  print(x)
10 // Prints Pair(fst: 2, snd: 3)
```

A type `Pair` is declared at line 1, as a simple record with two members, `fst` and `snd`. At line 6, a variable `x` is declared and initialized with an instance of the type `Pair`. Line 7 declares a second variable `y` which is assigned to `x`'s value. Contrary to how Java would behave, this does *not* result in the creation of an alias. Instead, `y` is assigned to an independent copy. Hence, the mutation at line 8 does not impact `x`'s value, as demonstrated by the print statement at line 9.

Value semantics contributes significantly to Swift's expressiveness and provides developers with means to avoid some of the abstraction leaks introduced by traditional reference-based languages. They also provide the compiler with further avenues for optimizations, in particular by avoiding the dynamic overhead of heap-memory management, and by avoiding unnecessary pointer indirections to leverage CPU

[1] `https://swift.org`

caches more aggressively. Unfortunately, this comes at the cost of increased complexity. The language features two different semantics, which its syntax does not clearly distinguish. Furthermore, existing documentation is provided in the form of examples that sometimes fail to capture the full extent of the actual underlying semantics, compelling users to reason in terms of intuitions only.

This paper proposes to address this issue by the means of a formal specification of Swift's value semantics. More specifically, it presents ongoing work on a core programming language exposing the expected observational behavior of value types.

2 Preliminaries

This section introduces the mathematical concepts and notations that are used throughout this document.

Functions Let $f : A \to B$ be a function, $dom(f)$ and $codom(f)$ denote its domain and codomain, respectively. 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 preimages 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 number to its absolute value. We omit the subscript when the function's domain and codomain are obvious from the context. We write $f \ll [a \mapsto b]$ 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 $(f \ll [0 \mapsto 3])(0) = 3$ and $(f \ll [0 \mapsto 3])(1) = 2$. More formally:

$$\forall x \in dom(f) \cup \{a\}, (f \ll [a \mapsto b])(x) = \begin{cases} b & \text{if } x = a \\ f(x) & \text{otherwise} \end{cases}$$
$$dom(f \ll [a \mapsto b]) = dom(f) \cup \{a\}$$

Structural Operational Semantics We define the semantics of our language by the means of inference rules. An inference rule is a logical statement of the form:

$$\frac{p_1 \; p_2 \; \cdots \; p_n}{q}$$

where $p_1, p_2, \ldots, p_n$ constitute the rule's *premise*, and is composed of n *hypotheses*, while q denotes the rule's *conclusion*. The ensemble reads as "we can conclude q if all $p_1, p_n, \ldots, p_n$ hold". In other words, the rule states that $(p_1 \wedge \cdots \wedge p_n) \implies q$, where $p_1 \ldots p_n, q$ are predicate terms (e.g., $a > b$).

Inference rules are used to represent deductive systems with predicate logic, and serve to establish formal proofs through a process called *deduction*. Given a set of inference rules, deduction is a recursive process that consists of satisfying a proof obligation by first finding an inference rule that concludes it, and then proving that this rule's hypotheses hold, optionally producing new proof obligations. The process terminates when the set of proof obligations is empty.

As an example, consider the following three inference rules, describing the small-step semantics of a minimal language able to express integer divisions:

EVAL-LEFT
$$\frac{e_1 \rightsquigarrow e_1'}{div(e_1, e_2) \rightsquigarrow div(e_1', e_2)}$$

EVAL-RIGHT
$$\frac{e_2 \rightsquigarrow e_2'}{div(e_1, e_2) \rightsquigarrow div(e_1, e_2')}$$

DIV
$$\frac{n_2 \neq 0}{div(n_1, n_2) \rightsquigarrow n_1 \div n_2}$$

The first and second rules describe the reduction of the left and right operands, respectively, while the third rule computes the division of two integer values, as long as the divisor is different from 0. Together, these three rules can be used to deduce the result of any expression built from integer constants and a constructor *div* of arity 2. For instance, the evaluation of an expression $div(12, div(6, 2))$ could be characterized as follows:

$$div(12, div(6, 2)) \rightsquigarrow div(12, 3) \rightsquigarrow 4$$

The first reduction is achieved by the application of the second rule, EVAL-RIGHT, whose premise triggers the application of the third rule DIV, on the constants 6 and 2. Then, the third rule applies a second time to produce the result 4. Note that, in the absence of any predefined reduction strategy, one can choose to apply inference rules in any order.

3 Formalizing Value Semantics

We build our formal presentation on top of simple first-order imperative language. We start with a description of its syntax before moving on to its operational semantics.

3.1 Syntax

The syntax of our imperative language is described as follows:

$$
\begin{array}{rrrcl}
\text{Programs:} & \pi \in & \Pi & ::= & f_1; \ldots; f_n; s \\
\text{Func. decl.} & f \in & Fd & ::= & \texttt{func}\ x(x_1 : \tau_1, \ldots, x_n : \tau_n)\ \texttt{\{}\ s\ \texttt{\}} \\
\text{Types:} & \tau \in & T & ::= & \texttt{owned} \mid \texttt{inout} \\
\text{Paths:} & p \in & P & ::= & x \mid p.i \\
\text{Statements} & s \in & S & ::= & \texttt{var}\ x := s \mid p := s \mid s_1; s_n \mid s_c\ \texttt{?}\ s_t\ \texttt{:}\ s_e \\
 & & & \mid & x \mid s.i \mid s(s_1, \ldots, s_n) \mid x(s_1, \ldots, s_n)
\end{array}
$$

where the metavariables x, c and i denote a program variable identifier, a non-zero natural number, and a data constructor, respectively.

A program is a sequence of function declarations, followed by a statement representing the behavior of the program (i.e., the top-level declarations of the *main* file in a Swift program). We assume that sequential composition is left-associative, that is $s_1; s_2; s_3 = (s_1; s_2); s_3$. The identifier of a function is visible to its body, so that functions can be mutually recursive. For the sake of simplicity, we assume that programs do not contain duplicate function or variable names. It follows that our language does not support overloading. Notice that statements conflate the notions of expressions and commands. All statements compute a value. The value of a sequence of statements is defined by that of its last element. This particular design choice contributes to the conciseness of the language's operational semantics.

Constructors serve to create data aggregates, a.k.a. tuples, or records. For instance, the term $\texttt{pair}(\texttt{z}(), \texttt{s}(\texttt{z}()))$ features three constructors, `pair`, `z` and `s`. Constructors of arity 0 can be understood as atomic constants (e.g., an integer literal). Hence, in our examples, we often omit empty trailing parentheses when we use them. The set of values V is given by the set of constructors whose arguments are also values. More formally:

$$V ::= c(v_1, \ldots, v_n)$$

Note that our language does not yet feature any notion of identity or reference. It follows that all values can be mapped onto trees, whose leaves and nodes are constructors of arity 0 and $n \geq 1$, respectively. For instance, the term $\texttt{pair}(\texttt{z}, \texttt{s}(\texttt{z}))$ denotes a value that can be mapped onto the following tree. A path is a sort of "map" that describes access to a specific node in the tree. Given an aggregate, we use 1-based indices to identify a member based on its position in its superterm. In more concrete words, $c(v_1, \ldots, v_n).i = v_i$.

Finally, although function parameters are associated with type annotations, note that our language is in fact *not* statically typed. For example, the sequence of statements $\texttt{var}\ x := \texttt{a}; x := \texttt{pair}(\texttt{a}, \texttt{b})$ is legal. Type annotations only serve to specify the argument-passing semantics [1]. At this stage, our language supports two policies:

- A *pass-by-value* style, specified by the keyword **`owned`**.
- A *pass-by-reference* style, specified by the keyword **`inout`**.

3.2 Semantics

We describe our language's operational semantics by the means of an operator $\rightsquigarrow$, that transforms stateful program configurations. A configuration is a pair $\langle \sigma, s \rangle$, where $\sigma : X \to V$ is a store mapping program variables onto values and s is a statement. A program's execution *successfully* terminates when it reaches a configuration $\langle \sigma, v \rangle$, i.e., when it has reduced to a value and there is no other statements to execute. Other executions may either diverge, if the program enters an infinite computation, or get stuck at a configuration $\langle \sigma, s \rangle$ such that $s \notin V$ and no reduction rule apply, thus denoting an ill-formed program.

Because assignments introduce side effects, we must fix an evaluation order to compute deterministic evaluations. We adopt a leftmost-outermost reduction strategy to mimic Swift's evaluation order. Intuitively, this implies that everything on the left of a term must be evaluated before it can be considered. Furthermore, to keep our rules concise, we borrow the traditional notion of *evaluation context* from [2]. An evaluation context $\mathcal{C}\Box$ is a meta-term representing a family of statements where $\Box$ is a "hole". This

hole roughly corresponds to the idea of a program counter and designates the term to evaluate next, while $\mathcal{C}$ represents the superterm surrounding the hole. For instance, the evaluation context $\texttt{var}\ x := \square$ denotes a family of variable declarations in which the hole is a placeholder for the expression of the initializing value. We write $\mathcal{C}[s]$ the substitution of the hole by the term s. For instance, $\texttt{var}\ x := [a]$ denotes the evaluation context of a variable declaration whose placeholder is substituted by a. Evaluation contexts are defined as follows:

$$\begin{array}{rcl} \mathcal{C} & ::= & \texttt{var}\ x := \square \mid p := \square \mid \square; s \mid \square\ \texttt{?}\ s_t\ \texttt{:}\ s_e \\ & \mid & \square.i \mid c(v_1, \ldots v_{i-1}, \square, \ldots s_n) \\ & \mid & \langle \iota, \texttt{let}\ (x_1 = v_1, \ldots, x_{i-1} = v_{i-1}, x_i = \square, \ldots, x_n = s_n)\ \texttt{in}\ s \rangle \end{array}$$

Notice the addition of an additional construct on the last line, namely $\langle \iota, \texttt{let}\ \ldots\ \texttt{in}\ s \rangle$, which does not appear in the formal syntax defined in the previous section. This serves to handle the preparation of a function call; we elaborate on its semantics later on.

We define our evaluation operator with a judgement of the form $\pi \vdash \langle \sigma, s \rangle \rightsquigarrow \langle \sigma', s' \rangle$. Informally, the judgement lets us conclude that the configuration $\langle \sigma, s \rangle$ reduces to $\langle \sigma', s' \rangle$, in the context of the program π. Formally, it is defined by the following set of inference rules:

CONTEXT
$$\frac{\pi \vdash \langle \sigma, s \rangle \rightsquigarrow \langle \sigma', s' \rangle}{\pi \vdash \langle \sigma, \mathcal{C}[s] \rangle \rightsquigarrow \langle \sigma', \mathcal{C}[s'] \rangle}$$

NEXT
$$\frac{}{\pi \vdash \langle \sigma, v; s_2 \rangle \rightsquigarrow \langle \sigma, s_2 \rangle}$$

DECLARE
$$\frac{\sigma' = \sigma \ll [x \mapsto v]}{\pi \vdash \langle \sigma, \texttt{var}\ x := v \rangle \rightsquigarrow \langle \sigma', \texttt{void} \rangle}$$

ASSIGN
$$\frac{\sigma' = \sigma \ll [x \mapsto v]}{\pi \vdash \langle \sigma, x := v \rangle \rightsquigarrow \langle \sigma', \texttt{void} \rangle}$$

ASSIGN-MEMBER
$$\frac{\sigma(x.i_1 \ldots i_n) = v' \qquad \sigma' = \sigma \ll [x \mapsto v'[{}^{v}/_{i_1 \ldots i_n}]]}{\pi \vdash \langle \sigma, x.i_1 \ldots i_n := v \rangle \rightsquigarrow \langle \sigma', \texttt{void} \rangle}$$

VAR
$$\frac{v = \sigma(x)}{\pi \vdash \langle \sigma, x \rangle \rightsquigarrow \langle \sigma, v \rangle}$$

MEMBER
$$\frac{v = c(v_1, \ldots, v_n)}{\pi \vdash \langle \sigma, v.i \rangle \rightsquigarrow \langle \sigma, v_i \rangle}$$

IF-THEN
$$\frac{}{\pi \vdash \langle \sigma, \texttt{true}\ \texttt{?}\ s_t\ \texttt{:}\ s_e \rangle \rightsquigarrow \langle \sigma, s_t \rangle}$$

IF-ELSE
$$\frac{}{\pi \vdash \langle \sigma, \texttt{false}\ \texttt{?}\ s_t\ \texttt{:}\ s_e \rangle \rightsquigarrow \langle \sigma, s_e \rangle}$$

PREPARE-CALL
$$\frac{(\texttt{func}\ x(x_1 : \tau_1, \ldots, x_n : \tau_n)\ \{\ s\ \}) \in \pi \qquad \iota = [x_i \mapsto s_i \mid i \leq n \wedge \tau_i = \texttt{inout}] \qquad \mathit{injective}(\iota)}{\pi \vdash \langle \sigma, x(s_1, \ldots s_n) \rangle \rightsquigarrow \langle \sigma, \langle \iota, \texttt{let}\ (x_1 = s_1, \ldots, x_n = s_n)\ \texttt{in}\ s \rangle \rangle}$$

EXECUTE-CALL
$$\frac{\pi \vdash \langle \sigma', s \rangle \rightsquigarrow \langle \sigma'', s'' \rangle}{\pi \vdash \langle \sigma, \langle \iota, \sigma', s \rangle \rangle \rightsquigarrow \langle \sigma, \langle \iota, \sigma'', s'' \rangle \rangle}$$

CALL
$$\frac{\sigma' = [x_i \mapsto v_i \mid i \leq n]}{\pi \vdash \langle \sigma, \langle \iota, \texttt{let}\ (x_1 = v_1, \ldots, x_n = v_n)\ \texttt{in}\ s \rangle \rangle \rightsquigarrow \langle \sigma, \langle \iota, \sigma', s \rangle \rangle}$$

FINISH-CALL
$$\frac{\alpha = (\iota(x_i) := \sigma'(x_i) \mid x_i \in \mathit{dom}(\iota))}{\pi \vdash \langle \sigma, \langle \iota, \sigma', v \rangle \rangle \rightsquigarrow \langle \sigma, \alpha; v \rangle}$$

Let us illustrate the reduction semantics with a couple of concrete examples. The first is a program that simply initializes a pair of values, mutates its first member, and assigns its second member to another program variable. Its evaluation is carried out as below. We highlight the term being considered at each step and write the rule relating to its evaluation on the right.

$\langle [\bot],\ \texttt{var}\ x := \texttt{pair(a, b)}; x.0 := \texttt{c}\ ; \texttt{var}\ y := x.1 \rangle$	CONTEXT
$\langle [\bot],\ \texttt{var}\ x := \texttt{pair(a, b)}\ ; x.0 := \texttt{c}; \texttt{var}\ y := x.1 \rangle$	DECLARE
$\langle [x \mapsto \texttt{pair(a, b)}],\ \texttt{void}; x.0 := \texttt{c}\ ; \texttt{var}\ y := x.1 \rangle$	NEXT
$\langle [x \mapsto \texttt{pair(a, b)}],\ x.0 := \texttt{c}; \texttt{var}\ y := x.1\ \rangle$	CONTEXT
$\langle [x \mapsto \texttt{pair(a, b)}],\ x.0 := \texttt{c}\ ; \texttt{var}\ y := x.1 \rangle$	ASSIGN-MEMBER
$\langle [x \mapsto \texttt{pair(c, b)}],\ \texttt{void}; \texttt{var}\ y := x.1\ \rangle$	NEXT
$\langle [x \mapsto \texttt{pair(c, b)}],\ \texttt{var}\ y := x.1\ \rangle$	CONTEXT
$\langle [x \mapsto \texttt{pair(c, b)}], \texttt{var}\ y :=\ x.1\ \rangle$	MEMBER
$\langle [x \mapsto \texttt{pair(c, b)}],\ \texttt{var}\ y := \texttt{b}\ \rangle$	ASSIGN
$\langle [x \mapsto \texttt{pair(c, b)}, y \mapsto \texttt{b}], \texttt{void} \rangle$	

We start with an empty store and a sequence of three statements, representing the program's behavior. The first step illustrates the use of the CONTEXT rule. Notice that no other rule can match a sequence of statements unless the first one is a value. However, by substituting x's declaration for the hole of an evaluation context, we can now uset the rule CONTEXT to focus on it. At the second step, the rule DECLARE deals with the declaration and populates the store with a binding $x \mapsto \texttt{pair}(\texttt{a}, \texttt{b})$. Then, the rule NEXT applies and simply discards the unit value produced by the declaration's evaluation. The rule CONTEXT applies, once again, to focus on the assignment. The rule ASSIGN-MEMBER processes this assignment and updates the store accordingly. We use the notation $v'[{}^{v}/_{i_1 \ldots i_n}]$ to denote a copy of v' in which v is substituted for the subterm designated by the path suffix $i_1 \ldots i_n$. In this particular example, $v' = \texttt{pair}(\texttt{a}, \texttt{b})$ and the path suffix identifies the second member of the pair. Once the member assignment completed, the rule NEXT is used to move on to the last assignment. Finally, the rule ASSIGN populates the store with an additional binding $y \mapsto b$, and the program terminates.

One challenge in evaluating function calls is to handle the two different argument-passing policies. In particular, the support of a pass-by-reference style prevents a naive β-reduction as found in traditional variants λ-calculus. There are two issues to consider. One stems from the fact that mutations of **inout** arguments, as carried out by the function being called, must be propagated back to the caller. Thus, we have to save the paths identifying these arguments before they are passed. Another issue stems from the fact that **owned** arguments must be passed by value, and therefore be evaluated before the function call, preventing a naive pass-by-name approach. We solve both issues by splitting the evaluation of a function call into four steps. The first (rule PREPARE-CALL) builds a function ι that maps **inout** parameter names to expression of their corresponding argument. Note that this should be a path in a well-formed program. This mapping is used to build an intermediate term of the form $\langle \iota, \textbf{let } x_1 = s_1 \ldots x_n = s_n \textbf{ in } s \rangle$, where $x_1, \ldots, x_n$ are parameter names and s denotes the statement(s) of function being called. This term serves to drive the evaluation of each argument $s_1, \ldots, s_n$, by successive applications of the CONTEXT rule. The second step (rule CALL) consists of building a new store σ' mapping each parameter name to its corresponding value. This store is used to produce a term of the form $\langle \iota, \sigma', s \rangle$. The third step (rule EXECUTE-CALL) evaluates this term, through successive applications of the CONTEXT rule. The fourth step (rule FINISH-CALL) finally reuses the function ι created at the beginning of the process to build a sequence of assignments for each **inout** parameter to their respective value in σ', represented by the seqence α.

We illustrate the complete process with a second example. This time, we assume that the identifier f refers to the following function declaration:

$$\textbf{func } f(x : \textbf{inout}, y : \textbf{owned}) \ \{\ x := y\ \}$$

We then define a program that initializes a pair of values and calls the function f to mutate its first member. For conciseness, we omit the intermediate applications of the CONTEXT rule.

$\langle [\bot],\ \textbf{var } x := \texttt{pair}(\texttt{a}, \texttt{b})\ ; f(x.0, \texttt{c}) \rangle$	DECLARE
$\langle [x \mapsto \texttt{pair}(\texttt{a}, \texttt{b})],\ \texttt{void}; f(x.0, \texttt{c})\ \rangle$	NEXT
$\langle [x \mapsto \texttt{pair}(\texttt{a}, \texttt{b})],\ f(x.0, \texttt{c})\ \rangle$	PREPARE-CALL
$\langle [x \mapsto \texttt{pair}(\texttt{a}, \texttt{b})],\ \langle [x \mapsto x.0], \textbf{let } (x = x.0, y = \texttt{c}) \textbf{ in } x := y \rangle\ \rangle$	CALL
$\langle [x \mapsto \texttt{pair}(\texttt{a}, \texttt{b})],\ \langle [x \mapsto x.0], [x \mapsto \texttt{a}, y \mapsto \texttt{c}], x := y \rangle\ \rangle$	EXECUTE-CALL
$\langle [x \mapsto \texttt{pair}(\texttt{a}, \texttt{b})], \langle [x \mapsto x.0], [x \mapsto \texttt{a}, y \mapsto \texttt{c}],\ x := y\ \rangle \rangle$	ASSIGN
$\langle [x \mapsto \texttt{pair}(\texttt{a}, \texttt{b})],\ \langle [x \mapsto x.0], [x \mapsto \texttt{c}, y \mapsto \texttt{c}], \texttt{void} \rangle\ \rangle$	FINISH-CALL
$\langle [x \mapsto \texttt{pair}(\texttt{a}, \texttt{b})],\ x.0 := \texttt{c}\ ; \texttt{void} \rangle$	ASSIGN-MEMBER
$\langle [x \mapsto \texttt{pair}(\texttt{c}, \texttt{b})],\ \texttt{void}; \texttt{void}\ \rangle$	NEXT
$\langle [x \mapsto \texttt{pair}(\texttt{c}, \texttt{b})], \texttt{void} \rangle$	

The evaluation of the function call starts with the third reduction step. The rule PREPARE-CALL applies and builds a function $\iota = [x \mapsto x.0]$, mapping f's formal parameter to the path $x.0$. The predicate *injective* captures the idea that ι should not map two different parameter names two paths that share a common prefix. It is formally defined as follows:

$$\mathit{injective}(\iota) \Leftrightarrow \forall x, x' \in \mathit{dom}(\iota), x \neq x' \implies \mathit{base}(\iota(x)) = \mathit{base}(\iota(x'))$$

where *base* is a function that accepts a path and returns the identifier at the base of this path (e.g., $base(x.1.2) = x$). The reader may remark that this predicate implements the runtime check performed by Swift's runtime to guarantee exclusive access of **inout** parameters. The rule CALL then builds a new store $[x \mapsto \mathtt{a}, y \mapsto \mathtt{c}]$ after all arguments have been evaluated. This will serve as the binding context in which the body of the function will be evaluated. Notice that such evaluation becomes agnostic of the argument passing policy. In this example more specifically, there is no difference in the way the parameters x and y appear in the store. Hence, the rule ASSIGN can apply just as in the previous example and update x's binding. The rule FINISH-CALL handles the end of the function's evaluation. It inserts a single assignment before the returned value to propagate the mutation of the parameter x. This mutation is carried out with the next step, by the rule ASSIGN-MEMBER.

4 Goals and Challenges

One prime objective of this work is to provide Swift's community with a more precise form of documentation than currently available, addressing the lack of a formal specification for value semantics. Although a formal semantics may be perceived as an unorthodox approach to document a mainstream programming language, this will establish a single source of truth for academics and practitioners alike. Recent efforts in this direction have been met with success in the context of WebAssembly [3]. Documentation is described both in English prose and with accompanying inference rules, similar to those presented in this paper [4]. This work has already been key in the definition of mechanized proofs of soundness properties [5].

Despite this promising observation, it remains an open question to determine how a formal specification of Swift should be presented. Swift is a much larger language than WebAssembly, by many measurements. It features a broader set of control structures, a far richer type system [6] and more elaborate memory management mechanisms. This is likely to result in a fairly complex formal specification, thus motivating the need for more research is required to identify appropriate and meaningful abstractions. Such an endeavor should ideally be carried out in concert with Swift's community.

One possible way to palliate complexity could be to define different *views* of the semantics, each focusing on a specific aspect while abstracting over the others. For instance, custom initializers have been discarded from the formal system proposed in this paper, eliding any consideration for the way Swift handles definite initialization. These considerations could either be the object of an extension of the present work, or as an alternative view that would focus solely on aspects necessary to describe safe initialization schemes. Views could also be interpreted as modules of a larger system defined in terms of composition. This idea is reminiscent of the concept of *modeling frames* as found in cyber-physical systems' modeling [7].

Another avenue for future work is to define a type system that could statically guarantee exclusive access of **inout** parameters. Such a type system could be exploited to improve on the current implementation of Swift's compiler and broaden the spectrum of ill-formed situations its static analysis is able to detect. The vast literature on substructural type systems [8], capabilities and effect systems [9], permissions [10], ownership types [11] and region-based memory management [12] has probably a lot of insights to offer in this regard. Of particular interest are type capabilities [13], whose core idea is to give up on the usual type invariance of memory locations in order to encode flow-sensitive properties and/or constraints. With this framework, exclusive access would be naturally expressed in the form of a linear (i.e., non-copyable) capability that flows in and out of a function. This approach has already been studied in the context of Mezzo [14], a programming language derived from OCaml.

5 Related Work

The notion of assignment has been studied in various extensions of the λ-calculus. One early representative is λ_{var} [15]. The language dissociates regular λ-terms (i.e., immutable variables, function abstractions and applications) from other constructs aimed at modeling assignment. Mutable variables are introduced into an expression by an additional construct **let** x **in** e, akin to a regular abstraction. Similar to the present work, reading or mutating a mutable variable is expressed by the means of so-called *primitive state transformers*, which interact with a store. However, the calculus handles arguments with a call-by-name strategy, therefore eliding the issues related to the distinction between **owned** and **inout** parameters. A subsequent line of research focused on equipping λ_{var} with type systems (e.g., [16, 17]) to guarantee freedom from side effects in pure applicative terms.

Another early work proposes to model memory explicitly, using mutable variables as references to memory locations [18]. This approach has been later revisited to formalize numerous imperative programming languages, notably including ClassicJava [19], a subset of the Java language. More recently, the author's thesis [20] revisited the notion of assignment, with a particular focus on its relationship with aliasing. This work has resulted in a formal framework that is able to express low-level memory assignment semantics with different assignment operators. The so-called *copy operator* of the assignment-calculus roughly models Swift's value semantics. However, contrary to the present work, the framework relies on an ad-hoc data structure that mimics actual memory. An alternative approach [21] proposes to abstract memory by encoding the variables' bindings at the term level. While this model supports a more direct reasoning, it presupposes object-orientation and reference semantics, thus lacking support to mutate simple values (i.e., terms of arity 0). This paper opts for a compromise, avoiding the simulation of an actual memory structure while still relying on an auxiliary store to ease the distinction between **`owned`** and **`inout`** parameters.

A great body of work has been dedicated to the design of various aliasing control mechanisms, some of which are able to emulate value semantics on top of reference semantics, to some extent. Most related to the present paper are substructural type systems, in particular linear [22] and affine types [23]. Those are used to guarantee the heap remains organized like a tree, thus preventing aliasing. Programs adopting this typing discipline behave similarly as programs adopting value semantics, even if variables denote in fact references. Many systems additionally feature a notion of *reference borrowing* [24] to support some restricted, temporary form of aliasing, and effectively emulating **`inout`** parameters. One successful, concrete application of this principle is found in Cyclone [25], and later refined in Rust[2]. Other manifestations in mainstream programming languages can be found in different extensions of Scala (e.g. [26, 27]).

6 Conclusion

I presented a formal specification of Swift's value semantics, in the form of a core first-order imperative language. The language supports pass-by-value and pass-by-reference styles, without modeling memory explicitly. One prime objective of this work is to shed light on the properties of value types in a clear, complete, and unambiguous way.

Ongoing work includes the extension of the language to support reference semantics and concurrency, and the definition of a type system to statically guarantee observable uniqueness of value type instances.

References

[1] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[2] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce F. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–237, 1987.

[3] Andreas Rossberg. Who is afraid of the turnstile? 12th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, 2019.

[4] Webassembly specification: Release 1.1. `https://webassembly.github.io/spec/core/`, 2020.

[5] Conrad Watt. Mechanising and verifying the webassembly specification. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 53–65. ACM, 2018.

[6] Dimitri Racordon and Didier Buchs. Featherweight swift: A core calculus for swift's type system. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, page 140–154, New York, NY, USA, 2020. ACM.

[7] Stefan Klikovits, Joachim Denil, Alexandre Muzy, and Rick Salay. Modeling frames. In *Proceedings of MODELS CEUR, September, 17, 2017*, volume 2019 of *CEUR Workshop Proceedings*, pages 315–320. CEUR-WS.org, 2017.

[2] `https://www.rust-lang.org`

[8] Amal J. Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 78–91. ACM, 2005.

[9] Colin S. Gordon. Designing with static capabilities and effects: Use, mention, and invariants (pearl). In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 10:1–10:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[10] Ayesha Sadiq, Yuan-Fang Li, and Sea Ling. A survey on the use of access permission-based specifications for program verification. *Journal of Systems and Software*, 159, 2020.

[11] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 15–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[12] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *High. Order Symb. Comput.*, 17(3):245–265, 2004.

[13] Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In Gert Smolka, editor, *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*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381. Springer, 2000.

[14] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. The design and formalization of mezzo, a permission-based programming language. *ACM Transactions on Programming Languages and Systems*, 38(4):14:1–14:94, 2016.

[15] Martin Odersky, Dan Rabin, and Paul Hudak. Call by name, assignment, and the lambda calculus. In Mary S. Van Deusen and Bernard Lang, editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 43–56. ACM Press, 1993.

[16] Kung Chen and Martin Odersky. A type system for a lambda calculus with assignments. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings*, volume 789 of *Lecture Notes in Computer Science*, pages 347–364. Springer, 1994.

[17] Hongseok Yang and Uday Reddy. Imperative lambda calculus revisited. Technical report, University of Illinois at Urbana-Champaign, Aug 1997.

[18] Matthias Felleisen and Daniel P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69(3):243–287, 1989.

[19] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 171–183, 1998.

[20] Dimitri Racordon. *Revisiting Memory Assignment Semantics in Imperative Programming Languages.* PhD thesis, University of Geneva, Geneva, Switzerland, 2019.

[21] Andrea Capriccioli, Marco Servetto, and Elena Zucca. An imperative pure calculus. *Electronic Notes in Theoretical Computer Science*, 322:87–102, 2016.

[22] Philip Wadler. Linear types can change the world! In *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, page 561, 1990.

[23] Jesse A. Tov and Riccardo Pucella. Practical affine types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 447–458, 2011.

[24] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 557–570, 2012.

[25] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, pages 275–288, 2002.

[26] Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, pages 354–378, 2010.

[27] Philipp Haller and Alexander Loiko. Lacasa: lightweight affinity and object capabilities in scala. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 272–291, 2016.