Featherweight Swift: A Core Calculus for Swift's Type System

Racordon, Dimitri; Buchs, Didier

How to cite

RACORDON, Dimitri, BUCHS, Didier. Featherweight Swift: A Core Calculus for Swift's Type System. In: Proceedings of the 13th ACM SIGPLAN International Conference on Software Language En- gineering (SLE '20). [s.l.] : [s.n.], 2020. doi: 10.1145/3426425.3426939

This publication URL: https://archive-ouverte.unige.ch/unige:144345
Publication DOI: 10.1145/3426425.3426939

Featherweight Swift: A Core Calculus for Swift's Type System

Dimitri Racordon
University of Geneva
Department of Computer Science
Switzerland
dimitri.racordon@unige.ch

Didier Buchs
University of Geneva
Department of Computer Science
Switzerland
didier.buchs@unige.ch

Abstract

Swift is a modern general-purpose programming language, designed to be a replacement for C-based languages. Although primarily directed at development of applications for Apple's operating systems, Swift's adoption has been growing steadily in other domains, ranging from server-side services to machine learning. This success can be partly attributed to a rich type system that enables the design of safe, fast, and expressive programming interfaces. Unfortunately, this richness comes at the cost of complexity, setting a high entry barrier to exploit Swift's full potential. Furthermore, existing documentation typically only relies on examples, leaving new users with little help to build a deeper understanding of the underlying rules and mechanisms.

This paper aims to tackle this issue by laying out the foundations for a formal framework to reason about Swift's type system. We introduce Featherweight Swift, a minimal language stripped of all features not essential to describe its typing rules. Featherweight Swift features classes and protocol inheritance, supports retroactive modeling, and emulates Swift's overriding mechanisms. Yet its formalization fits on a few pages. We present Featherweight Swift's syntax and semantics. We then elaborate on the usability of our framework to reason about Swift's features, future extensions, and implementation by discussing a bug in Swift's compiler, discovered throughout the design of our calculus.

CCS Concepts: • **Software and its engineering** → **Semantics**; • **Theory of computation** → *Operational semantics.*

Keywords: protocol oriented programming, language semantics, language calculus, type systems, swift

SLE '20, November 16–17, 2020, Virtual, USA

ACM ISBN 978-1-4503-8176-5/20/11…$15.00
https://doi.org/10.1145/3426425.3426939

ACM Reference Format:
Dimitri Racordon and Didier Buchs. 2020. Featherweight Swift: A Core Calculus for Swift's Type System. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE '20), November 16–17, 2020, Virtual, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3426425.3426939

1 Introduction

Swift is a modern general-purpose programming language, developed by Apple to be a successor to C-based languages. Often introduced in the context of mobile and desktop applications, Swift's adoption in other domains has been growing steadily since its open-source release in 2014, notably in the context of server-side and machine learning development. This success can be explained on one front by the language's efficient memory model and seamless integration into the C ecosystem, and on another by its powerful type system. Swift enables the design of safe, fast and expressive programming interfaces by leveraging advanced features such as bounded polymorphism [11], existential types [31], and traits [19]. The latter in particular enables a discipline called Protocol-Oriented Programming (POP). POP is a type-driven paradigm that advocates for the use of *protocols* over concrete types. Protocols in Swift are similar to interfaces in Java, abstract classes and concepts in C++, or traits in Scala. The central idea is to reason about *type requirements* (i.e., what types should be able to do) rather than properties of a specific implementation. Sorting algorithms are a good canonical use case for such an approach. Although they are often defined over numerical values, most sorting algorithms are in fact agnostic of the type of the values being sorted, as long as it defines a total order relation. Following that observation, a protocol-oriented approach consists of first capturing this requirement in a protocol, and then writing an algorithm that would rely on some existential type satisfying this protocol, while remaining otherwise completely agnostic of any other implementation detail.

Unfortunately, Swift's type system comes at the cost of complexity. While object-orientation is generally well known and understood by most developers, knowledge on POP patterns and related concepts is not nearly as widespread. As a result, exploiting the language's full potential requires a high-level of expertise and experience. Furthermore, existing

documentation focuses on examples and does not provide a comprehensive overview of the type system's rules and mechanisms. This impedes new users, specifically students, to build a deeper understanding. The lack of a formal definition of the language's semantics also precludes one's ability to reason about its design and implementation, in particular to foresee interactions between different features.

We propose to address these issues by the means of a formal framework describing Swift's type system. We introduce Featherweight Swift (FS), a core calculus inspired by Featherweight Java (FJ) [23]. FS captures the fundamental concepts of Swift's type system and discards all features that are not essential to the description of its typing rules. It features class and protocol inheritance, supports retroactive modeling, and emulates Swift's overriding mechanism, but leaves out local and global variables, exceptions, concurrency, and even assignment. This results in a functional subset whose complete formalization fits on a few pages. While our approach elides many peculiarities, such as the difference between value and reference semantics [35], the bareness of the language exposes Swift's method lookup mechanism and type coercion rules clearly and concisely. As a result, FS helps language users and designers alike to generalize assumptions and test them against the compiler's implementation.

We present Featherweight Swift's syntax, type system, and operational semantics. Although we do not provide a full type soundness proof, we present the key properties related to the method lookup mechanism. We then illustrate the framework's usability to reason about Swift by discussing one of the bugs we found in Apple's compiler implementation, discovered throughout FS's formalization. The bug has been reported and fixed in Swift 5.2.

2 Swift's Type System in a Nutshell

This section briefly introduces Swift's type system, with a particular focus on its support for protocols. It is worth noting that POP solves similar problems as Object-Oriented Programming (OOP) [2]. However, it aims to avoid common pitfalls of class inheritance, such as the fragile base class problem [38], by advocating for composition over inheritance to keep class hierarchies as flat as possible. It also differs from mixin-based inheritance [9], a technique that consists of composing classes through multiple inheritance, specifically because it does not rely on inheritance as a composition mechanism. Nonetheless, POP does not intend to replace object-orientation. Rather, it introduces features that can be used to refine essential object-oriented aspects, such as encapsulation, polymorphism, and separation of concerns.

In Swift, a protocol represents a collection of requirements to satisfy. These requirements are specified in the form of methods to implement or properties (a.k.a. fields in Java) to expose. For instance, one could define a protocol that expresses the requirements of serializable types:

```
protocol Serializable {
  func serialize() -> String
}
```

The above protocol describes a type with a single method `serialize`. The method takes no argument and is expected to return a serialized representation of its receiver (i.e., the object on which it is called), as a character string.

There are two ways to indicate that a type conforms to a particular protocol. The first and most straightforward is to specify the conformance directly within the type's declaration and to provide an implementation for all of the protocol's requirements:

```
class Customer: Serializable {
  let id: String
  let name: String
  init(id: String, name: String) {
    self.id = id; self.name = name
  }
  func serialize() -> String {
    "(id:\(self.id), name:\(self.name))"
  }
}
```

Here, the class `Customer` is declared conforming to the protocol `Serializable`. The conformance is valid because the class has a method that satisfies the protocol's requirement.

An alternative approach, called *retroactive modeling* [41], consists of extending an existing *concrete* type to specify additional conformances and provide it with the associated implementations. For example, one can specify that Swift's integers conform to the `Serializable` protocol:

```
extension Int: Serializable {
  func serialize() -> String { "i(\(self))" }
}
```

To preserve subtyping, retroactive modeling can only *add* constructs to a type and may not modify nor remove any of its pre-existing characteristics. Simply put, an extension cannot alter any method or property from a type, nor can it subtract any protocol from its conformance set.

A protocol can refine (i.e., inherit from) one or several other protocols to represent an aggregate of requirements. For instance, we can compose a protocol `Doc`, describing the requirements for types representing a document of some sort, with `Serializable` to represent persistent documents:

```
protocol Doc {
  func title() -> String
}
protocol PersistentDoc: Serializable, Doc {
  func save(filename: String)
}
```

Protocols may also appear in signatures to express the constraints of some concrete existential type [31] satisfying their

requirements. Anonymous compositions can be created locally when composition through inheritance does not result in a reusable abstraction:

```
let doc: Serializable & Doc
```

Unlike Java interfaces, protocols may refer to the concrete type that conforms to them. For instance, consider a protocol `Orderable` describing types with a total order:

```
protocol Orderable {
  func lesser(other: Self) -> Bool
}
```

This protocol requires a single method `lesser`, that should return whether another instance is smaller than the receiver[1]. Notice the use of the type **`Self`** (a.k.a. `MyType` in related literature [10]), that serves as a placeholder for the conforming type. This allows a protocol to specify methods that are not compatible with other types conforming to the same protocol, which contributes to a stronger type safety. Indeed, the use of **`Self`** guarantees that two values of non-related types (e.g., a number and a string) cannot be compared, even if both types conform to the protocol `Orderable`. In other words, if a type `T` conforms to the protocol `Orderable`, then it should contain a method `lesser` with the type `T` → `Bool`. Consequently, the method will not accept a value of type `U`, even if `U` also conforms to `Orderable`. This solves a common pitfall referred to as the *binary method problem* [10]. However, protocols with self requirements (i.e., defining a method or property requirement annotated with **`Self`**) cannot be used to type an expression in a protocol composition. The reason for this limitation is linked to the type safety guarantee we have just discussed. Should it be possible to type a variable `x` with the protocol `Orderable`, then there would be no way to type-check a call to the method `x.lesser` statically. There are workarounds, like explicit parameterisation (e.g., Haskell type classes and C++ concepts), multiple dispatch [13] or type erasure [14], but an extensive discussion about such techniques is beyond the scope of this paper.

Swift supports protocol extensions as well. While such extensions cannot add additional conformances to a protocol, they can be used to provide additional methods and/or default implementations for a protocol's method and property requirements. The reader will remark that this feature is akin to the concept of partially implemented traits in languages such as Scala or Rust. For example, one can define a protocol for equatable types which, similarly to `Orderable`, defines method requirements to check for equality between two instances[2]. Based on this description, a default implementation of a method `notEquals` is obvious, and can be provided in a protocol extension.

[1]Note that the requirement is purely syntactical and does not prescribe anything about the method's semantics.

[2]The reader proficient in Swift will notice that this definition of `Equatable` differs from Swift's built-in protocol. This choice is deliberate and made to brush over the notion of static function requirements.

```
protocol Equatable {
  func equals(other: Self) -> Bool
}
extension Equatable {
  func notEquals(other: Self) -> Bool {
    !self.equals(other: other)
  }
}
```

Conforming types may still override default implementations, typically to provide optimized alternatives that can rely on the properties of the conforming type. Otherwise, they will inherit all default implementations. One problem related to this feature occurs when two distinct protocols have implementations for the same method requirement. This introduces an ambiguity for conforming types, which must specify which implementation should be inherited. There exists a handful of techniques to handle these cases [33]. However, in Swift, conflicts can only be resolved by overriding the default implementation in the conforming type, as of this writing. Consider the following example:

```
1  protocol Dumpable {
2    func dump()
3  }
4  extension Dumpable {
5    func dump() { print("Some Dumpable val.") }
6  }
7
8  protocol Doc {
9    func title() -> String
10 }
11 extension Doc {
12   func dump() { print(self.title()) }
13 }
14
15 class DummyDoc: Dumpable, Doc {
16   func title() -> String { "A document" }
17   func dump() { print("Some DummyDoc val.") }
18 }
19
20 DummyDoc().dump()
21 // Prints "Some DummyDoc val."
```

The class `DummyDoc` conforms to `Dumpable` and `Doc`, and obtains two default implementations for `dump`, defined at line 5 and 12 respectively. Hence, it must solve the conflict by overriding the method, which is done at line 17.

Notice that protocol extensions can provide implementations even when they are not defined as requirements. This is the case in the above example. The method `dump` is provided by an extension at line 12 even if the protocol `Doc` does not require it. This results in a subtle difference in the lookup mechanism, reminiscent of non-virtual methods in C++ [4], that allows the compiler to dispatch methods statically. Hence, if the `dump` is called on an existential type

satisfying `Doc` only, the implementation declared at line 12 will be called even if it is reimplemented in the concrete type. This behavior is specific to protocol extensions and cannot be reproduced with concrete types. It is not formalized in FS.

3 Featherweight Swift

Featherweight Swift is a strict functional subset of Swift, which focuses on the latter's essential features to describe its type system. A program is described by a set of classes, protocols and extensions, and a single expression. The latter represents the behavior of the entire program.

```
1  protocol Thing {
2    func duplicated() -> Pair
3  }
4
5  extension Thing {
6    func duplicated() -> Pair {
7      Pair(fst: self, snd: self)
8    }
9  }
10
11 class A: Object, Thing {}
12
13 class B: Object, Thing {
14   let foo: A
15 }
16
17 class C: B {
18   let bar: A
19 }
20
21 class Pair: Object {
22   let fst: protocol<Thing>
23   let snd: protocol<Thing>
24   func withFst(v: protocol<Thing>) -> Pair {
25     Pair(fst: v, snd: self.snd)
26   }
27 }
28
29 Pair(fst: A(), snd: B(foo: A()))
30   .snd
31   .duplicated()
32   .withFst(C(foo: A(), bar: A()))
33 // Evaluates to "Pair(
34 //   fst: C(foo: A(), bar: A()),
35 //   snd: B(foo: A()))"
```

Figure 1. A typical Featherweight Swift program

From Swift, FS keeps protocols, extensions and classes, and its type system supports protocol conformance, protocol composition, and class inheritance. However, among Swift's most common features, FS drops non-member functions and properties (i.e., functions and variables declared outside of a class), computed properties, exceptions, concurrency, and side effects. All properties are considered to be **`let`** (a.k.a. constant) bindings, assigned only once in their class' initializer, and method arguments cannot be reassigned. We also elide the difference between value and reference types, whose respective observable behaviors are indistinguishable in the absence of side effects. Despite all these omissions, FS remains expressive enough to be Turing complete (one can encode the λ-calculus in FS). Note that FS does not feature generic types. While these account for a significant part of Swift's type system, we prefer to focus solely on class inheritance and protocol conformance.

Figure 1 illustrates an example of a FS program. It starts with the declaration of a protocol `Thing`, which contains a single requirement for a method `duplicated`. It is extended at line 5 to provide a default implementation for its unique requirement, which consists of initializing a new instance of the class `Pair` using the method receiver for both elements. The program declares four classes, `A`, `B`, `C` and `Pair` at lines 11, 13, 17 and 21 respectively. `A` and `B` inherit from a built-in root class `Object` and conform to the protocol `Thing`, but `B` additionally declares a property `foo`, which is inherited by the class `C`. `Pair` declares two properties `fst` and `snd`, along with a method `withFst` that returns a copy of itself in which the first element is substituted with the method's argument. The use of these types is finally illustrated by the expression at line 29, which initializes a pair, duplicates its second element to produce another pair, and substitutes its first element with an instance of the class `C`.

3.1 Formal Syntax

Let C denote a syntactic category, we write $C^{\#}$ for a possibly empty set of syntactic constructions that are generated by C. Similarly, let S be a set, we write $S^{\#} \subseteq \mathcal{P}(S)$.

Definition 3.1 (Featherweight Swift's syntax)**.** Let I_p denote the set of protocol identifiers, I_c denote the set of class identifiers and I_x denote the set of variable, property, and method identifiers. Let the metasyntactic variable p range over I_p. Let c and d range over I_c, and x range over I_x. FS's syntax is described as follows:

Prog.	Π	::=	$Pd^{\#}\ Cd^{\#}\ Xd^{\#}\ e$
Prot.	Pd	::=	**`protocol`** $p : I_p^{\#}\ \{\ Rd^{\#}\ \}$
Class	Cd	::=	**`class`** $c : d, I_p^{\#}\ \{\ Vd^{\#}\ Md^{\#}\ \}$
Ext.	Xd	::=	**`extension`** $p\ \{\ Md^{\#}\ \}$
		|	**`extension`** $c : I_p^{\#}\ \{\ Md^{\#}\ \}$
Prop.	Vd	::=	**`let`** $x : \tau$
Meth.	Md	::=	**`func`** $x(x_1 : \tau_1, \ldots, x_n : \tau_n) \to \tau\ \{\ e\ \}$
Req.	Rd	::=	**`func`** $x(x_1 : \tau_1, \ldots, x_n : \tau_n) \to \tau$
Expr.	e	::=	$x \mid e.x \mid e$ **`as`** $\tau \mid e$ **`as!`** τ
		|	$e(e_1, \ldots, e_n) \mid c(x_1 : e_1, \ldots, x_n : e_n)$
Type	τ	::=	$c \mid$ **`protocol`**$\langle I_p^{\#} \rangle \mid \tau \to \tau \mid$ `Self`

FS's syntax is defined to be as close as possible to that of Swift. In fact, FS programs can almost be copy-pasted into a Swift compiler and have the same semantics. We assume that a program does not contain duplicate type declarations. Similarly, we assume that classes do not contain duplicate property or method declarations and that methods do not feature duplicate parameters.

A protocol declaration **protocol** $p : P \{ R \}$ declares a protocol p that refines each of the protocols in P. The declaration's body is a set of method requirements (i.e., methods that conforming types should implement), which are declared the just as regular methods, except that they do not have a body. For simplicity, we omit property requirements. These would require the addition of computed properties (i.e., properties whose values are the result of a method call), whose typing mechanism overlaps with regular methods.

A class declaration **class** $c : d, P \{ V\ M \}$ introduces a class named c that inherits from a base class d, and conforms to each of the protocols in P. All class declarations define a supertype, for the sake of syntactic regularity. Consequently, classes that in standard Swift do not inherit from any base class are declared inheriting from a root class `Object` in FS[3]. The body of a class declaration consists of a set of property declarations and a set of method declarations. Since the language is free of side effects, all properties can be treated as **let** (a.k.a. constant) bindings. In other words, all property declarations are of the form **let** $x : \tau$, where x is the name of the property being declared and τ is its type, and we omit default values. FS does not support overloading, including for initializers. Thus, all classes necessarily have a single memberwise initializer (i.e., an initializer that accepts an argument for each of the class' properties), which we keep implicit for conciseness. Unlike in Swift, nested classes (a.k.a. inner classes in Java) are prohibited. While such a feature can serve to declutter the global namespace, it does not contribute to expressiveness. Furthermore, we also omit custom deinitializer (a.k.a. destructors), as those cannot affect a program's behavior in the absence of side effects.

An extension declaration of the form **extension** $p \{ M \}$ is called a *protocol extension*. It provides a protocol p with method implementations for its conforming types. Similarily, an extension of the form **extension** $c : P \{ M \}$ is called a *class extension*. It can provide a class c with additional protocol conformances and method implementations.

A method declaration **func** $x(x_1 : \tau_1, \ldots, x_n : \tau_n) \to \tau \{ e \}$ declares a method x that accepts n parameters and always returns a value, computed by evaluating the expression e corresponding to the method's body. A method's body can refer to a special variable **self** $\in I_x$ that identifies the method's receiver and allows recursion. It is treated as a regular variable rather than a keyword, so that no additional typing or evaluation rule is required to describe property accesses. We omit static and class methods, as well as non-member functions (i.e., functions declared outside of a class). Methods are first-class citizen in FS. Therefore, they must be treated as standard expressions (e.g., as arguments or return values), and an application of the form $e(e_1, \ldots, e_n)$ may feature any kind of expression for e. An application whose receiver is a class identifier (i.e., $c(x_1 : e_1, \ldots, x_n : e_n)$) denotes a call to the latter's implicit initializer. In Swift, while all initializer and method parameters are positional, they must *also* be named at the call site by default, unless defined otherwise explicitly. However, as of this writing, parameter names are actually not part of a method's type. This means that a function assigned to a variable, or passed as a parameter, cannot be called with named parameters. We adopt a more consistent approach in FS. In order to avoid determining an order on property declarations and inheritance thereof, we always require parameters to be named in calls to a class' implicit memberwise initializer. On the other hand, parameters to method calls are kept positional only.

We distinguish two sorts of cast expressions: e **as** τ is called a *guaranteed cast* and e **as!** τ is called a *forced cast*. FS's typing rules ensure that a guaranteed cast cannot trigger an error at runtime, whereas such an assumption is not verified statically for forced casts. We elide Swift's safe casts (i.e., expressions of the form e **as?** τ in Swift), as they involve generic option types (a.k.a. the maybe monad [32]), which are not supported in FS.

An expression can be typed by a class, a protocol composition (i.e., a set of protocols), a function type, or the special type variable **Self**. The reader proficient in Swift will remark the use of a legacy syntax for protocol composition, which consists of a set of protocol identifiers enclosed in angle brackets and prefixed by the keyword **protocol**. While no longer supported, this syntax lets us express the empty composition **protocol<>** which corresponds to Swift's built-in `Any` type (i.e., an existential type without any requirement). A method or property may refer to the type in which it is declared, allowing the declaration of self-referencing types. In protocol declarations and extensions, the special type variable **Self** designates the concrete conforming type. In class declarations and extensions, **Self** designates any derived type that can inherit from the class:

```
1 class Parent: Object {
2   func identity() -> Self { self }
3 }
4 class Child: Parent {}
```

The type of the method `identity` depends on that of the receiver. If the method is called on an object of type `Parent`, then the result will be an instance of `Parent`. On the other hand, if it is called on an object of type `Child`, then the result will be an instance of `Child`. We elaborate further on the reasons for this subtlety later.

[3] Such a root class does not exist in actual Swift, but can be easily reproduced.

3.2 Typing Semantics

We now describe FS's typing semantics. We start by introducing some notation to help formalizing predicates on declarations more concisely. Let $\pi \in \Pi$ be a program, we write $\pi \vdash \textbf{protocol}\ p\ :\ P\ \{\ R\ \}$ to denote that the protocol p's declaration is part of the program π. Similarly, we write $\pi \vdash \textbf{class}\ c\ :\ d, P\ \{\ V\ M\ \}$ for class declarations, $\pi \vdash \textbf{extension}\ p\ \{\ M\ \}$ for protocol extensions and $\pi \vdash \textbf{extension}\ c : P\ \{\ M\ \}$ for class extensions. We use letters decorated with a tilde symbol (i.e., $\sim$) to denote declarations. Let $\widetilde{d}$ be a protocol, class, property, method or method requirement declaration, we write $\widetilde{d}.name$ for its name and $\widetilde{d}.type$ for its type. For example, let $\widetilde{m} = \textbf{func}\ m(x : \tau) \to \sigma\ \{e\}$, $\widetilde{m}.name = m$ and $\widetilde{m}.type = \tau \to \sigma$.

Definition 3.2 (Signature equivalence)**.** Let $\widetilde{m}_1$ and $\widetilde{m}_2$ be two method or method requirement declarations. We say that they are signature-equivalent, written $\widetilde{m}_1 \approx \widetilde{m}_2$, if their names and type signatures are identical. More formally:

$$\widetilde{m}_1 \approx \widetilde{m}_2 \iff \widetilde{m}_1.name = \widetilde{m}_2.name \wedge \widetilde{m}_1.type = \widetilde{m}_2.type$$

Definition 3.3 (Class inheritance)**.** Let c and d denote two classes in a program π, we write $\pi \vdash c \le d$ if c inherits from d in π. Class inheritance is the reflexive and transitive closure of the immediate subclass relation specified in class declarations. In formal terms, let c, d and e be class identifiers, $\le$ is the minimal relation such that:

$$\frac{}{\pi \vdash c \le c} \qquad \frac{\pi \vdash c \le d \qquad \pi \vdash d \le e}{\pi \vdash c \le e}$$

$$\frac{\pi \vdash \textbf{class}\ c : d, P\ \{\ V\ M\ \}}{\pi \vdash c \le d}$$

Definition 3.4 (Protocol conformance)**.** Let p and q denote two protocols in a program π, we write $\pi \vdash p \sqsubseteq q$ if p conforms to (i.e., inherits from) q in π. Similarly, let c denote a class, we write $\pi \vdash c \sqsubseteq q$ if c conforms to q in π. Protocol conformance (i.e., $\sqsubseteq$) is the reflexive and transitive closure of the immediate conformance relations declared in protocol and class declarations, as well as class extensions. In formal terms, let p, q and r be protocol identifiers, and c and d be class identifiers, $\sqsubseteq$ is the minimal relation such that:

$$\frac{}{\pi \vdash p \sqsubseteq p} \qquad \frac{\pi \vdash \textbf{protocol}\ p : Q\ \{\ R\ \} \qquad q \in Q}{\pi \vdash p \sqsubseteq q}$$

$$\frac{\pi \vdash p \sqsubseteq q \qquad \pi \vdash q \sqsubseteq r}{\pi \vdash p \sqsubseteq r} \qquad \frac{\pi \vdash c \le d \qquad \pi \vdash d \sqsubseteq p}{\pi \vdash c \sqsubseteq p}$$

$$\frac{\pi \vdash \textbf{class}\ c : d, P\ \{\ V\ M\ \} \quad q \in P}{\pi \vdash c \sqsubseteq q} \qquad \frac{\pi \vdash \textbf{extension}\ c : P\ \{\ M\ \} \quad q \in P}{\pi \vdash c \sqsubseteq q}$$

Example 3.5 (Protocol inheritance)**.** Let π be a program defined as follows:

```
1 protocol P {}
2 protocol Q: P {}
3 protocol R {}
4 class C: Object, Q {}
5 class D: C, R {}
6 D()
```

From π, we can deduce that $\pi \vdash \texttt{Q} \sqsubseteq \texttt{P}$ and $\pi \vdash \texttt{D} \sqsubseteq \texttt{P}$.

Definition 3.6 (Conformance set)**.** Let τ be a protocol or class in a program π. We write $conf(\tau, \pi)$ for the set of protocols to which τ conforms in π. More formally:

$$conf(\tau, \pi) = \{p \in I_p \mid \pi \vdash \tau \sqsubseteq p\}$$

Example 3.7 (Conformance set)**.** Let π be the program illustrated in Example 3.5. program defined as follows: From π, we can deduce that $conf(\texttt{D}, \pi) = \{\texttt{P}, \texttt{Q}, \texttt{R}\}$.

Both the subtyping and the conformance relations allow us to define polymorphic substitutability, which is referred to as type *coercion* in Swift. This is a purely syntactical application of Liskov's substitution principle [29].

Definition 3.8 (Coercion)**.** Let τ and σ be two types in a program π. We write $\pi \vdash \tau \trianglelefteq \sigma$ if τ can be coerced into σ in π. In more formal terms, let τ and σ denote two types and let c be a class identifier, coercion is defined as the minimal relation such that:

$$\frac{}{\pi \vdash \tau \trianglelefteq \tau} \qquad \frac{\forall i, \pi \vdash \sigma_i \trianglelefteq \tau_i \qquad \pi \vdash \tau_r \trianglelefteq \sigma_r}{\pi \vdash \tau_1, \ldots, \tau_n \to \tau_r \trianglelefteq \sigma_1, \ldots, \sigma_n \to \sigma_r}$$

$$\frac{\pi \vdash \tau \le \sigma}{\pi \vdash \tau \trianglelefteq \sigma} \qquad \frac{\forall p \in P, c \sqsubseteq p}{\pi \vdash c \trianglelefteq \textbf{protocol}\langle P \rangle}$$

$$\frac{\forall p \in P, \exists q \in Q, \pi \vdash p \sqsubseteq q}{\pi \vdash \textbf{protocol}\langle P \rangle \trianglelefteq \textbf{protocol}\langle Q \rangle}$$

Example 3.9 (Coercion)**.** Let π be the program illustrated in Example 3.5. program defined as follows: From π, we can deduce that the following relations hold:

$$\pi \vdash (\texttt{C}, \texttt{C} \to \texttt{D}) \trianglelefteq (\texttt{D}, \texttt{C} \to \texttt{C}) \qquad \pi \vdash \texttt{C} \trianglelefteq \textbf{protocol}\langle\rangle$$

$$\pi \vdash \textbf{protocol}\langle \texttt{Q}, \texttt{R} \rangle \trianglelefteq \textbf{protocol}\langle \texttt{P}, \texttt{R} \rangle$$

3.2.1 Name Lookup.

We define a function *props* that accepts a class and returns the set of properties available in that class. Note that *props* does not need to check extensions nor conformed protocol, as these may only define additional methods in FS.

$$\frac{}{props(\texttt{Object}, \pi) = \varnothing}$$

$$\frac{\pi \vdash \textbf{class}\ c : d, P\ \{\ V\ M\ \} \qquad props(d, \pi) = V'}{props(c, \pi) = V \cup V'}$$

We define a function $lookup_v$ that returns the set of the property declarations with the given name in the given class:

$$\widetilde{v} \in lookup_v(x, c, \pi) \iff \widetilde{v} \in props(c, \pi) \wedge \widetilde{v}.name = x$$

Example 3.10 (Property lookup)**.** Let π be a program defined as follows:

```
1 class C: Object {}
2 class D: C { let foo: C }
3 class E: D { let bar: D }
4 E(foo: C(), bar: D(foo: C()))
```

We can deduce that $props(\mathtt{E}, \pi)$ contains `let foo : C` and `let bar : D`. It follows that $lookup(\mathtt{foo}, \mathtt{D}, \pi) = \{$`let foo : C`$\}$.

Both class inheritance and protocol conformance contribute to FS's method lookup mechanism. The methods that are associated with a class can be defined in the class' declaration itself, in the declaration of another class from which it inherits, in extensions of the class, or even in extensions of the protocols to which the class conforms. While FS does not support method overloading, it allows overriding. A method declared in a class can be overridden in any of its derived classes. Furthermore, methods required in a protocol and implemented in an extension are meant to define a *default* behavior for all conforming types, and may therefore be overridden as well. Method lookup is thus performed along two dimensions. The first and most straightforward relates to the class hierarchy. A method's declaration is searched starting from the most derived class and climbing up through each superclass. Note that methods in class extensions are interpreted as being part of the class declaration and belong to the class hierarchy. The second dimension relates to protocol conformance. If a method's declaration cannot be found within the class hierarchy, it is searched within the extensions of the protocol to which the class conforms.

Figure 2 illustrates different scenarios of method inheritance and overriding. The protocols `P`, `Q` and `R` are all associated with a single default implementation for a method `foo`, defined in protocol extensions at line 5, 10 and 15, respectively. The protocol `P` additionally requires that conforming types implement a method `ham`, as indicated by the method requirement declared at line 2. Note also that the protocol `R` inherits from both the protocols `P` and `Q`. The class `A` has only one method, `bar`, defined directly in the class' declaration, at line 19. This method is inherited by the class `B` through the traditional class inheritance mechanism. In addition, the class `B` has a method `ham`, declared at line 23, which satisfies the method requirement of the protocol `P`, whose conformance is declared at line 22. By the same conformance, the class also inherits the default implementation for the method `foo`, declared at line 5. The class `C1` has three methods `foo`, `bar` and `ham`. However, the two former are overridden in the class' declaration, at line 27 and 28, respectively[4]. Note

[4]Swift demands that overridden methods be explicitly prefixed with the keyword **`override`**. We omit this requirement for the sake of conciseness.

```
1  protocol P {
2    func ham() -> B
3  }
4  extension P {
5    func foo() -> A { A() }
6  }
7
8  protocol Q {}
9  extension Q {
10   func foo() -> A { B() }
11 }
12
13 protocol R: P, Q {}
14 extension R {
15   func foo() -> A { C1() }
16 }
17
18 class A: Object {
19   func bar() -> A { A() }
20 }
21
22 class B: A, P {
23   func ham() -> B { B() }
24 }
25
26 class C1: B, Q {
27   func foo() -> A { C1() }
28   func bar() -> A { C1() }
29 }
30
31 class C2: B, Q, R {}
32 extension C2 {
33   func qux() -> A { C2() }
34 }
```

Figure 2. Examples of method overriding

that the method `foo` *must* be overridden. The choice of its implementation would be ambiguous otherwise, as class `C1` inherits the default implementations of both the protocols `P` and `Q`. This is not the case for the class `C2`. The class conforms to the protocol `R` and thus inherits the latter's default implementation, declared at line 15. The latter overrides those associated with the protocols `P` and `Q`, since `R` refines both of them. The class `C2` also declares the method `qux`, but in an extension at line 33, rather than directly in its declaration.

Let $M, N \in \mathcal{P}(Md) \cup \mathcal{P}(Rd)$ be two sets of method and method requirement declarations. We write $N \gg M$ the union of M with N subtracted by the declarations in M that have a signature-equivalent declaration in N. More formally:

$$N \gg M = \{\widetilde{m} \in M \mid \nexists \widetilde{n} \in N, \widetilde{m} \approx \widetilde{n}\} \cup N$$

We define a function *xmeths* that accepts a protocol or class and gathers all methods declarations from its extensions. For instance, let π be the program shown in Figure 2, then $xmeths(\texttt{R}, \pi) = \{\textbf{func}\ \texttt{foo}() \to \texttt{A}\ \{\ \texttt{C1}()\ \}\}$. More formally, let τ denote a protocol or a class in a program π, $xmeths(\tau, \pi)$ is the minimal set such that:

$$\frac{\pi \vdash \textbf{extension}\ \tau\ \{\ M\ \}}{M \subseteq xmeths(\tau, \pi)} \qquad \frac{\pi \vdash \textbf{extension}\ \tau : P\ \{\ M\ \}}{M \subseteq xmeths(\tau, \pi)}$$

We then define a function *pmeths* that accepts a protocol and gathers all method requirements and default implementations that are either associated directly with this protocol, or inherited from another protocol higher in its hierarchy. For instance, let π be the program shown in Figure 2, then $pmeths(\texttt{R}, \pi) = \{\textbf{func}\ \texttt{foo}() \to \texttt{A}\ \{\ \texttt{C1}()\ \}, \textbf{func}\ \texttt{ham}() \to \texttt{B}\}$. More formally, let p denote a protocol in a program π, $pmeths(p, \pi)$ is defined as the set such that:

$$\frac{\begin{array}{cc}\pi \vdash \textbf{protocol}\ p : Q\ \{\ R\ \} & M = xmeths(p, \pi) \\ CS = conf(p, \pi) - \{p\} & M_{CS} = \bigcup_{q \in CS} pmeths(q, \pi)\end{array}}{pmeths(p, \pi) = (M \gg R) \gg M_{CS}}$$

Note that *pmeths* does not apply overriding between protocols that do not have a conformance relationship. In other words, if two protocols q and r belong to the conformance set of some protocol p in a program π such that neither $\pi \vdash q \sqsubseteq r$ nor $\pi \vdash r \sqsubseteq q$, then all requirements and default implementations of q and r are inherited by p, even if they have the same signature. This is why `foo` must be overridden by the class `C1` in the program shown in Figure 2. We show later how FS's type system checks that ambiguous method references are rejected in a well-typed program.

Similarly to *pmeths*, we define a function *cmeths* that accepts a class and gathers all methods and method requirements that are declared directly in the class' declaration, or its extensions, or inherited from a superclass, or inherited as a default implementation from a conformed protocol. We decompose the lookup process into three steps. First, let c be a class in a program π, we write $cmeths_0(c, \pi)$ for the set of methods declared directly within its declaration and extensions. For instance, let π be the program shown in Figure 2, $cmeths_0(\texttt{B}, \pi)$ is a singleton containing the method `ham`'s declaration. More formally, $cmeths_0$ is defined as follows:

$$\frac{\pi \vdash \textbf{class}\ c : d, P\ \{\ V\ M\ \}}{cmeths_0(c, \pi) = M \cup xmeths(c, \pi)}$$

Next, we write $cmeths_1(c, \pi)$ for the set that also includes the declarations inherited from superclasses. For instance, $cmeths_1(\texttt{B}, \pi)$ contains the declarations for `ham` and `bar`, inherited from the class `A` in the program shown in Figure 2.

$$\frac{\pi \vdash \textbf{class}\ c : d, P\ \{\ \ldots\ \}}{cmeths_1(c, \pi) = cmeths_0(c, \pi) \gg cmeths_1(d, \pi)}$$

Finally, we write $cmeths(c, \pi)$ for the set that also gathers the implementations associated with conformed protocols:

$$\frac{\begin{array}{c}CS = conf(c, \pi) \\ M_X = \bigcup_{q \in CS} xmeths(q, \pi) \qquad M = cmeths_1(c, \pi) \gg M_X \\ M' = \{\widetilde{m}[\texttt{Self} \mapsto c] \mid \widetilde{m} \in M\}\end{array}}{cmeths(c, \pi) = M'}$$

Recall that methods defined in the class hierarchy take precedence over method requirements and default implementations, even if the conformance is not defined on a superclass. More formally, if a declaration $\widetilde{m}_c$ appears in a class c or any superclass d such that $\pi \vdash c \leq d$, and a protocol $q \in conf(c, \pi)$ defines a requirement or default implementation $\widetilde{m}_q$, then $\widetilde{m}_c \approx \widetilde{m}_q \implies \widetilde{m}_q \notin cmeths(c, \pi)$. Moreover, notice that *cmeths* substitutes references to `Self` by the class for which it builds the set of methods. For instance, if a class c conforms to a protocol with a default implementation $\textbf{func}\ foo(x : \texttt{Self}) \to \texttt{Self}\ \{\ x\ \}$, then it will be transformed as $\textbf{func}\ foo(x : c) \to c\ \{\ x\ \}$ in $cmeths(c, \pi)$. We define a function $lookup_{cm}$ that returns the set of methods and method requirements with the given name in the given class:

$$lookup_{cm}(x, c, \pi) = \{\widetilde{m} \in cmeths(c, \pi) \mid \widetilde{m}.name = x\}$$

Similarly, we define a function $lookup_{pm}$ that returns the set of method requirement and default implementations with the given name in the given protocol:

$$lookup_{pm}(x, p, \pi) = \{\widetilde{m} \in pmeths(p, \pi) \mid \widetilde{m}.name = x\}$$

Example 3.11 (Method lookup)**.** Let π be the program in Figure 2. The set $cmeths_0(\texttt{C2}, \pi)$ of methods declared within `C2`'s declaration contains only $\textbf{func}\ \texttt{qux}() \to \texttt{A}\ \{\ \texttt{C2}()\ \}$. The set $cmeths_1(\texttt{C2}, \pi)$ that also includes the methods declared in `C2`'s class hierarchy additionally contains $\textbf{func}\ \texttt{ham}() \to \texttt{B}\ \{\ \texttt{B}()\ \}$ and $\textbf{func}\ \texttt{bar}() \to \texttt{A}\ \{\ \texttt{A}()\ \}$. Finally, the set containing the four methods available to the class `C2` is given by $cmeths(\texttt{C2}, \pi) = \{\textbf{func}\ \texttt{ham}() \to \texttt{B}\ \{\ \texttt{B}()\ \}, \textbf{func}\ \texttt{bar}() \to \texttt{A}\ \{\ \texttt{C2}()\ \}, \textbf{func}\ \texttt{foo}() \to \texttt{A}\ \{\ \texttt{C1}()\ \}, \textbf{func}\ \texttt{qux}() \to \texttt{A}\ \{\ \texttt{C2}()\ \}\}$

3.2.2 Type Checking.

The typing rules for expressions are presented in Figure 3. All rules are described in the form of a typing judgment $\pi, \Gamma \vdash e : \tau$, where π is a program, Γ is a mapping from variable identifiers to types, e is an expression and τ is the type of the expression. We draw the reader's attention on a few subtleties. The rules TE-CProp and TE-CMeth correspond to the typing of a class member selection (i.e., a property or a method). Recall that $lookup_v$ and $lookup_{cm}$ returns sets of declarations. Consequently, the rules check that there is at least one candidate. However, it does not ensure that this candidate is unique, which could lead to ambiguous situations. Fortunately, this cannot happen in a well-typed program because the typing rule for class declaration (described later) ensures that class members are not overloaded. The same assumption does not hold for protocol compositions (i.e., a protocol composition *can*

$$\text{TE-Var}\ \frac{\tau = \Gamma(x)}{\pi, \Gamma \vdash x : \tau} \qquad \text{TE-GCast}\ \frac{\pi, \Gamma \vdash e : \sigma \quad \pi \vdash \sigma \trianglelefteq \tau \quad inst(\tau, \pi)}{\pi, \Gamma \vdash (e\ \textbf{as}\ \tau) : \tau} \qquad \text{TE-FCast}\ \frac{\pi, \Gamma \vdash e : \sigma \quad \pi \vdash \tau \trianglelefteq \sigma \quad inst(\tau, \pi)}{\pi, \Gamma \vdash (e\ \textbf{as!}\ \tau) : \tau}$$

$$\text{TE-CProp}\ \frac{\pi, \Gamma \vdash e : c \quad \widetilde{v} \in lookup_v(x, c, \pi)}{\pi, \Gamma \vdash e.x : \widetilde{v}.type} \qquad \text{TE-CMeth}\ \frac{\pi, \Gamma \vdash e : c \quad \widetilde{m} \in lookup_{cm}(x, c, \pi)}{\pi, \Gamma \vdash e.x : \widetilde{m}.type}$$

$$\text{TE-PMeth}\ \frac{\pi, \Gamma \vdash e : \textbf{protocol}\langle P \rangle \quad \{\widetilde{m}\} = \bigcup_{p \in P} lookup_{pm}(x, p, \pi)}{\pi, \Gamma \vdash e.x : \widetilde{m}.type} \qquad \text{TE-Call}\ \frac{\pi, \Gamma \vdash e : (\tau_1, \ldots, \tau_n) \to \tau \quad \forall i, \pi, \Gamma \vdash e_i : \sigma_i \quad \forall i, \pi \vdash \sigma_i \trianglelefteq \tau_i}{\pi, \Gamma \vdash e(e_1, \ldots, e_n) : \tau}$$

$$\text{TE-Init}\ \frac{props(c, \pi) = \{\widetilde{v}_1, \ldots, \widetilde{v}_n\} \quad \forall i, \pi, \Gamma \vdash e_i : \sigma_i \quad \forall i, \widetilde{v}_i.name = x_i \wedge \pi \vdash \sigma_i \trianglelefteq \widetilde{v}_i.type}{\pi, \Gamma \vdash c(x_1 : e_1, \ldots, x_n : e_n) : c}$$

Figure 3. Featherweight Swift's expression typing

be associated with overloaded symbols). As a result, the rule T-PMeth must also check for the candidate's uniqueness.

A cast is statically "guaranteed" (TE-GCast) if the type of the expression to cast can be coerced into the specified type. On the other hand, a cast whose specified type is a subtype of the expression's type cannot be type-checked statically, and is therefore dubbed "forced" (TE-FCast). Note that casting between completely unrelated types (i.e., a pair τ, σ such that neither $\pi \vdash \tau \leq \sigma$ nor $\pi \vdash \sigma \leq \tau$) necessarily results in a type error. Recall that protocols with self requirements cannot be used to type an expression, due to the binary method problem [10]. Hence, casting rules must also ensure whether the specified type is instantiable, which is performed by checking a predicate *inst*, formally defined as follows:

$$\frac{\pi \vdash \textbf{protocol}\ p, Q\ \{\ R\ \} \quad \forall q \in Q, inst(q, \pi) \quad \forall \widetilde{r} \in R, inst(\widetilde{r}.type, \pi)}{inst(p, \pi)} \qquad \frac{\pi \vdash \textbf{class}\ c : d, P\ \{\ \ldots\ \} \quad inst(d, \pi)}{inst(c, \pi)}$$

$$\frac{}{inst(\texttt{Object}, \pi)} \qquad \frac{\forall q \in P, inst(q, \pi)}{inst(\textbf{protocol}\langle P \rangle, \pi)}$$

$$\frac{\forall i \in \{0, \ldots, n\}, \tau_i \neq \texttt{Self} \wedge inst(\tau_i, \pi)}{inst(\tau_1, \ldots, \tau_n \to \tau_0, \pi)}$$

The type-checking rules are presented in Figure 4. Typing a program (TD-Program) consists of type-checking all its declarations as well as the expression that represents its behavior. Type-checking for class declarations guarantees that all the requirements from the protocols to which the class conforms are actually fulfilled. Notice that protocol conformance is not a structural property. In other words, even if a class c has an implementation for all the method requirements in p, $c \models p$ does not hold unless p is explicitly defined as a protocol to which c must conform (i.e., $p \in conf(c)$). We write $impls(c) \subseteq cmeths(c)$ the set of complete method declarations associated with c, that is the set of declarations that have a body, and define conformance checking as follows:

Definition 3.12 (Protocol conformance checking)**.** Let c be a class and p a protocol in a program π such that $\pi \vdash c \sqsubseteq p$. We say that c satisfies its conformance to p, written $\pi \vdash c \models p$, if it has a single implementation for each of the method requirements defined by p and the protocols from which p inherits. More formally, $\sqsubseteq$ is the minimal relation such that:

$$\frac{\pi \vdash c \sqsubseteq p \quad \pi \vdash \textbf{protocol}\ p : Q\ \{\ R\ \} \quad \forall \widetilde{r} \in R, \exists! \widetilde{m} \in impls(c), \widetilde{m} \approx \widetilde{r} \quad \forall q \in conf(p, \pi) - \{p\}, \pi \vdash c \models q}{\pi \vdash c \models p}$$

Type-checking for method declarations is described by TD-PMeth and TD-CMeth, which determine whether a declaration is valid in the context of the protocol or class, respectively. Both rules verify that the type of the method's body is compatible with the method's signature. This is done in a fresh typing environment, in which only **`self`** and the method's parameters are defined, to prevent methods from capturing any identifiers by closure. Recall that overriding does not apply to protocols' requirements and default implementations. Nonetheless, FS's type system requires that all methods with the same name have the same signature, which is checked in rule TD-PMeth. The rule TD-Protocol checks that the conformance set of a protocol p forms a directed acyclic graph rooted by p, to avoid circular inheritance. It also prevents protocol extensions to declare a method implementation without a matching requirement, thus disallowing statically dispatched methods. Class declarations additionally require that properties and methods cannot be overloaded. There should be a single implementation of each class member, which is checked in rules TD-CProp and TD-CMeth. This is not a requirement in TD-PMeth, as a protocol may receive multiple default implementations from the protocols that it refines. Consequently, a class that inherits multiple default implementations (typically obtained via different protocol extensions) must override them, so that method calls are not ambiguous. For instance, the declaration of the class `C1` in Figure 2 would no longer be well-typed if line 27 were commented out.

TD-Program
$$\frac{\forall \widetilde{p} \in P, (P, C, X, e) \vdash \widetilde{p} : \widetilde{p}.name \quad \forall \widetilde{c} \in C, (P, C, X, e) \vdash \widetilde{c} : \widetilde{c}.name \quad (P, C, X, e), \varnothing \vdash e : \tau}{\vdash (P, C, X, e) : \tau}$$

TD-Protocol
$$\frac{\nexists q \in conf(p, \pi), q \sqsubseteq p \quad \forall \widetilde{m} \in xmeths(p, \pi), \exists \widetilde{r} \in R, \widetilde{m} \approx \widetilde{r} \quad \forall \widetilde{m} \in pmeths(p, \pi), (\pi, p) \vdash \widetilde{m} : \widetilde{m}.type}{\pi \vdash \textbf{protocol}\ p, P\ \{\ R\ \} : p}$$

TD-Class
$$\frac{\forall p \in conf(c, \pi), \pi \vdash c \models p \quad \forall \widetilde{v} \in props(c, \pi), (\pi, c) \vdash \widetilde{v} : \widetilde{v}.type \quad \forall \widetilde{m} \in cmeths(c, \pi), (\pi, c) \vdash \widetilde{m} : \widetilde{m}.type}{\pi \vdash \textbf{class}\ c : d, P\ \{\ V\ M\ \} : c}$$

TD-PReq
$$\frac{\forall \widetilde{m} \in lookup_{pm}(x, p, \pi), \widetilde{m}.type = \tau_1, \ldots, \tau_n \to \tau}{\pi, p \vdash \textbf{func}\ x(x_1 : \tau_1, \ldots, x_n : \tau_n) \to \tau : \tau_1, \ldots, \tau_n \to \tau}$$

TD-CProp
$$\frac{\| \ lookup_v(x, c, \pi) \ \| = 1}{\pi, c \vdash (\textbf{let}\ x : \tau) : \tau}$$

TD-PMeth
$$\frac{\pi, [\textbf{self} \mapsto p, x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n] \vdash e : \sigma \qquad \pi \vdash \sigma \trianglelefteq \tau \qquad \forall \widetilde{m} \in lookup_{pm}(x, p, \pi), \widetilde{m}.type = \tau_1, \ldots \tau_n \to \tau}{\pi, p \vdash \textbf{func}\ x(x_1 : \tau_1, \ldots, x_n : \tau_n) \to \tau\ \{\ e\ \} : \tau_1, \ldots, \tau_n \to \tau}$$

TD-CMeth
$$\frac{\| \ lookup_{cm}(x, c, \pi) \ \| = 1 \qquad \pi, [\textbf{self} \mapsto c, x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n] \vdash e : \sigma \qquad \sigma \trianglelefteq \tau}{\pi, c \vdash \textbf{func}\ x(x_1 : \tau_1, \ldots, x_n : \tau_n) \to \tau\ \{\ e\ \} : \tau_1, \ldots, \tau_n \to \tau}$$

Figure 4. Featherweight Swift's declaration typing

3.3 Operational Semantics

As mentioned earlier, thanks to the omission of assignment, FS's semantics can be defined purely within its syntax. However, because methods are first-class citizens in FS, we need a way to represent bounded methods (i.e., methods in which references to **`self`** are bound to a receiver) as first-class terms. This is done by "demoting" a method to a non-member function in which all occurrences of self are syntactically substituted by the receiver. FS does not support unbounded methods: an expression of the form $c.m$ is not well-typed if $c \in I_c$ is a class identifier, as we would be unable to bind occurrences of **`self`** to a proper class instance. The set of values $\mathbb{V}$ to which a terminating program may evaluate is therefore composed of bounded methods and class initializers only. More formally, it is defined inductively as the minimal set such that $Md \subseteq \mathbb{V}$ and $c(x_1 : v_1, \ldots, x_n : v_n) \in \mathbb{V}$, where $c \in I_c$ is a class identifier, $x_1, \ldots, x_n \in I_x$ are property names and $v_1, \ldots, v_n \in \mathbb{V}$ are values.

We describe FS's operational semantics as "big-step" semantics rules, with a judgement of the form $\pi \vdash e \Downarrow v$, where $\pi \in \Pi$ is a program, $e \in E$ is an expression and $v \in \mathbb{V}$ is a value. All rules are presented in Figure 5. Unbounded methods are bound upon method selection by the rule E-Meth. It results that function calls can remain completely agnostic of this process, and simply substitute function parameters with their corresponding argument (rule E-Call).

Example 3.13. Consider the expression following expression, defined in the context of the program shown in Figure 1:

```
Pair(fst: A(), snd: B(foo: A())).snd
  .duplicated().withFst(C(foo: A(), bar: A()))
```

The following steps describe the evaluation of this expression. The rule E-Call applies first to evaluate the call to the method `withFst`. This triggers the evaluation of the function's callee, which happens to be another function call.

```
Pair(fst: A(), snd: B(foo: A())).snd
  .duplicated().withFst(C(foo: A(), bar: A()))
```

The rule E-Call applies again, this time to evaluate the call to the method `duplicated`. This triggers the evaluation of the function's callee, which is the access class property `snd`:

```
Pair(fst: A(), snd: B(foo: A())).snd
  .duplicated().withFst(C(foo: A(), bar: A()))
```

The rule E-Prop applies on the property selection, evaluating `Pair(fst: A(), snd: B(foo: A()))` with the rule E-Init. The term remains unchanged, as it is already in reduced form (i.e., it only features calls to class initializers). Therefore E-Prop's application produces a term `B(foo: A())` (i.e., the second element of the pair), which now acts as the receiver for the call to `duplicated`:

```
B(foo: A()).duplicated()
  .withFst(C(foo: A(), bar: A()))
```

This results in the pair `Pair(fst: B(foo: A()), snd: B(foo: A()))`, allowing the call to `withFst` to be evaluated:

```
Pair(fst:C(foo: A(), bar: A()), snd:B(foo: A()))
```

The program successfully terminates with this term, which cannot be further reduced.

The rule E-FCast, which corresponds to the evaluation of an forced cast, refers to a function *typeof* to obtain the runtime type of a particular term, formally given as follows:

$$typeof(c(x_1 : v_1, \ldots, x_n : v_n)) = c$$
$$typeof(\textbf{func}\ x(x_1 : \tau_1, \ldots, x_n : \tau_n) \to \tau) = \tau_1, \ldots, \tau_n \to \tau$$

The reader will notice that protocols are completely absent from the operational semantics, including the definition of the function *typeof*. Indeed, while protocols may be use

$$\textsc{E-Init}\ \frac{\forall i, \pi \vdash e_i \Downarrow v_i}{\pi \vdash c(x_1 : e_1, \ldots, x_n : e_n) \Downarrow c(x_1 : v_1, \ldots, x_n : v_n)} \qquad \textsc{E-Prop}\ \frac{\pi \vdash e \Downarrow c(\ldots, x_i : v_i, \ldots)}{\pi \vdash e.x_i \Downarrow v_i} \qquad \textsc{E-FCast}\ \frac{\pi \vdash e \Downarrow v \qquad \pi \vdash \mathit{typeof}(v) \trianglelefteq \tau}{\pi \vdash e\ \texttt{as!}\ \tau \Downarrow v}$$

$$\textsc{E-Meth}\ \frac{\pi \vdash e \Downarrow c(\overline{v}) \qquad \{\texttt{func}\ x(\overline{p}) \to \tau\ \{\ e\ \}\} = \mathit{lookup}_{cm}(x, c, \pi)}{\pi \vdash e.x \Downarrow \texttt{func}\ x(\overline{p}) \to \tau\ \{\ e[\texttt{self} \mapsto c(v)]\ \}} \qquad \textsc{E-Call}\ \frac{\pi \vdash e \Downarrow \texttt{func}\ x(x_1 : \tau_1, \ldots x_n : \tau_n) \to \tau\ \{\ e'\ \} \qquad \forall i, \pi \vdash e_i \Downarrow v_i \qquad \pi \vdash e'[x_1 \mapsto v_i, \ldots, x_n \mapsto v_n] \Downarrow v}{\pi \vdash e(e_1, \ldots, e_n) \Downarrow v} \qquad \textsc{E-GCast}\ \frac{\pi \vdash e \Downarrow v}{\pi \vdash e\ \texttt{as}\ \tau \Downarrow v}$$

Figure 5. Featherweight Swift's operational semantics

to type an expression, runtime values can only be class instances and bounded methods.

3.4 Properties

This section presents some key properties of FS's type system. The first relate to overloading and guarantees that classes do not define overloaded properties and/or methods in a well-typed program. The two first lemmas imply that there is a unique declaration corresponding to a name lookup in the rules TE-CProp and TE-CMeth, which respectively correspond to the resolution of property and method names. The third additionally guarantees that $\widetilde{m}$ be a proper method declaration and not a mere requirement in TE-CMeth.

Lemma 3.14. *Let c denote a class in a well-typed program $\pi \in \Pi$, then c does not define any overloaded property.*

$$\pi \vdash \texttt{class}\ c : d, P\ \{\ V\ M\ \} \implies \forall x \in I_x, \| \mathit{lookup}_v(x, c, \pi) \| \leq 1$$

Proof. The proof is straightforward by TD-Class. □

Lemma 3.15. *Let c denote a class in a well-typed program $\pi \in \Pi$, then c does not define any overloaded method.*

$$\pi \vdash \texttt{class}\ c : d, P\ \{\ V\ M\ \} \implies \forall x \in I_x, \| \mathit{lookup}_{cm}(x, c, \pi) \| \leq 1$$

Proof. The proof is straightforward by TD-Class. □

Lemma 3.16. *Let c denote a class in a well-typed program $\pi \in \Pi$, then all methods $\widetilde{m} \in \mathit{cmeths}(c, \pi)$ have an implementation.*

$$\pi \vdash \texttt{class}\ c : d, P\ \{\ V\ M\ \} \implies \forall \widetilde{m} \in \mathit{cmeths}(c, \pi), \widetilde{m} \in Md$$

Proof. The proof is straightforward by TD-Class. □

Next, we state the substitution lemmas.

Lemma 3.17. *Let τ and σ be two types in a well-typed program π such that $\pi \vdash \tau \trianglelefteq \sigma$. The set of properties and methods of σ is a subset of the set of properties and methods of τ.*

- *if $\tau \in I_c$ and $\sigma \in I_c$ denote classes, then $\mathit{props}(\sigma, \pi) \subseteq \mathit{props}(\tau, \pi)$ and $\mathit{cmeths}(\sigma, \pi) \subseteq \mathit{cmeths}(\tau, \pi)$;*
- *if $\tau \in I_c$ denotes a class and $\sigma = \texttt{protocol}\langle P \rangle$ is an instantiable protocol composition for some set of protocols $P \subseteq \mathcal{P}(I_p)$, then $\bigcup_{p \in P} \mathit{pmeths}(p, \pi) \subseteq \mathit{cmeths}(\tau, \pi)$;*
- *if $\tau = \texttt{protocol}\langle P \rangle$ and $\sigma = \texttt{protocol}\langle Q \rangle$ are two protocol compositions for some sets of protocols $P, Q \subseteq \mathcal{P}(I_p)$, then $\bigcup_{q \in Q} \mathit{pmeths}(q, \pi) \subseteq \bigcup_{p \in P} \mathit{pmeths}(p, \pi)$.*

Proof. The proof is by induction on the definition of the coercion operator (Definition 3.8). □

Lemma 3.18. *Let e be a well-typed expression such that $\pi, \Gamma \vdash e : \tau$. Let Γ' be typing context such that $\forall x \in \mathit{dom}(\Gamma), \Gamma'(x) \trianglelefteq \Gamma(x)$. Then $\pi, \Gamma' \vdash e : \tau'$ and $\tau' \trianglelefteq \tau$.*

Proof. The proof is by induction over the derivation FS's expression typing (Figure 3). □

FS supports recursion through **self**. As we chose to express its operational semantics with big step inference rules (in part to sidestep the so-called *stupid cast problem* [23]), stating type soundness with the usual progress and preservation theorems [42] requires to take diverging evaluations into account. This disqualifies the classical approach which consists of defining a predicate $\pi \vdash e \Downarrow \texttt{error}$ to characterize type errors, as it does not allow the distinction between well-typed programs that terminate and those that diverge. Another strategy is to rely on coinduction to define a relation of the form $\pi \vdash e \Downarrow^{\infty}$ which denotes non-terminating evaluations [26]. Although we do not include such definitions, for spatial reasons, we state type soundness with this principle.

Lemma 3.19 (Preservation)**.** *Let e is a well-typed expression such that $\pi, \varnothing \vdash e : \tau$ and $\pi \vdash e \Downarrow v$, then $\pi, \varnothing \vdash v : \sigma \implies \sigma \trianglelefteq \tau$.*

Proof. The proof is by induction over the derivation of the relation $\pi \vdash e \Downarrow v$ (Figure 5), using the substitution lemmas. □

Lemma 3.20 (Progress)**.** *Let e is a well-typed expression such that $\pi, \varnothing \vdash e : \tau$ and $\nexists v, \pi \vdash e \Downarrow v$, then $\pi \vdash e \Downarrow^{\infty}$.*

Proof (Structure). The proof is by coinduction and case analysis over e. □

Theorem 3.21 (Type Soundness). *Let e is a well-typed expression such that $\pi, \varnothing \vdash e : \tau$, then either $\pi \vdash e \Downarrow^{\infty}$ or $\exists v, \pi \vdash e \Downarrow v$.*

4 Reasoning About Swift

The design of FS allowed us to unveil or rediscover a handful of bugs with the official implementation of Swift's compiler. These bugs were discovered on the version 5.1.2. Three of them directly affect the language's handling of protocol composition. However, one relates to unbounded methods [37], which were supported in an earlier design of FS's type system but were eventually dropped for the sake of simplicity, and another involves generic types [24]. The remainder of this section focuses on the third one [36].

Swift protocols may refer to the variable **`Self`** to serve as a placeholder for the types that conform to them. It follows that if **`Self`** appears in a method requirement, then conforming types must implement a method that substitutes it with themselves. For instance, consider the following protocol:

```
1  protocol Boxable {
2    func boxed() -> Self
3  }
```

`Boxable` defines a single requirement for a method `boxed` that returns instances of the conforming type. Because of class inheritance, there is one important caveat to consider when conforming to this protocol. Indeed, the following class declaration is not well-typed:

```
1  class List: Boxable {
2    let next: List?
3    func boxed() -> List { List(next: self) }
4  }
```

While `List` satisfies `Boxable`'s requirements, a subclass will not. It will inherit a method `boxed` with a type $() \to$ `List`, which does not match its corresponding requirement. The solution is to substitute `List` with **`Self`** so that the method declaration refers to the type of the subclass. Unfortunately, this introduces another problem, as the type of the expression `List(next: self)` is not compatible with **`Self`**. A simple workaround is to force cast the expression:

```
1  class List: Boxable {
2    let next: List?
3    func boxed() -> Self {
4      List(next: self) as! Self
5    }
6  }
7
8  class DerivedList: List {
9    let foo: String
10 }
11
12 print(DerivedList(foo: "bar", next: nil)
13   .boxed().foo)
```

This program is now well-typed in both Swift and FS but should fail at runtime with a cast error. At line 12, the method `boxed` is invoked with a type $() \to$ `DerivedList`. Hence, **`Self`** should refer to `DerivedList` rather than `List`. It follows that `List(next: self) as! DerivedList` must fail, because `List` is not a subclass of `DerivedList`. However, as of the version 5.1.2, Swift would not detect such a failure and happily cast an instance of `List` as a `DerivedList`, thus bypassing type safety. Consequently, the program would have an undefined behavior upon accessing the property `foo` on the value returned from the method call.

We discovered this bug while stating FS's progress theorem. In order to review each case of the operational semantics systematically, we designed a collection of use-cases to test various derivation scenarios and compare them to actual executions to validate the result of our semantics. This eventually led us to the above program. It turns out that the problem is obvious under FS's semantics. Since `List` $\not\lhd$ `DerivedList`, the rule E-FCast cannot apply and the derivation is stuck at the cast expression.

5 Related Work

There is a rich literature dedicated to the study of programming languages by the means of minimal core calculi, usually building on top of the λ-calculus. Unfortunately, in its simplest, purest form, the λ-calculus is not ideal to reason about object-oriented abstractions, such as inheritance and composition. This has spawned the development of several calculi, including highly influential work such as Abadi and Cardelli's object calculus [1] and Featherweight Java [23]. This work borrows heavily from the latter. FJ is a popular language calculus that aims to provide a sound base for studying extensions to the Java language. Just as FS, it discards most of Java's features to focus on a minimal functional subset akin to the λ-calculus. FS differs from FJ in three significant ways. The first obviously relates to protocols, which add an extra dimension to method lookup and type polymorphism. Nonetheless, intuitions about substitution lemmas remain identical. The second difference is that methods in FS are first-class citizens. As such, they may be passed as arguments or returned from a function, whereas Featherweight Java is a first-order calculus. This slightly complicates the typing rules and operational semantics related to method calls, since we cannot assume that e is necessarily an expression of the form $e_0.m$ in a method call $e(e_1, \ldots, e_n)$. Lastly, FS is formalized in terms of a "big-step" operational semantics, whereas FJ is formulated using "small-step" semantics. This provides the calculus with a simpler model to express complex features such as concurrency and exceptions, although these shortcomings can be tackled by expressing properties on derivation at a meta-level [35]. On the other hand, this introduces inconsistencies in the derivation of expressions whose evaluation order has to be taken into account. For

instance, a special rule has to be defined to state the progress theorem with respect to failed cast expressions.

There exist a number of theoretical work aimed at formalizing protocol-oriented approaches. Traits, from which Swift protocols directly derive, are introduced in [39] for an untyped calculus [19]. The first attempt to type-check them statically is proposed in [20]. While their language offers more features than FS, including for instance property updates (i.e. assignments), composition is only supported for disjoint traits (i.e. traits without any common methods), and support for casting is limited. This latter limitation is addressed in [40]. Another paper, more closely related to this work, studies traits as an extension of FJ called FeatherTrait [28]. While FS only relies on overriding to disambiguate overlapping default implementations, FeatherTrait also formalizes aliasing and exclusion as composition mechanisms [33]. However, traits must be type-checked every time they are imported into a class, whereas protocol type-checking only occurs once in FS, promoting modularity. This limitation is later addressed in [27] by treating trait compositions as interfaces. FS adopts a similar approach, defining type coercion over class inheritance *and* protocol composition. More recent work on calculi with traits focus on state (e.g. [6, 16]) and behavioral properties (e.g. [3, 17]).

The POP discipline can be adopted in a variety of programming languages, including Scala, that shares a number of similarities with Swift with respect to its features and design choices. A formalization of Scala's type system is presented in the form of a core calculus, that includes traits [15]. The model features a more complex lookup mechanism than FS to accommodate the richer, more expressive set of type constructions supported by the Scala language. Typescript and Go are two modern languages that offer a different take on POP, using structural typing as a mechanism for retroactive modeling. This contrasts with FS, which requires protocol conformance to be explicitly stated in type declarations or extensions thereof. Both TypeScript and Go have been formalized in the form of core language calculi [7, 22].

6 Conclusion and Future Directions

We have presented Featherweight Swift, a core calculus to understand and reason about Swift's type system. Our approach mimics that of Featherweight Java [23]. FS drops all non-essential features and focuses only on the fundamental concepts that characterize Swift's type system. Our language comes with classes and protocol inheritance, supports retroactive modeling, and reproduces Swift's overriding mechanisms. On the other hand, it discards local and global variables, exceptions, concurrency and assignment. As a result, its syntax and semantics can be defined concisely, in a style reminiscent of the λ-calculus, and highlight Swift's type coercion and method lookup mechanism clearly and unambiguously.

We have discussed how FS served us to reason about Swift's semantics, which eventually led us to the discovery of a few bugs. This convinces us of the necessity to provide language users and developers alike with a suitable formal framework to generalize assumptions and envision how new ideas may interact with existing features. The entry barrier to new, modern programming languages is constantly pushed higher, as elaborate typing mechanisms make their way into mainstream languages. Therefore, it is paramount that these be accompanied by rigorous definitions.

This work prompts a number of exciting perspectives for future developments:

- Generic types contribute significantly to the power of Swift's type system, in particular when used in concert with protocols, and would therefore constitute a welcomed extension of our work. Although the generic extension of FJ [23] is an obvious starting point, one challenge is to provide a support for Swift's bounded polymorphism [11], whereas most approaches are restricted to parametric polymorphism. Fortunately, the more recent formalization of Go [22] provides promising insights to tackle this issue.
- Another natural extension would relate to the support of assignments. Once again, a number of extensions to Featherweight Java have already been proposed (e.g. [8, 30]). Here, the challenge is to properly reproduce Swift's distinction between value and reference types [35]. A promising lead in that direction would be to adopt a similar approach as Giannini's pure imperative calculus [21], that proposes to represent aliasing directly at the syntactic level.
- A third axis relates to concurrency. Although Swift's current concurrency model is based on a traditional multithreaded approach (for which FJ extensions have already been proposed [12, 34]), future implementations will feature actors on the top of coroutines [25]. Hence, a formal framework to reason about such approaches in the context of Swift's type system would be of great interest. There exist multiple language calculi from which we can draw inspiration in that front [5, 18].
- Some of the features that we have dismissed can be easily encoded on top of FS, but a formalization of these encodings has yet to be provided to validate their soundness. This would let us study a larger subset of Swift, for instance, to reason about observers, lazy, and computed properties.

Acknowledgments

The authors would like to thank the members of the Swift forums community for their precious help in the understanding of some of the most intricate edge cases of Swift's type system, and for their feedback on early drafts of this paper.

References

[1] Martín Abadi and Luca Cardelli. 1995. A Theory of Primitive Objects: Second-Order Systems. *Science of Computer Programming* 25, 2-3 (1995), 81–116. https://doi.org/10.1016/0167-6423(95)00010-0

[2] Martín Abadi and Luca Cardelli. 1996. *A Theory of Objects*. Springer, Berlin. https://doi.org/10.1007/978-1-4419-8598-9

[3] Reza Ahmadi, K. Rustan M. Leino, and Jyrki Nummenmaa. 2015. Automatic verification of Dafny programs with traits. In *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs, FTfJP 2015, Prague, Czech Republic, July 7, 2015*, Rosemary Monahan (Ed.). ACM, New York, 4:1–4:5. https://doi.org/10.1145/2786536.2786542

[4] Gerald Aigner and Urs Hölzle. 1996. Eliminating Virtual Function Calls in C++ Programs. In *ECOOP'96 - Object-Oriented Programming, 10th European Conference, Linz, Austria, July 8-12, 1996, Proceedings (Lecture Notes in Computer Science, Vol. 1098)*, Pierre Cointe (Ed.). Springer, Berlin, 142–166. https://doi.org/10.1007/BFb0053060

[5] Konrad Anton and Peter Thiemann. 2010. Typing Coroutines. In *Trends in Functional Programming - 11th International Symposium, TFP 2010, Norman, OK, USA, May 17-19, 2010. Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6546)*, Rex L. Page, Zoltán Horváth, and Viktória Zsók (Eds.). Springer, Berlin, 16–30. https://doi.org/10.1007/978-3-642-22941-1_2

[6] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. 2008. Stateful traits and their formalization. *Computer Languages, Systems and Structures* 34, 2-3 (2008), 83–108. https://doi.org/10.1016/j.cl.2007.05.003

[7] Gavin M. Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8586)*, Richard E. Jones (Ed.). Springer, Berlin, 257–281. https://doi.org/10.1007/978-3-662-44202-9_11

[8] Gavin M. Bierman, Matthew J. Parkinson, and Andrew M. Pitts. 2003. *MJ: An imperative core calculus for Java and Java with effects*. Technical Report. University of Cambridge, Computer Laboratory.

[9] Gilad Bracha and William R. Cook. 1990. Mixin-based Inheritance. In *Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming (OOPSLA/ECOOP), Ottawa, Canada, October 21-25, 1990, Proceedings*, Akinori Yonezawa (Ed.). ACM, New York, 303–311. https://doi.org/10.1145/97945.97982

[10] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. 1995. On Binary Methods. *TAPOS* 1, 3 (1995), 221–242.

[11] Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *Comput. Surveys* 17, 4 (1985), 471–522. https://doi.org/10.1145/6041.6042

[12] Elias Castegren and Tobias Wrigstad. 2018. OOlong: an extensible concurrent object calculus. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*. ACM, New York, 1022–1029. https://doi.org/10.1145/3167132.3167243

[13] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd D. Millstein. 2000. MultiJava: modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000*, Mary Beth Rosson and Doug Lea (Eds.). ACM, New York, 130–145. https://doi.org/10.1145/353171.353181

[14] Karl Crary, Stephanie Weirich, and J. Gregory Morrisett. 2002. Intensional polymorphism in type-erasure semantics. *Journal of Functional Programming* 12, 6 (2002), 567–600. https://doi.org/10.1017/S0956796801004282

[15] Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. 2006. A Core Calculus for Scala Type Checking. In *Mathematical Foundations of Computer Science 2006, 31st International Symposium, MFCS 2006, Stará Lesná, Slovakia, August 28-September 1, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4162)*, Rastislav Kralovic and Pawel Urzyczyn (Eds.). Springer, Berlin, 1–23. https://doi.org/10.1007/11821069_1

[16] Tom Van Cutsem, Alexandre Bergel, Stéphane Ducasse, and Wolfgang De Meuter. 2009. Adding State and Visibility Control to Traits Using Lexical Nesting. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5653)*, Sophia Drossopoulou (Ed.). Springer, Berlin, 220–243. https://doi.org/10.1007/978-3-642-03013-0_11

[17] Ferruccio Damiani, Johan Dovland, Einar Broch Johnsen, and Ina Schaefer. 2014. Verifying traits: an incremental proof system for fine-grained reuse. *Formal Asp. Comput.* 26, 4 (2014), 761–793. https://doi.org/10.1007/s00165-013-0278-3

[18] Ana Lúcia de Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems* 31, 2 (2009), 6:1–6:31. https://doi.org/10.1145/1462166.1462167

[19] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. 2006. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems* 28, 2 (2006), 331–388. https://doi.org/10.1145/1119479.1119483

[20] Kathleen Fisher and John Reppy. 2003. *Statically Typed Traits*. Technical Report. AT&T Labs.

[21] Paola Giannini, Marco Servetto, and Elena Zucca. 2018. A Syntactic Model of Mutation and Aliasing, In Proceedings of the 12th Workshop on Developments in Computational Models and 9th Workshop on Intersection Types and Related Systems, DCM/ITRS 2018, Oxford, UK, 8th July 2018, Michele Pagani and Sandra Alves (Eds.). *Electronic Proceedings in Theoretical Computer Science* 293, 39–55. https://doi.org/10.4204/EPTCS.293.4

[22] Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. 2020. Featherweight Go. *CoRR* abs/2005.11710 (2020). arXiv:2005.11710 https://arxiv.org/abs/2005.11710

[23] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems* 23, 3 (2001), 396–450. https://doi.org/10.1145/503502.503505

[24] Hamish Knight. 2020. The dynamic nature of a protocol's Self type isn't checked when equated with another associated type. https://bugs.swift.org/browse/SR-10713. 2020 (Accessed on July 22, 2020).

[25] Chris Lattner. 2020. Swift Concurrency Manifesto. https://gist.github.com/lattner/31ed37682ef1576b16bca1432ea9f782. 2020 (Accessed on July 22, 2020).

[26] Xavier Leroy and Hervé Grall. 2009. Coinductive big-step operational semantics. *Information and Computation* 207, 2 (2009), 284–304. https://doi.org/10.1016/j.ic.2007.12.004

[27] Luigi Liquori and Arnaud Spiwack. 2008. Extending FeatherTrait Java with Interfaces. *Theor. Comput. Sci.* 398, 1-3 (2008), 243–260. https://doi.org/10.1016/j.tcs.2008.01.051

[28] Luigi Liquori and Arnaud Spiwack. 2008. FeatherTrait: A modest extension of Featherweight Java. *ACM Transactions on Programming Languages and Systems* 30, 2 (2008), 11:1–11:32. https://doi.org/10.1145/1330017.1330022

[29] Barbara Liskov and Jeannette M. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems* 16, 6 (1994), 1811–1841. https://doi.org/10.1145/197320.197383

[30] Julian Mackay, Hannes Mehnert, Alex Potanin, Lindsay Groves, and Nicholas Robert Cameron. 2012. Encoding Featherweight Java with

assignment and immutability using the Coq proof assistant. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, FTfJP 2012, Beijing, China, June 12, 2012*. ACM, New York, 11–19. https://doi.org/10.1145/2318202.2318206

[31] John C. Mitchell and Gordon D. Plotkin. 1988. Abstract Types Have Existential Type. *ACM Transactions on Programming Languages and Systems* 10, 3 (1988), 470–502. https://doi.org/10.1145/44501.45065

[32] Dmitri Nesteruk. 2018. *Maybe Monad*. Apress, Berkeley, CA, 305–308. https://doi.org/10.1007/978-1-4842-3603-1_25

[33] Oscar Nierstrasz, Stéphane Ducasse, and Nathanael Schärli. 2006. Flattening Traits. *Journal of Object Technology* 5, 4 (2006), 129–148. https://doi.org/10.5381/jot.2006.5.4.a4

[34] Johan Östlund and Tobias Wrigstad. 2010. Welterweight Java. In *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*. Springer, Berlin, 97–116. https://doi.org/10.1007/978-3-642-13953-6_6

[35] Dimitri Racordon. 2019. *Revisiting Memory Assignment Semantics in Imperative Programming Languages*. Ph.D. Dissertation. University of Geneva, Geneva, Switzerland.

[36] Dimitri Racordon. 2020. Bad downcast to Self not caught at runtime. https://bugs.swift.org/browse/SR-11818. 2020 (Accessed on July 22, 2020).

[37] Dimitri Racordon. 2020. Crash when attempting to assign an unbound method from a protocol. https://bugs.swift.org/browse/SR-11769. 2020 (Accessed on July 22, 2020).

[38] Aminata Sabane, Yann-Gaël Guéhéneuc, Venera Arnaoudova, and Giuliano Antoniol. 2017. Fragile base-class problem, problem? *Empirical Software Engineering* 22, 5 (2017), 2612–2657. https://doi.org/10.1007/s10664-016-9448-2

[39] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. 2003. Traits: Composable Units of Behaviour. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*. Springer, Berlin, 248–274. https://doi.org/10.1007/978-3-540-45070-2_12

[40] Charles Smith and Sophia Drossopoulou. 2005. *Chai*: Traits for Java-Like Languages. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3586)*, Andrew P. Black (Ed.). Springer, Berlin, 453–478. https://doi.org/10.1007/11531142_20

[41] Stefan Wehr and Peter Thiemann. 2011. JavaGI: The Interaction of Type Classes with Interfaces and Inheritance. *ACM Transactions on Programming Languages and Systems* 33, 4 (2011), 12:1–12:83. https://doi.org/10.1145/1985342.1985343

[42] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093