SFJ: An implementation of Semantic Featherweight Java *

Artem Usov and Ornela Dardha

School of Computing Science, University of Glasgow, Glasgow, United Kingdom
2296905U@student.gla.ac.uk, ornela.dardha@glasgow.ac.uk

Abstract. There are two approaches to defining subtyping relations: the syntactic and the semantic approach. In semantic subtyping, one defines a model of the language and an interpretation of types as subsets of this model. Subtyping is defined as inclusion of subsets denoting types. An orthogonal subtyping question, typical of object-oriented languages, is the nominal versus the structural subtyping. Dardha et al. [11,12] defined boolean types and semantic subtyping for Featherweight Java (FJ) and integrated both nominal and structural subtyping, thus exploiting the benefits of both approaches. However, these benefits were illustrated only at a theoretical level, but not exploited practically.

We present SFJ—Semantic Featherweight Java, an implementation of FJ which features boolean types, semantic subtyping and integrates nominal as well as structural subtyping. The benefits of SFJ, illustrated in the paper and the accompanying video (with audio/subtitles) [27], show how static type-checking of boolean types and semantic subtyping gives higher guarantees of program correctness, more flexibility and compactness of program writing.

Keywords: Nominal subtyping · Structural subtyping · Semantic Featherweight Java · Object-oriented languages · Boolean types · Type theory.

1 Introduction

There are two approaches to defining subtyping relations: the syntactic and the semantic approach. Syntactic subtyping [20] is more mainstream in programming languages and is defined by means of a set of formal deductive subtyping rules. Semantic subtyping [11,10] is more recent and less known: one defines a formal model of the language and an interpretation of types as subsets of this model. Then, subtyping is defined as set inclusion of subsets denoting types.

Orthogonally, for object-oriented languages there are two approaches to defining subtyping relations: the nominal and the structural approach [21,22]. Nominal subtyping is based on declarations by the developer and is name-based: “A is

* Supported by the UK EPSRC grant EP/K034413/1, “From Data Types to Session Types: A Basis for Concurrency and Distribution” (ABCD), and by the EU HORIZON 2020 MSCA RISE project 778233 “Behavioural Application Program Interfaces” (BehAPI).
A subtype of $B$ if and only if it is declared to be so, that is if the class $A$ extends (or implements) the class (or interface) $B$’. Structural subtyping instead is based on the structure of a class, its fields and methods: “a class $A$ is a subtype of a class $B$ if and only if the fields and methods of $A$ are a superset of the fields and methods of $B$, and their types in $A$ are subtypes of the types in $B$”. For example, the set of inhabitants of a class Student is smaller than the set of inhabitants of a class Person, as each Student is a Person, but not the other way around. However, the set of fields and methods of Student is a superset of that of Person. Hence, Student is a structural subtype of Person, even if it is not declared so.

Dardha et al. [11,12] define boolean types—based on set-theoretic operations such as and, not, or—and semantic subtyping for Featherweight Java (FJ) [17]. This approach allows for the integration of both nominal and structural subtyping in FJ, bringing in higher guarantees of program correctness, flexibility and compactness in program writing. Unfortunately, these benefits were only presented at a theoretical level and not exploited practically, due to the lack of an implementation of the language, its types and type system.

In this paper, we present SFJ—Semantic Featherweight Java § 3, an implementation of FJ with boolean types and semantic subtyping. In SFJ the developer has a larger and more expressive set of types, by using boolean connectives and, not, or, with the expected set-theoretic interpretation. On the other hand, this added expressivity does not add complexity. Rather the opposite is true, as the developer has an easier, more compact and elegant way of programming. SFJ integrates both structural and nominal subtyping, and the developer can choose which one to use. Finally, as discussed in Dardha et al. [12] §§ 8.4, thanks to semantic subtyping, we can easily encode in SFJ standard programming constructs and features of the full Java language, such as lists, or overloading classes via multimethods [5], which are missing in FJ, thus making SFJ a more complete language closer to Java.

**Example 1 (Polygons).** This will be our running example both in the paper and in the tool video [27] to illustrate the benefits of boolean types and semantic subtyping developed by Dardha et al. [11,12] and implemented as SFJ.

Consider the set of polygons, such as triangles, squares and rhombuses given by a class hierarchy. We want to define a method diagonal that takes a polygon and returns the length of its longest diagonal. This method makes sense only if the polygon passed to it has at least four sides, hence triangles are excluded. In Java this could be implemented in the following ways:

```java
class Polygon {
    ...
}
class Triangle extends Polygon {
    ...
}
class Other_Polygons extends Polygon {
    double diagonal(Other_Polygons shape) {
        ...
    }
    ...
}
class Square extends Other_Polygons {
    ...
}
class Rhombus extends Other_Polygons {
    ...
}
```
Or by means of an interface `Diagonal`:

```
public interface Diagonal {
    double diagonal(Polygon shape);
}
class Polygon {...}
class Triangle extends Polygon {...}
class Square extends Polygon implements Diagonal {...}
class Rhombus extends Polygon implements Diagonal {...}
    // other polygons ...
```

Now, suppose our class hierarchy is such that `Polygon` is the parent class and all other geometric figures extend `Polygon`, which is how one would naturally define the set of polygons. Suppose the class hierarchy is given and is part of legacy code, which cannot be changed. Then again, a natural way to implement this in Java is by defining the method `diagonal` in the class `Polygon` and using an `instanceof`, for example, inside a `try-catch` construct. Then, an exception would be thrown at run time, if the argument passed to the method is a triangle.

We propose a more elegant solution, by combining boolean types and semantic subtyping, where only static type-checking is required and we implement this in SFJ [27]: it is enough to define a method `diagonal` that has an argument of type `Polygon` and not `Triangle`, thus allowing the type-checker to check at compile time the restrictions on types:

```
class Polygon {...}
class Triangle extends Polygon {...}
class Square extends Polygon {...}
class Rhombus extends Polygon {...}
    ...

class Diagonal {
    ...
        double diagonal(((Polygon and not Triangle) shape){...})
    }
```

We can now call `diagonal` on an argument of type `Polygon`: if the polygon is not a `Triangle`, then the method computes and returns the length of its longest diagonal; otherwise, there will be a type error at compile time.

**Structure of the paper:** In § 2 we present the types and terms of the SFJ language. In § 3 we present the design and implementation of SFJ; we discuss our two main algorithms, Algo. 1 in § 3.1 which checks the validity of type definitions, and Algo. 2 in § 3.2 which generates the semantic subtyping relation. Further, we discuss typing of SFJ in § 3.3 nominal vs. structural subtyping in § 3.5 method types in § 3.6 and code generation in § 3.7. We discuss related work and conclude the paper in § 4.
2 Background

The technical developments behind semantic subtyping and its properties are complex, however, they are completely transparent to the programmer. The framework is detailed and proved correct in the relevant work by Dardha et al. [11, 12], and SFJ builds on that framework.

In this section we will briefly detail the types and terms of SFJ.

2.1 Types

The syntax of types $\tau$ is defined by the following grammar:

$$
\begin{align*}
\tau & ::= \alpha \mid \mu \\
\alpha & ::= 0 \mid B \mid [\tilde{l} : \tau] \mid \alpha \text{ and } \alpha \mid \text{ not } \alpha \\
\mu & ::= \alpha \rightarrow \alpha \mid \mu \text{ and } \mu \mid \text{ not } \mu
\end{align*}
$$

Types $\alpha$-types are used to type fields and the $\mu$-types are used to type methods. Type 0 is the empty type. Type $B$ denotes the basic types, such as integers, booleans, etc. Record types $[\tilde{l} : \tau]$, where $\tilde{l}$ is a sequence of disjoint labels, are used to type objects. Arrow types $\alpha \rightarrow \alpha$ are used to type methods.

The boolean connectives and and not in the $\alpha$-types and $\mu$-types have their expected set-theoretic meanings. We let $\alpha \\\\lor \\\alpha'$ denote $\alpha$ and (not $\alpha'$), and $\alpha$ or $\alpha'$ denote not(not $\alpha$ and (not $\alpha'$)).

2.2 Terms

The syntax of terms is defined by the following grammar and is based on the standard syntax of terms in FJ [17]:

- **Class declaration**
  $$
  L ::= \text{class } C \text{ extends } C \{ \tilde{\alpha} a; K; \tilde{M} \}
  $$

- **Constructor**
  $$
  K ::= C (\tilde{x}) \{ \text{ super}(\tilde{x}); \text{ this}.a = \tilde{x}; \}
  $$

- **Method declaration**
  $$
  M ::= \alpha m(\alpha x) \{ \text{ return } e; \}
  $$

- **Expressions**
  $$
  e ::= x \mid c \mid e.a \mid e.m(e) \mid \text{ new } C(e)
  $$

We assume an infinite set of names, with some special names: Object denotes the root class, this denotes the current object and super denotes the parent object. We let $A, B, \ldots$ range over classes; $a, b, \ldots$ over fields; $m, n, \ldots$ over methods and $x, y, z, \ldots$ over variables.

A program $(\tilde{L}, e)$ is a pair of a sequence of class declarations $\tilde{L}$, giving rise to a class hierarchy as specified by the inheritance relation, and an expression $e$ to be evaluated. A class declaration $L$ specifies the name of the class, the name of the parent class it extends, its typed fields, the constructor $K$ and its method declarations $M$. The constructor $K$ initialises the fields of the object by assigning values to the fields inherited by the super-class and to the fields declared in the current this class. A method declaration $M$ specifies the signature of the
method, namely the return type, the method name and the formal parameter as well as the body of the method. Notice that in our theoretical development we use unary methods, without loss of generality: tuples of arguments can be modelled by an object that instantiates a “special” class containing as fields all the needed arguments. Expressions $e$ include variables, constants, field access, method call and object creation.

Following FJ [17], we rule out ill-formed programs, such as declaring a constructor named $B$ within a class named $A$, or multiple fields or methods having the same name, or fields having the same type as the class they are defined in.

3 The SFJ Language

3.1 On Valid Type Definitions

Since we want to use types $\tau$ to program in SFJ, we restrict them to finite trees whose leaves are basic types $B$ with no cycles. For example, a recursive type $\alpha = [ a : \alpha ]$ denotes an infinite program tree $\text{new } C(\text{new } C(\cdots))$, hence we rule it out as it is not inhabitable. Similarly, the types $\alpha = [ b : \beta ]$, $\beta = [ a = \alpha ]$ create a cycle and thus would not be inhabitable. Notice that these types can be defined and inhabited in Java by assigning $null$ to all fields in a class, however they are not useful in practice.

SFJ is implemented using ANTLR [24]. We start by defining the grammar of the language in Extended Backus-Naur Form (EBNF), following § 2.1 and by running ANTLR, we can automatically generate a parser for SFJ and extend it in order to implement the required checks for our types and type system.

Running the parser on an SFJ program returns an abstract syntax tree (AST) of that program. When visiting the AST, we check if the program is well-formed, following the intuition at the end of § 2. We mark any classes containing only fields typed with basic types as resolved, otherwise we mark them as unresolved. Using this information, Algo. 1 checks if the type definitions in a program are valid, namely if they are finite trees whose leaves are basic types with no cycles. The algorithm returns $True$ only if all the types in the SFJ program are resolved, otherwise it returns $False$, meaning there is at least one type definition which is invalid and contains a cycle.

3.2 Building Semantic Subtyping for SFJ

If Algo. 1 returns $True$, meaning all type definitions in a program are valid, we can then build the semantic subtyping. Leveraging the interpretation of types as sets of values to define semantic subtyping for FJ [11,12], in SFJ we keep track of the semantic subtyping relation by defining a map from a type to the set of its subtypes, satisfying the property that the set of values of a subtype is included in the set of values of the type.
Algorithm 1: Validity Check for Type Definitions

Input : classes, the set of classes in an SFJ program marked resolved, if their fields contain only basic types, unresolved otherwise.

Output : True if all classes are valid type definitions, False otherwise.

1 begin
2 do
3   resolutionOccured ← False
4   for class that is unresolved in classes do
5     resolved ← True
6     for field in class that contains a class type do
7       if type of field is unresolved then
8         resolved ← False
9       end
10   end
11   if resolved = True then
12     class ← resolved
13     resolutionOccured ← True
14   end
15 while resolutionOccured = True
16 if not all classes are resolved then
17   return False
18 else
19   return True
20 end
21 end

We start with basic types and let Universe be a supertype of all types. The full mapping for basic types is defined in Map 3.1.

\[
\begin{align*}
  \text{Double} & = \{\text{Double, Float, Int, Short, Byte}\} \\
  \text{Long} & = \{\text{Long, Int, Short, Byte}\} \\
  \text{Short} & = \{\text{Short, Byte}\} \\
  \text{Boolean} & = \{\text{Boolean}\} \\
  \text{Universe} & = \{\text{Double, Float, Long, Int, Short, Byte, Boolean, Void}\}
\end{align*}
\]

Note that Int is not a subtype of Float as a 32-bit float cannot represent the whole set of 32-bit integer values accurately and therefore Int is not fully set-contained in Float, however this is not the case for Int and Double. Similarly, Long is not a subtype of Double.

Algo. 2 builds the semantic subtyping relation for all class types of an SFJ program by calling the function generateRelation. Given that classes are valid type definitions by Algo. 1, we are guaranteed that Algo. 2 will terminate.
Algorithm 2: Semantic Subtyping for SFJ—\textit{generateRelation}

\begin{verbatim}
Input : \textit{classes}, the set of classes in an SFJ program.
\textit{relation}, the mapping of types to the set of subtypes, initially Map 3.1.

\begin{verbatim}
1 begin
2     Function \textit{generateRelation}(\textit{classes: List<Class>}):
3         \textit{unprocessed}: List < Class > ←− []
4         for \textit{class} in \textit{classes} do
5             if addClass(\textit{class}) = \texttt{False} then
6                 \textit{unprocessed}.add(\textit{class})
7             end
8         end
9         if \textit{untyped} \neq [] then
10            \textit{generateRelation}(\textit{unprocessed})
11        end
12     end
13 /* algorithm continued on next page... */
\end{verbatim}
\end{verbatim}

The semantic subtyping generated by Algo. 2 is a preorder: it is reflexive and transitive. This is also illustrated by Map 3.1.

Some comments on Algo. 2 follow. In function \textit{generateRelation} we iterate over the set of classes in an SFJ program. If the class currently being processed contains types in its fields or methods not present in the subtyping relation (lines 5 and 30, 42 in the continuation of the algorithm in the next page), then we add the current class to the list of \textit{unprocessed} classes (line 6) so we can process its fields and methods first and the class itself later after having all required type information. The set of unprocessed classes will then be inspected again in a recursive call (line 10).

The next two functions of the algorithm, addClass and checkSuperSet given in the next page, check subtyping for the current \textit{class} being processed and update \textit{relation}, which is a mapping from a type to its subtypes and originally only consists of entries from Map 3.1. In function \textit{addClass(\textit{class})} we check if the type \textit{class} is a subtype of an existing type in \textit{relation} (lines 15-18), as well as the opposite, meaning if \textit{class} is a supertype of an existing type in \textit{relation} (line 19). In order to do so \textit{checkSuperSet} checks all fields (lines 28-39) and all methods (lines 40-51) in \textit{class} and compares them with an \textit{existingClass} in \textit{relation}. If a subtyping relation is established, then it is added to \textit{relation} (line 53). Finally, upon returning from \textit{checkSuperSet}, we also add \textit{class} to its own relation (line 21) to satisfy reflexivity and to \textit{Universe} (line 22), which is a supertype of all types.

It is worth noticing that the subtyping algorithm finds all nominal and structural subtypes of a given type. This is due to the fact that all pairs of types are inspected. Recall from § 3.1 that nominal subtyping is name-based and given by the class hierarchy defined by the programmer, whereas structural subtyping is structure-based and given by the set-inclusion of fields and methods. In par-
Algorithm 2: Semantic Subtyping for SFJ—\textit{addClass} and \textit{checkSuperSet}

/* ...algorithm continued from previous page */

13 14  \textbf{Function} \textit{addClass}(class: Class) \rightarrow boolean:
15  \hspace{1em} \textbf{for} existing class type in relation \textbf{do}
16  \hspace{2em} \textbf{if} checkSuperSet(class, existingClass) = False \textbf{then}
17  \hspace{3em} \textbf{return} False
18  \hspace{2em} end
19  \hspace{1em} checkSuperSet(existingClass, class)
20  \hspace{1em} end
21  \hspace{1em} relation[class].add(class)
22  \hspace{1em} relation[Universe].add(class)
23  \textbf{return} True
24  \textbf{end}
25
26  \textbf{Function} \textit{checkSuperSet}(class: Class, other: Class) \rightarrow boolean:
27  \hspace{1em} flag \leftarrow True
28  \hspace{1em} \textbf{for} field in class \textbf{do}
29  \hspace{2em} \textbf{if} field contains type not in relation \textbf{then}
30  \hspace{3em} \textbf{return} False
31  \hspace{2em} end
32  \hspace{1em} \textbf{if} other does not contain field \textbf{then}
33  \hspace{2em} flag \leftarrow False
34  \hspace{1em} else
35  \hspace{2em} \textbf{if} other.field.types does not fully contain field.types \textbf{then}
36  \hspace{3em} flag \leftarrow False
37  \hspace{2em} end
38  \hspace{1em} end
39  \hspace{1em} \textbf{for} method in class \textbf{do}
40  \hspace{2em} \textbf{if} method contains type not in relation \textbf{then}
41  \hspace{3em} \textbf{return} False
42  \hspace{2em} end
43  \hspace{1em} \textbf{if} other does not contain method \textbf{then}
44  \hspace{2em} flag \leftarrow False
45  \hspace{1em} else
46  \hspace{2em} \textbf{if} other.method.types does not fully contain method.types \textbf{then}
47  \hspace{3em} flag \leftarrow False
48  \hspace{2em} end
49  \hspace{1em} end
50  \hspace{1em} if flag = True \textbf{then}
51  \hspace{2em} relation[other].add
52  \hspace{1em} end
53 \textbf{end}
ticular, structural subtyping is contra-variant with respect to this set-inclusion. Algo. 2 finds all structural subtypes of a given class because it checks that its fields and methods are a superset of existing types in relation. For example, all classes are structural subtypes of type empty = []. On the other hand, it also finds all nominal subtypes because a class inherits all fields and methods of its superclass and as such its fields and methods are a superset of its superclass. This means that checking for structural subtyping is enough because nominal subtyping will be captured due to inheritance of fields and methods.

Finally, a note on complexity. The complexity of Algo. 1 is \( O(n) \), and the complexity of Algo. 2 is \( O(n^2) \), with \( n \) being the size of the input. The reason for a quadratic complexity of Algo. 2 is due to the symmetric check of structural subtyping between a class and an existingClass in relation. Notice that if we were to only work with nominal subtyping, then we would only require traversing the class hierarchy once, which gives an \( O(n) \) complexity.

### 3.3 Type System for SFJ

The type system for the SFJ language, given in § 2.2, is based on the type system by Dardha et al. [11,12] where the formal typing rules and soundness properties are detailed. As these formal developments are beyond the scope of this paper, we discuss typing for SFJ only informally.

A program \((L,e)\) is well typed if both \(L\) and \(e\) are well typed. Class declaration \(L\) and method declaration \(M\) are well typed if all their components are well typed. Let us move onto expressions \(E\). Field access \(e.a\), method call \(e.m(e)\) and object creation \(\text{new } C(\tilde{e})\) are typed in the same way as in Java: we inspect the type of the field and the type of the method and its arguments to determine the type of the field access and method call, respectively. The type of an object creation is determined by the type of its class. Regarding constants, in order to respect the set-theoretic interpretation of types as sets of values, we type constants with the most restrictive type, i.e., the type representing the smallest set of values containing the value itself. For example, the type system would assign to the value 42 the type byte, which is the smallest in the sequence byte, short, int (see Map 3.1 for details).

Finally, the subtyping relation generated by Algo. 2 is used in the type system for the SFJ language via a subsumption typing rule:

\[
\Gamma \vdash e : \alpha_1 \\ \alpha_1 \leq \alpha_2 \\ \Gamma \vdash e : \alpha_2
\]

We read this typing rule as follows: if an expression \(e\) is of type \(\alpha_1\) under a typing context \(\Gamma\) (details of a typing context are irrelevant here) and type \(\alpha\) is a subtype \(\leq\) of \(\alpha_2\), then expression \(e\) can be typed with \(\alpha_2\).

### 3.4 Polygons: Continued

Let us illustrate the semantic subtyping algorithm on our Polygons given in Ex. 1. Algo. 2 generates the subtyping relation given in Map 3.2 together with
the subtyping relation for basic types, omitted here and defined in Map 3.1. Notice that the mapping for Universe is extended with the new types for polygons.

\[
\begin{align*}
\text{Polygon} &= \{\text{Polygon, Triangle, Square, Rhombus}\} & \text{Triangle} &= \{\text{Triangle}\} \\
\text{Square} &= \{\text{Square}\} & \text{Rhombus} &= \{\text{Rhombus}\} \\
\text{Universe} &= \{\text{Double, Float, Long, Int, Short, Byte, Boolean, Void, Polygon, Square, Square, Rhombus, Diagonal}\}
\end{align*}
\]

(3.2)

Recall the method diagonal in class Diagonal, with signature

\[
\text{double diagonal}((\text{Polygon and not Triangle}) \text{ shape})
\]

The result of the set operation on its parameter type gives the following set of polygons:

\[
\text{Polygon and not Triangle} = \{\text{Polygon, Square, Rhombus}\}
\]

In order to define the not Triangle type we need the Universe type so that we can define it as Universe \ Triangle. Then, the and connective is the intersection of sets of Polygon with not Triangle.

If we write in our SFJ program the following expression:

\[
\text{(new Diagonal()).diagonal(new Square())}
\]

the argument \text{new Square()} of the diagonal method is of type Square, by the type system in § 3.3 and Square is contained in the set of the parameter type of the method, so this expression will successfully type-checks.

However, if we write the following SFJ expression:

\[
\text{(new Diagonal()).diagonal(new Triangle())}
\]

Type Triangle is not contained in \{Polygon, Square, Rhombus\}, therefore this expression will not type-check and will return a type error at compile time.

This is further illustrated in the accompanying video of this paper [27].

3.5 Nominal vs. Structural Subtyping

In this section we will comment on pros and cons of nominal vs. structural subtyping.

Structural subtyping allows for more flexibility in defining this relation and the user does not need to explicitly define it, as would do with nominal subtyping. However, for this flexibility we might need to pay in meaning. For example, consider the following two structurally equivalent classes, hence record types coordinate = \{x : int, y : int, z : int\} and colour = \{x : int, y : int, z : int\}. While they can be used interchangeably in a type system using structural subtyping, their “meaning” is different and we might want to prohibit it, because intuitively speaking we do not want to use a colour where a coordinate is expected.
On the other hand, while nominal subtyping can avoid the above problem, it can introduce others and in particular, a developer can define an overridden method to perform the opposite logic to what the super class is expecting, as illustrated by the following classes in Java:

```java
class A extends Object {
    int n;
    int length() { return -n; }
}

class B extends A {
    int length() { return n; }
}
```

Both approaches have their pros and cons, and they leave an expectation on the developer to use the logic behind subtyping correctly when writing code. The integration of both subtyping approaches in SFJ gives the developer the freedom to choose the most suitable subtyping relation to use for a given task.

### 3.6 Methods in SFJ

**On multimethods** Since FJ is a core language, some features of the full Java are removed, such as overloading methods. In our framework, by leveraging the expressivity of boolean connectives and semantic subtyping, we are able to restore overloading, among other features [12, §8.4]. We can thus model multimethods, [3], which according to the authors is “very clean and easy to understand [...] it would be the best solution for a brand new language”. As an example, taken from Dardha et al. [11,12], consider the following class declarations:

```java
class A extends Object {
    int length(String s) { ... }
}

class B extends A {
    int length(int n) { ... }
}
```

Method length has type `String → int` in class A. However, because class B extends class A, length has type `String → int` and `int → int` in class B, which can be simplified to `(String or Int) → int`.

**Method types** Let us illustrate the method types given in §2.1 via an alternative implementation of the class `Diagonal` at the end of Ex. 1.

```java
class Diagonal {
    ...
    double diagonal((diagonal : void → double) shape)
    { return shape.diagonal(); }
}
```

We define the type of the (outside) `diagonal` method as accepting any type and its subtypes implementing the (inside) `diagonal` method with type signature `void` to `double`.
In order to type check an argument passed to the (outside) diagonal method, at compile time we build a collection of types \{type_1, type_2, \ldots\} which are class types where the (inside) diagonal method is defined. As such, we iterate over the list of classes in an SFJ program (as we did in Algorithm 2) to check for the required method. The resulting collection of types is the union of all classes where diagonal is defined together with their subtypes \( [\text{type}_1] \cup [\text{type}_2] \cup \ldots \), where each \([\text{type}_i]\) denotes a mapping of \( \text{type}_i \) to the set of its subtypes, similar to Map 3.2.

However, calculating this collection of types for each method of every class would be computationally inefficient and most importantly unnecessary as only few methods would in turn be used as method types. Therefore we only compute them on demand during type-checking when we come across such a type.

We can therefore use method types to statically include or exclude a portion of our class hierarchy. However, unlike with interfaces as in Example 1, the values that can be accepted by a method type do not have to be related to each other in any way in the class hierarchy. This indeed is useful if we are dealing with legacy code as we can still accept all classes where diagonal is defined, without having to go back and add interface implementations.

### 3.7 Code Generation

SFJ only includes the typechecking component of the language. In this section, we provide a sketch of the code generation algorithm, which is work in progress.

Given the similarity of SFJ to Java, our approach is to translate an SFJ program into Java bytecode and then run it on the Java Virtual Machine (JVM) \[19\]. This is a standard approach also used by other object-oriented languages, for example, Kotlin\[1\].

The main challenge in translating SFJ into bytecode is translating types using boolean connectives. For example, a field \( f_1 \) of type \text{int or bool}, will be translated as two Java fields, one of type \text{int} and one of type \text{bool}, and only one of the two types will be inhabited by a value. In order to achieve this, we first analyse our program and reduce the boolean types by keeping only the alternatives which actually get used in the program. For example, if the field of type \text{int or bool} only ever gets initialised with a boolean value, we can reduce it and make it a field of a single type \text{bool}. After this reduction phase, we then consider the remaining types which use boolean connectives and could not be reduced. On the example above, consider again field \( f_1 \) of type \text{int or bool}. This will be translated as two fields \text{int}_f_1 and \text{bool}_f_1 with the corresponding types. In order to initialise these fields, we use the constructor overloading capabilities of the JVM to generate an overloaded version for each alternative type. In each constructor, only the field that matches the type of the parameter is initialised with all other fields set to \text{null}. To access the field of an object, we generate code that checks each alternative of the field if it is non-null and includes the rest of the code generation for the expression for each branch.

\[1\] https://kotlinlang.org/
At run time, only one branch will be true, and this is the branch of generated code which will be executed.

For methods, we also use the overloading capabilities of the JVM to define an overloaded method for each type in the expanded method parameter, all with the same method body. Depending on which alternative the argument inhabits at run time, a different method will be dispatched to. Like methods with boolean types, we similarly implement methods with method types, as we discussed in §3.6, by defining an overloaded alternative for each type that implements the specified method. This concludes the code generation phase for all expressions in SFJ given in §2.2.

4 Related Work and Conclusion

Semantic subtyping goes back to more than two decades ago [1][10]. Hosoya and Pierce [11][12][13] define XDuce, an XML-oriented language designed specifically to transform XML documents in other XML documents satisfying certain properties. Frisch et al. [13] extend XDuce by introducing less XML specific types such as records, boolean connectives and arrow types, and implement it as CDuce. Their work is similar to ours in that our class-based semantic type system is a combination of the CDuce record types with arrow types. Castagna et al. define \( \mathbb{C}\pi \) [7], a variant of the asynchronous \( \pi \)-calculus, where channel types are augmented with boolean connectives; semantic subtyping for ML-like languages [8] and semantic subtyping in a gradual typing framework [6]. Ancona and Lagorio [3] define subtyping for infinite types by using union and object type constructors, where types are interpreted as sets of values of the language. Bonsangue et al. [4] study a coalgebraic approach to coinductive types and define a set-theoretic interpretation of coinductive types with union types. Pearce [26] defines semantic subtyping for rewriting rules in the Whiley Rewrite Language and for a flow-typing calculus [25].

Regarding implementations of semantic subtyping, to the best of our knowledge, there are only a few works in the literature. Muehlboeck and Tate [23] define a syntactic framework with boolean connectives which has been implemented in the Ceylon programming language [18]. Ancona and Corradi [2] define semantic subtyping for an imperative object-oriented language with mutable fields. In our framework we are considering only the functional fragment of Java, which is FJ, and as a result the semantic subtyping framework is simpler. The authors also propose a prototype implementation of their subtyping algorithm. Chaudhuri et al. [9] present the design and implementation of Flow, which is a type checker for JavaScript. They use boolean connectives and, not, or for their predicates, however they do not define semantic subtyping for their language.

In this paper we presented the design and implementation of SFJ—Semantic Featherweight Java, an extension of Featherweight Java featuring boolean types, semantic subtyping, and integrating both nominal and structural subtyping. Due to the expressivity of semantic subtyping, in SFJ we are able to restore standard Java constructs and features for example, lists and overloading methods,
which were not present in FJ, thus making SFJ a more complete language. We presented [Algo. 1] on validity of type definitions and [Algo. 2] on semantic subtyping, which finds all nominal and structural subtypes for all types in an SFJ program. We also described typing of terms in SFJ, which follows that of Java and builds upon relevant work [11,12]. As future work, we aim to finalise the code generation phase, which is sketched in § 3.7.

References

1. Aiken, A., Wimmers, E.L.: Type inclusion constraints and type inference. In: Proceedings of the conference on Functional Programming Languages and Computer Architecture, FPCA. pp. 31–41. ACM, New York, NY, USA (1993). https://doi.org/10.1145/165180.165188
2. Ancona, D., Corradi, A.: Semantic subtyping for imperative object-oriented languages. In: Visser, E., Smaaragdakis, Y. (eds.) Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA. pp. 568–587. ACM (2016). https://doi.org/10.1145/2983990.2983992
3. Ancona, D., Lagorio, G.: Coinductive subtyping for abstract compilation of object-oriented languages into horn formulas. In: Proceedings of the Symposium on Games, Automata, Logic, and Formal Verification, GANDALF. EPTCS, vol. 25, pp. 214–230 (2010). https://doi.org/10.4204/EPTCS.25.20
4. Bonsangue, M.M., Rot, J., Ancona, D., de Boer, F.S., Rutten, J.J.M.M.: A coalgebraic foundation for coinductive union types. In: Proceedings of the International Colloquium on Automata, Languages, and Programming, ICALP. LNCS, vol. 8573, pp. 62–73. Springer (2014). https://doi.org/10.1007/978-3-662-43951-7_6
5. Boyland, J., Castagna, G.: Parasitic methods: An implementation of multimethods for java. In: Proceedings of the Conference on Object-Oriented Programming Systems, Languages & Applications OOPSLA. pp. 66–76. ACM (1997). https://doi.org/10.1145/263698.263721
6. Castagna, G., Lanvin, V.: Gradual typing with union and intersection types. Proc. ACM Program. Lang. 1(ICFP), 41:1–41:28 (2017). https://doi.org/10.1145/3110285
7. Castagna, G., Nicola, R.D., Varacca, D.: Semantic subtyping for the pi-calculus. Theor. Comput. Sci. 398(1-3), 217–242 (2008). https://doi.org/10.1016/j.tcs.2008.01.049
8. Castagna, G., Petrucciani, T., Nguyen, K.: Set-theoretic types for polymorphic variants. In: Garrigue, J., Keller, G., Sumii, E. (eds.) Proceedings of the International Conference on Functional Programming, ICFP. pp. 378–391. ACM (2016). https://doi.org/10.1145/2951913.2951928
9. Chaudhuri, A., Vekris, P., Goldman, S., Roch, M., Levi, G.: Fast and precise type checking for javascript. Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 1(OOPSLA), 48:1–48:30 (2017). https://doi.org/10.1145/3133872
10. Damm, F.M.: Subtyping with union types, intersection types and recursive types. In: Proceedings of the International Conference on Theoretical Aspects of Computer Software, TACS. pp. 687–706. Springer-Verlag, London, UK (1994)
11. Dardha, O., Gorla, D., Varacca, D.: Semantic Subtyping for Objects and Classes. In: Proceedings of the International Conference on Formal Techniques for Distributed Systems, FMOODS/FORTE. LNCS, vol. 7892, pp. 66–82. Springer (2013). https://doi.org/10.1007/978-3-642-38592-6_6
12. Dardha, O., Gorla, D., Varacca, D.: Semantic Subtyping for Objects and Classes. Comput. J. **60**(5), 636–656 (2017). [https://doi.org/10.1093/comjnl/bxw080](https://doi.org/10.1093/comjnl/bxw080)

13. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. J. ACM **55**(4), 1–64 (2008). [https://doi.org/10.1145/1391289.1391293](https://doi.org/10.1145/1391289.1391293)

14. Hosoya, H., Pierce, B.C.: Regular expression pattern matching for xml. SIGPLAN Not. **36**(3), 67–80 (2001). [https://doi.org/10.1145/373243.360209](https://doi.org/10.1145/373243.360209)

15. Hosoya, H., Pierce, B.C.: Xduce: A statically typed xml processing language. ACM Trans. Internet Technol. **3**(2), 117–148 (2003). [https://doi.org/10.1145/767193.767195](https://doi.org/10.1145/767193.767195)

16. Hosoya, H., Vouillon, J., Pierce, B.C.: Regular expression types for xml. ACM Trans. Program. Lang. Syst. **27**(1), 46–90 (2005). [https://doi.org/10.1145/1053468.1053470](https://doi.org/10.1145/1053468.1053470)

17. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight java: a minimal core calculus for java and gj. ACM Trans. Program. Lang. Syst. **23**(3), 396–450 (2001). [https://doi.org/10.1145/503502.503505](https://doi.org/10.1145/503502.503505)

18. King, G.: The Ceylon Language Specification, Version 1.3 (2016). [https://ceylon-lang.org/documentation/1.3/spec/](https://ceylon-lang.org/documentation/1.3/spec/)

19. Lindholm, T., Yellin, F., Bracha, G., Buckley, A.: The Java Virtual Machine Specification, Java SE 7 Edition: Java Virt Mach Spec Java _3_. Addison-Wesley (2013)

20. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6), 1811–1841 (1994). [https://doi.org/10.1145/197320.197383](https://doi.org/10.1145/197320.197383)

21. Malayeri, D., Aldrich, J.: Integrating nominal and structural subtyping. In: Proceedings of the European Conference on Object-Oriented Programming, ECOOP. pp. 260–284. Springer-Verlag, Berlin, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-70592-5_12](https://doi.org/10.1007/978-3-540-70592-5_12)

22. Malayeri, D., Aldrich, J.: Is structural subtyping useful? an empirical study. In: Proceedings of the European Symposium on Programming ESOP. pp. 95–111 (2009)

23. Muehlboeck, F., Tate, R.: Empowering union and intersection types with integrated subtyping. Proceedings of the Conference on Object-Oriented Programming Systems, Languages & Applications OOPSLA **2**, 1–29 (2018). [https://doi.org/10.1015/0.3276482](https://doi.org/10.1015/0.3276482)

24. Parr, T.: The definitive ANTLR 4 reference. Pragmatic Bookshelf (2013)

25. Pearce, D.J.: Sound and complete flow typing with unions, intersections and negations. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI. LNCS, vol. 7737, pp. 335–354. Springer (2013). [https://doi.org/10.1007/978-3-642-35873-9_21](https://doi.org/10.1007/978-3-642-35873-9_21)

26. Pearce, D.J.: On declarative rewriting for sound and complete union, intersection and negation types. J. Comput. Lang. **50**, 84–101 (2019). [https://doi.org/10.1016/j.jvlc.2018.10.004](https://doi.org/10.1016/j.jvlc.2018.10.004)

27. Usov, A., Dardha, O.: SFJ: An implementation of Semantic Featherweight Java (2020), On YouTube [https://youtu.be/oTFIjm0A2O8](https://www.dcs.gla.ac.uk/~ornela/publications/SFJ.mp4) and on Dardha’s website [http://www.dcs.gla.ac.uk/~ornela/publications/SFJ.mp4](http://www.dcs.gla.ac.uk/~ornela/publications/SFJ.mp4)