SFJ: An implementation of Semantic Featherweight Java *

Artem Usov and Ornela Dardha[0000−0001−9927−7875]

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.* [9,10] 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) [26], 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 [18] is more mainstream in programming languages and is defined by means of a set of formal deductive subtyping rules. Semantic subtyping [6,5,11] 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 [19,20]. 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.* [9,10] define boolean types—based on set-theoretic operations such as **and**, **not**, **or**—and semantic subtyping for *Featherweight Java* (FJ) [16]. 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.* [10, §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 [4], 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 [26] to illustrate the benefits of boolean types and semantic subtyping developed by Dardha *et al.* [9,10] 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:

$$
\begin{array}{l}
\textbf{class } Polygon\ \{\ldots\} \\
\textbf{class } Triangle \textbf{ extends } Polygon\ \{\ldots\} \\
\textbf{class } Other\_Polygons \textbf{ extends } Polygon\ \{ \\
\quad \textbf{double } diagonal(Other\_Polygons\ shape)\ \{\ldots\} \\
\qquad\qquad\qquad\qquad \ldots \\
\} \\
\textbf{class } Square \textbf{ extends } Other\_Polygons\ \{\ldots\} \\
\textbf{class } Rhombus \textbf{ extends } Other\_Polygons\ \{\ldots\}
\end{array}
$$

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 runtime, 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 [26]: 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, Algorithm 1 in § 3.1 which checks the validity of type definitions, and Algorithm 2 in § 3.2 which generates the semantic subtyping relation. Further, we discuss typing in SFJ in § 3.3; nominal vs. structural subtyping in § 3.5; and method types in § 3.6. We discuss future work, 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.* [9,10], 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 τ is defined by the following grammar:

$$
\begin{array}{ll}
\tau ::= \alpha \mid \mu & \textit{Types} \\
\alpha ::= \mathbf{0} \mid \mathbb{B} \mid [\widetilde{l : \tau}] \mid \alpha \textbf{ and } \alpha \mid \textbf{not } \alpha & \textit{Field types } (\alpha\text{-types}) \\
\mu ::= \alpha \to \alpha \mid \mu \textbf{ and } \mu \mid \textbf{not } \mu & \textit{Method types } (\mu\text{-types})
\end{array}
$$

The α-types are used to type fields and the μ-types are used to type methods. Type $\mathbf{0}$ is the empty type. Type $\mathbb{B}$ denotes the *basic* types, such as integers, booleans, etc. Record types $[\widetilde{l : \tau}]$, where $\widetilde{l}$ is a sequence of disjoint labels, are used to type objects. Arrow types $\alpha \to \alpha$ are used to type methods.

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

2.2 Terms

The syntax of terms is defined by the following grammar and is based on the standard syntax of terms in FJ [16]:

$$
\begin{array}{lrl}
\textit{Class declaration} & L & ::= \textbf{class } C \textbf{ extends } C \ \{\widetilde{\alpha\ a};\ K;\ \widetilde{M}\ \} \\
\textit{Constructor} & K & ::= C\ (\widetilde{\alpha\ x})\ \{\ \textbf{super}(\widetilde{x});\ \widetilde{\textbf{this}.a = x};\} \\
\textit{Method declaration} & M & ::= \alpha\ m\ (\alpha\ x)\ \{\ \textbf{return } e;\} \\
\textit{Expressions} & e & ::= x \mid c \mid e.a \mid e.m(e) \mid \textbf{new } C(\widetilde{e})
\end{array}
$$

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 $(\widetilde{L}, e)$ is a pair of a sequence of class declarations $\widetilde{L}$, 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 [16], 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 type of the class they are defined in.

3 The SFJ Language

3.1 On Valid Type Definitions

Since we want to use types τ in practice in SFJ, we restrict them to *finite trees* whose leaves are basic types $\mathbb{B}$ § 2.2 with no cycles. For example, a recursive type $\alpha = [a : \alpha]$ denotes an infinite program tree $\textbf{new}\ C(\textbf{new}\ C(\cdots))$, 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 [22]. 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 fields typed with only basic types as *resolved* otherwise, as *unresolved*. Using this information, Algorithm 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. At each iteration of the algorithm we can resolve at least one type, until all the types in the SFJ program are resolved, and return $True$. However, if there is at least one type unresolved, meaning there is a cycle in the type definition, we return $False$.

3.2 Building Semantic Subtyping for SFJ

If Algorithm 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 [9,10], 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

Input : *classes*, the set of classes marked *resolved* if their fields contain only basic types, *unresolved* otherwise.
Output : *True* if all classes are valid type definitions, *False* otherwise.

```
begin
    do
        resolutionOccured ⟵ False
        for class that is unresolved in classes do
            resolved ⟵ True
            for field in class that contains a class type do
                if type of field is unresolved then
                    resolved ⟵ False
                end
            end

            if resolved = True then
                class ⟵ resolved
                resolutionOccured ⟵ True
            end
        end
    while resolutionOccured = True

    if not all classes are resolved then
        return False
    else
        return True
    end
end
```

Algorithm 1: Validity Check for Type Definitions

in the set of values of the type. We start with basic types and let $Universe$ be a supertype of all types. The full mapping for basic types is defined in Mapping 3.1.

$$\begin{array}{llll} Double & = \{Double, Float, Int, Short, Byte\} & Float & = \{Float, Short, Byte\} \\ Long & = \{Long, Int, Short, Byte\} & Int & = \{Int, Short, Byte\} \\ Short & = \{Short, Byte\} & Byte & = \{Byte\} \\ Boolean & = \{Boolean\} & Void & = \{Void\} \\ Universe & = \{Double, Float, Long, Int, & & \\ & \quad Short, Byte, Boolean, Void\} & & \end{array} \tag{3.1}$$

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*.

Algorithm 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 Algorithm 1, we are guaranteed that Algorithm 2 will terminate. The semantic subtyping generated by Algorithm 2 is a preorder: it is reflexive and transitive. This is also illustrated by Mapping 3.1.

Some comments on Algorithm 2 follow. In function *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, 30, 42), then we add the current class to the list of *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*, check subtyping for the current *class* being processed and update *relation*, which is a mapping from a type to its subtypes and originally only consists of entries from Mapping 3.1. In function $addClass(class)$ we check if the type $class$ is a subtype of an existing type in $relation$ (lines 15-18), as well as the opposite, meaning if *class* is a supertype of an existing type in $relation$ (line 19). In order to do so *checkSuperSet* checks all fields (lines 28-39) and all methods (lines 40-51) in *class* and compares them with an *existingClass* in *relation*. If a subtyping relation is established, then it is added to $relation$ (line 53). Finally, upon returning from $checkSuperSet$, we also add *class* to its own relation (line 21) to satisfy reflexivity and to $Universe$ (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 § 1 that nominal subtyping is name-based and given by the class hierarchy defined by the programmer, whether structural subtyping is structure-based and given by the set-inclusion of fields and methods. In particular, it is contra-variant with respect to this set-inclusion. Algorithm 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 Algorithm 1 is $\mathcal{O}(n)$, and the complexity of Algorithm 2 is $\mathcal{O}(n^2)$, with n being the size of the input. The reason for a quadratic complexity of Algorithm 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 only require traversing the class hierarchy once, which gives an $\mathcal{O}(n)$ 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.* [9,10] 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.

Input : *classes*, the set of classes in an SFJ program for which we have not yet defined the subtyping relation.
relation, the mapping of types to the set of subtypes, initially being Mapping 3.1.

```
begin
    Function generateRelation(classes: List<Class>):
        unprocessed : List < Class > ⟵ []
        for class in classes do
            if addClass(class) = False then
                unprocessed.add(class)
            end
        end
        if untyped ≠ [] then
            generateRelation(unprocessed)
        end
    end

    Function addClass(class: Class) → boolean:
        for existing class type in relation do
            if checkSuperSet(class, existingClass) = False then
                return False
            end
            checkSuperSet(existingClass, class)
        end
        relation[class].add(class)
        relation[Universe].add(class)
        return True
    end

    Function checkSuperSet(class: Class, other: Class) → boolean:
        flag ⟵ True
        for field in class do
            if field contains type not in relation then
                return False
            end
            if other does not contain field then
                flag ⟵ False
            else
                if other.field.types does not fully contain field.types then
                    flag ⟵ False
                end
            end
        end
        for method in class do
            if method contains type not in relation then
                return False
            end
            if other does not contain method then
                flag ⟵ False
            else
                if other.method.types does not fully contain method.types then
                    flag ⟵ False
                end
            end
        end
        if flag = True then
            relation[other].add
        end
    end
end
```

Algorithm 2: Semantic Subtyping for SFJ Classes

A program $(\tilde{L}, e)$ is well typed if both $\tilde{L}$ 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 **new** $C(\widetilde{e})$ 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 Mapping 3.1 for details).

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

$$\frac{\varGamma \vdash e : \alpha_1 \qquad \alpha_1 \leq \alpha_2}{\varGamma \vdash e : \alpha_2}$$

We read this typing rule as follows: if an expression e is of type α_1 under a typing context $\varGamma$ (details of a typing context are irrelevant here) and type α is a subtype $\leq$ of α_2, then expression e can be typed with α_2.

3.4 Polygons: Continued

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

$$\begin{array}{ll} Polygon \;= \{Polygon, Triangle, Square, Rhombus\} & Triangle = \{Triangle\} \\ Square \;\;\;= \{Square\} & Rhombus = \{Rhombus\} \\ Universe = \{Double, Float, Long, Int, Short, Byte & Diagonal = \{Diagonal\} \\ \qquad\qquad\; Boolean, Void, Polygon, Square & \\ \qquad\qquad\; Square, Rhombus, Diagonal\} & \end{array} \tag{3.2}$$

Recall the method *diagonal* in class *Diagonal*, with signature

$$\textbf{double}\ \textit{diagonal}((\textit{Polygon}\ \textbf{and not}\ \textit{Triangle})\ \textit{shape})$$

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

$$Polygon\ \textbf{and not}\ Triangle = \{Polygon, Square, Rhombus\}$$

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

If we write in our SFJ program the following expression:

$$(\textbf{new } Diagonal()).diagonal(\textbf{new } Square())$$

the argument **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:

$$(\textbf{new } Diagonal()).diagonal(\textbf{new } Triangle())$$

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 [26].

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 definite 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:

```
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. Hence, the integration of both approaches in SFJ makes it possible to overcome these drawbacks, as one can choose on which subtyping to focus 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 [10, §8.4]. We can thus model *multimethods*, [4], 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.* [9,10], consider the following class declarations:

$$
\begin{array}{ll}
\textbf{class}\ A\ \textbf{extends}\ \mathit{Object}\ \{ & \qquad \textbf{class}\ B\ \textbf{extends}\ A\ \{ \\
\quad \textbf{int}\ \mathit{length}\ (\textbf{string}\ s)\{\ \ldots\ \} & \qquad \quad \textbf{int}\ \mathit{length}\ (\textbf{int}\ n)\{\ \ldots\ \} \\
\} & \qquad \}
\end{array}
$$

Method *length* has type **string** $\rightarrow$ **int** in class A. However, because class B extends class A, *length* has type (**string** $\rightarrow$ **int**) **and** (**int** $\rightarrow$ **int**) in class B, which can be simplified to (**string or int**) $\rightarrow$ **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 Example 1.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Diagonal}\ \{ \\
\qquad\qquad \ldots \\
\quad \textbf{double}\ \mathit{diagonal}((\mathit{diagonal} : \textbf{void} \rightarrow \textbf{double})\ \mathit{shape}) \\
\qquad\qquad\qquad\qquad \{\ \textbf{return}\ \ \mathit{shape.diagonal}();\ \} \\
\}
\end{array}
$$

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 ($[\![type_1]\!] \cup [\![type_2]\!] \cup \ldots$), where each $[\![type_i]\!]$ denotes a mapping of $type_i$ to the set of its subtypes, similar to Mapping 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 the use of interfaces, as in one of the proposed Java approaches 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.

4 Conclusion, Related Work and Future Work

Semantic subtyping approach goes back to more than two decades ago [1,8]. Notable lines of work include: Hosoya and Pierce [13,15,14] who define XDuce, an XML-oriented language designed specifically to transform XML documents in other XML documents satisfying certain properties. Castagna *et al.* [6,5,12] extend XDuce with first-class functions and arrow types and implement it as $\mathbb{C}$Duce. The starting point of their framework is a higher-order $\lambda-$calculus with pairs and projections. Muehlboeck and Tate [21] define a syntactic framework with boolean connectives which has been implemented in the Ceylon programming language [17]. Castagna *et al.* [7] define $\mathbb{C}\pi$, a variant of the asynchronous $\pi-$calculus [25], where channel types are augmented with boolean connectives. Ancona and Lagorio [2] define subtyping for infinite types coinductively by using union and object type constructors, where types are interpreted as sets of value of the language. Bonsangue *et al.* [3] study a coalgebraic approach to coinductive types and define a set-theoretic interpretation of coinductive types with union types. Pearce [24] defines semantic subtyping for rewriting rules in the Whiley Rewrite Language and for a flow-typing calculus [23].

To conclude, in this paper we presented the design and implementation of SFJ, an extension of Featherweight Java with boolean types and semantic subtyping, which allow for both structural and nominal subtyping in FJ as well as restore standard Java constructs for example, lists and features for example, overloading which were not present in FJ, making SFJ a more complete language.

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., 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
3. 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
4. Boyland, J., Castagna, G.: Parasitic methods: An implementation of multi-methods 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
5. Castagna, G.: Semantic subtyping: Challenges, perspectives, and open problems. In: ICTCS. pp. 1–20 (2005)
6. Castagna, G., Frisch, A.: A gentle introduction to semantic subtyping. In: Proceedings of the Conference on Principles and Practice of Declarative Programming, PPDP. vol. 2005, pp. 198–208 (2005). https://doi.org/10.1145/1069774.1069793
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. 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)
9. 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
10. 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
11. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. Journal of the ACM **55** (2008). https://doi.org/10.1145/1391289.1391293
12. 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
13. 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
14. 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
15. 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

16. 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
17. King, G.: The Ceylon Language Specification, Version 1.3 (2016), `https://ceylon-lang.org/documentation/1.3/spec/`
18. 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
19. 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
20. Malayeri, D., Aldrich, J.: Is structural subtyping useful? an empirical study. In: Proceedings of the European Symposium on Programming ESOP. pp. 95–111 (2009)
21. 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/10.1145/3276482
22. Parr, T.: The definitive ANTLR 4 reference. Pragmatic Bookshelf (2013)
23. 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
24. 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
25. Sangiorgi, D., Walker, D.: The Pi-Calculus: A Theory of Mobile Processes. Cambridge University Press, New York, NY, USA (2003)
26. Usov, A., Dardha, O.: SFJ: An implementation of Semantic Featherweight Java (2020), On YouTube `https://youtu.be/oTFIjm0A2O8` and on Dardha's website `http://www.dcs.gla.ac.uk/~ornela/publications/SFJ.mp4`