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 and 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 typechecking 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*

* 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).

is 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". Intutively, extending a class with more fields and methods results in a smaller set of inhabitants for that class. For example, the set of objects of a class *Student* is smaller than the set of objects of a class *Person* as each Student is a Person, but not the other way around. Hence, *Student* is a structural subtype of *Person*, even if it is not declared to be 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 and 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:

```
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 {...}
                        ...
```

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 typechecking 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 typechecker 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 returns the length of its longest diagonal; otherwise, if the polygon is a triangle, then there will be a type error at compile time.

Structure of the paper: § 2 details the technical developments behind semantic subtyping and its properties. § 3 details the design and implementation of SFJ. § 4 concludes the paper.

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:

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

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]:

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

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 initializes 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 accesses, method invocations and object creations.

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:

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

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

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 SFJ: Design and Implementation

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 **new** $C($**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 the above types can be defined in Java but as well are not inhabitable.

The type system of SFJ builds upon the type system by Dardha *et al.* [9,10]. It 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 using ANTLR, we can automatically generate a parser for SFJ and extend it in order to implement the required checks for our type system. Running the parser returns an abstract syntax tree (AST) of a program in SFJ.

When visiting the AST of a program, we check if it 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, 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. At each iteration of the algorithm we resolve at least one type, until all the types are resolved, and return $True$. However, if there is at least one type unresolved, that means there is a cycle in the type definition and we return $False$.

Now, we can define the subtyping relation. Building upon the interpretation of types as sets of values to define the semantic subtyping for FJ [9,10], in our implementation of SFJ we keep track of the subtyping relation by defining a map from a type to the set of its subtypes, with the property that the set of

Algorithm 1: ValidityCheck

input : classes marked *resolved*/*unresolved* depending if their fields contain only basic types.
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
```

values of a subtype is included in the set of values of the type. As a first step, we start with basic types.

$Double = \{Double, Float, Int, Short, Byte\}$ $Float = \{Float, Short, Byte\}$
$Long = \{Long, Int, Short, Byte\}$ $Int = \{Int, Short, Byte\}$
$Short = \{Short, Byte\}$ $Byte = \{Byte\}$ $Boolean = \{Boolean\}$

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, however this is not the case for *Int* and *Double*. Similarly, *Long* is not a subtype of *Double*.

Finally, the following algorithm defines the subtyping relation for all class types, which concludes the process.

```
function generateRelation(classes):
    List<class> untyped = []
    for class in classes:
        if addClass(class) is false:
            untyped.add(class)
```

```
    if untyped is not []:
        generateRelation(untyped)

function addClass(class):
    for existing class type in relation:
        if checkSuperSet(class, existingClass) is false:
            return false
        checkSuperSet(existingClass, class)
        add class to its own subtype relation

function checkSuperSet(class, other):
    boolean flag = true
    for field in class:
        if field contains type not in relation:
            return false
        if other does not have field:
            flag = false
        else:
            if other.field.types not fully contains field.types:
                flag = false
    for method in class:
        if method contains type not in relation:
            return false
        if other does not have method:
            flag = false
        else:
            if other.method.types not fully contains method.types:
                flag = false
    if flag == true:
        add class to other subtype relation
```

Let us illustrate the subtyping algorithm with our *Polygons* Example 1. The algorithm generates the following subtyping relation, which also includes the subtyping relation for basic types defined in the first step.

$Polygon = \{Polygon, Triangle, Square, Rhombus\}$ $Triangle = \{Triangle\}$
$Square = \{Square\}$ $Rhombus = \{Rhombus\}$ $Diagonal = \{Diagonal\}$

Recall method *diagonal* in class *Diagonal*, we can see that the result of the set operation on its parameter type gives the following set of polygons:

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

If we pass to *diagonal* an argument of type *Triangle*, as it is not contained in the computed set of subtypes, we get a typechecking error at compile time [26].

The subtyping algorithm finds all nominal and structural subtypes. It finds all nominal subtypes as they inherit all fields and methods of their super class, so are always guaranteed to be a superset. Similarly, given all pairs of types, the algorithm checks if one is a superset of the other, i.e., they share common fields and methods, and if so, they are related by structural subtyping. For example, type $empty = []$, will have all classes as structural subtypes.

Note that both approaches to subtyping have their drawbacks. Consider two structurally equivalent class/record types $coordinate = [x : int, y : int, z : int]$ and $colour = [x : int, y : int, z : int]$. While structurally they can be used interchangeably in our type system, their "meaning" is completely different. On the

other hand, with nominal subtyping in Java one can define an overridden method to perform the opposite logic to what the super class is expecting. Therefore, both approaches leave an expectation on the developer to check what they are doing is correct, hence the integration of both in SFJ is the key to overcoming these drawbacks, as one can choose which approach to use for a given task.

Furthermore, we have extended FJ with method types, demonstrated below to show another possible implementation of the *Polygons* Example 1.

class *Diagonal* {**int** *diagonal*(*(getDiagonal: void→int) shape*){...}}

We instead only implement the *getDiagonal* method on polygons that support it. We can then specify to only accept classes that implement the method *getDiagonal*. If we pass an argument of type *Triangle*, it is not in the computed set of subtypes, so we get a typechecking error at compile time. We do not have to generate a relation for all methods in all classes as this is computationally unnecessary and instead only generate the set of subtypes on encountering a method type.

4 Related Work and Conclusion

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.* [**?**,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`