SFJ: An implementation of Semantic Featherweight Java

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

School of Computing Science, University of Glasgow, United Kingdom
{artem.usov, ornela.dardha}@glasgow.ac.uk

Abstract. Subtyping is a key notion in programming languages, as it allows more flexibility in coding. 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* vs. *structural* subtyping. Dardha *et al.* [9,10] defined boolean types and semantic subtyping for Featherweight Java (FJ) and integrated both structural and nominal subtyping, thus exploiting the benefits of both approaches. However, these benefits were illustrated only at a theoretical level, but not exploited practically.
In this paper, we present SFJ—Semantic Featherweight Java, an implementation of FJ which features boolean types and structural subtyping as well as nominal subtyping. The benefits of SFJ, illustrated in the paper and the video (with audio/subtitles) [21], show how static typechecking of boolean types and semantic subtyping gives higher guarantees of program correctness, more flexibility and compactness of program writing.

1 Introduction

There are two approaches to defining subtyping relations: the *syntactic* and the *semantic* approach. Syntactic subtyping is more mainstream in programming languages and is defined by means of a set of formal deductive subtyping rules. Semantic subtyping is less know: one defines a formal model of the language and an interpretation of types as subsets of this model, ultimately, these will be subsets of values in the language. Then, subtyping is defined as set inclusion of these subsets denoting types. Orthogonally, for object-oriented languages there are two approaches to defining subtyping relations: the *nominal* and the *structural* approach. Nominal subtyping is based on declarations by the developer and is *name*-based: A 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. It is not surprising that the nominal subtyping aligns well with the syntactic approach, and in fact it is more well-known than the structural subtyping, which in turn aligns well with the semantic approach.

Dardha *et al.* [9,10] define boolean types and semantic subtyping for *Featherweight Java* (FJ) [15]. 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 programming. 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* § 2, 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 introduce added complexity, but rather the opposite is true, as the developer has an easier, more compact and more 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 Java, 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 [21] 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 that our class hierarchy is such that *Polygon* is the parent class and all other geometric figure classes extend *Polygon*, which is how one would naturally define the set of polygons. If we think of the class hierarchy as given and 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 e.g., inside a **try-catch** which throws an exception 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 implement this in SFJ [21]: 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.

The technical developments behind semantic subtyping and its properties are complex, however, they are completely transparent to the developer. Due to space limit we will remove most theoretical details from this submission, as they are not necessary to understand the design and implementation of SFJ, which is the focus of this paper. We refer the reader to the relevant work [9,10] and we will include a more complete background on semantic subtyping in a possible full paper.

2 SFJ: Design and Implementation

The syntax of types is given by the following grammar [9,10]:

$\tau ::= \alpha \mid \mu$	*Type term*
$\alpha ::= \mathbf{0} \mid \mathbb{B} \mid \widetilde{[l : \tau]} \mid \alpha$ **and** $\alpha \mid$ **not** α	*Object type* (α-type)
$\mu ::= \alpha \to \alpha \mid \mu$ **and** $\mu \mid$ **not** μ	*Method type* (μ-type)

α-types are used to type fields and 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 types using **and** and **not** have their expected set-theoretic meanings, and **or** is obtained by their combination.
The syntax of terms is given by the following grammar and is based on the standard syntax of terms in FJ [15,9,10]:

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

Since we want to use types τ in practice in SFJ, we restrict them to finite trees whose leaves are constants with no cycles. For e.g., a recursive type $\alpha = [a : \alpha]$ denotes an infinite program tree **new** $C($**new** $C(\cdots))$, hence we avoid it as it is inhabitable. When processing the abstract syntax tree (AST) of a program in SFJ, we forbid fields to have the same type as the type of the class they are defined in. Secondly, we mark any classes containing fields typed with only basic types as *resolved* otherwise, as *unresolved*. This is captured by the following algorithm, which checks if the type definitions in the program are valid.

```
do
    boolean resolutionOccured = false
    for class that is unresolved:
        boolean resolved = true
        for field in class that contains a class type:
            if class type is not resolved
                resolved = false
        if resolved is true:
            set class to be resolved
            resolutionOccured = true
while resolutionOccured is true

if all classes are not resolved:
    program contains invalid type definition
```

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

$$\begin{array}{llll} Double = \{Double, Float, Int, Short, Byte\} & & Float = \{Float, Short, Byte\} \\ Long = \{Long, Int, Short, Byte\} & & Int = \{Int, Short, Byte\} \\ Short = \{Short, Byte\} & Byte = \{Byte\} & Boolean = \{Boolean\} \end{array}$$

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 [21].

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 e.g., 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.

$$\textbf{class}\ \mathit{Diagonal}\ \{\textbf{int}\ \mathit{diagonal}(\mathit{(getDiagonal:\ void}{\rightarrow}\mathit{int)}\ \mathit{shape})\{\ldots\}\}$$

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.

3 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 [12,14,13] who define XDuce, an XML-oriented language designed specifically to transform XML documents in other XML documents satisfying certain properties. Castagna *et al.* [6,5,11] 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 [17] define a syntactic framework with boolean connectives which has been implemented in the Ceylon programming language [16]. Castagna *et al.* [7] define $\mathbb{C}\pi$, a variant of the asynchronous $\pi-$calculus [20], 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 [19] defines semantic subtyping for rewriting rules in the Whiley Rewrite Language and for a flow-typing calculus [18].

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 e.g., lists and features for e.g., 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: FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture. 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: Montanari, A., Napoli, M., Parente, M. (eds.) Proceedings First Symposium on Games, Automata, Logic, and Formal Verification, GANDALF 2010, Minori (Amalfi Coast), Italy, 17-18th June 2010. EPTCS, vol. 25, pp. 214–230 (2010). https://doi.org/10.4204/EPTCS.25.20, 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: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II. Lecture Notes in Computer Science, vol. 8573, pp. 62–73. Springer (2014). https://doi.org/10.1007/978-3-662-43951-7_6, 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: Loomis, M.E.S., Bloom, T., Berman, A.M. (eds.) Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA). pp. 66–76. ACM (1997). https://doi.org/10.1145/263698.263721, 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: PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming. pp. 198–199. ACM, New York, NY, USA (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: TACS '94: Proceedings of the International Conference on Theoretical Aspects of Computer Software. pp. 687–706. Springer-Verlag, London, UK (1994)
9. Dardha, O., Gorla, D., Varacca, D.: Semantic subtyping for objects and classes. In: Beyer, D., Boreale, M. (eds.) Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOODS/FORTE. LNCS, vol. 7892, pp. 66–82. Springer (2013). https://doi.org/10.1007/978-3-642-38592-6_6, 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, 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. J. ACM **55**(4), 1–64 (2008). https://doi.org/10.1145/1391289.1391293
12. 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

13. 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
14. 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
15. 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
16. King, G.: The Ceylon Language Specification, Version 1.3 (2016), `https://ceylon-lang.org/documentation/1.3/spec/`
17. Muehlboeck, F., Tate, R.: Empowering union and intersection types with integrated subtyping. Proceedings of the ACM on Programming Languages **2**(OOPSLA), 1–29 (2018). https://doi.org/10.1145/3276482
18. Pearce, D.J.: Sound and complete flow typing with unions, intersections and negations. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings. Lecture Notes in Computer Science, 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`
19. 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`
20. Sangiorgi, D., Walker, D.: The Pi-Calculus: A Theory of Mobile Processes. Cambridge University Press, New York, NY, USA (2003)
21. 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`