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 the latter, one starts by defining a model of the language and an interpretation of types as subsets of this model. Subtyping is defined as *set inclusion of types* denoting sets. An orthogonal issue, typical of object-oriented languages, is the *nominal* vs. *structural* subtyping. Dardha *et al.* [9,10] combined structural subtyping and boolean types, on one hand, with nominal subtyping, on the other, in Featherweight Java (FJ), by 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 with features structural subtyping and boolean types as well as nominal subtyping. The benefits of SFJ, also illustrated in the accompanied video, show how static type checking 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. The syntactic subtyping is more mainstream in programming languages and is defined by means of a set of formal deductive subtyping rules. The semantic subtyping is less know and is defined as inclusion of sets denoting types. Starting from a formal model of the language, we define an interpretation of types as subsets of this model, typically, subsets of values in the language. Then, subtyping is defined as inclusion of these subsets.

Orthogonally, for object-oriented languages there are two approaches to defining subtyping relations: the *nominal* and the *structural* approach. The 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. The 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. At this point it is not surprising that the nominal subtyping aligns well with the

syntactic approach and in fact is more well-known than the structural subtyping which in turn aligns well with the semantic approach.

Dardha *et al.* [9,10] define semantic subtyping for *Featherweight Java* (FJ) [15] with *boolean types*. This approach allows the co-existence of 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 integrated 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**. 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], we can easily encode in SFJ standard programming constructs and features of Java, such as lists, or overloading classes (via multimethods, see below), which are missing in FJ, making SFJ a more complete and closer to Java programming language.

Example 1 (Polygons). This will be our running example both in the paper and in the tool video [19] 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, rectangles, rumbles 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 {
    real diagonal(Other_Polygons p) {...}
                        ...
}
class Rectangle extends Other_Polygons {...}
class Rhombus extends Other_Polygons {...}
```

Or by means of an interface *Diagonal*:

```
public interface Diagonal {
    real diagonal(Polygon p);
}
class Polygon {...}
class Triangle extends Polygon {...}
class Rectangle 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** that throws an exception 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 typechecking is required and implement this in SFJ [19]: 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* **extends** *Object* {...}
class *Triangle* **extends** *Polygon* {...}
class *Rectangle* **extends** *Polygon* {...}
class *Rhombus* **extends** *Polygon* {...}
...
class *Diagonal* **extends** *Object* {
  real *diagonal*(*Polygon* **and not** *Triangle p*){...}
}

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.

On multimethods Since FJ is a core language, some features of the full Java are removed: overloading methods is one of them. In our framework, leveraging the expressivity of boolean connectives and semantic subtyping, many of these features, and language constructs, e.g., lists, are restored: overloading methods is one of them. 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 Dardha *et al.* [9,10], consider the following class declarations:

class *A* **extends** *Object* {
  int *length* (**string** *s*){ ... }
}

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

Method *length* has type **string** → **int** in *A*. However, in *B* it has type (**string** → **int**) ∧ (**int** → **int**), which can be simplified as (**string** ∨ **int**) → **int**.

The technical developments behind semantic subtyping and its properties are not easy, however, they are completely transparent to the developer. Due to space limit we will remove 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 the background on semantic subtyping in a possible full paper.

2 SFJ: Design and Implementation

As already discussed in the introduction, the key idea behind semantic subtyping is interpreting types as sets, in particular, as sets of values of the language at hand. Since we want these types to be able to be finitely representable, we restrict our types to finite trees whose leaves are constants with no cycles. For example a recursive type $\alpha = [a : \alpha]$ would be an infinite tree **new** $C($**new** $C(\cdots))$. Similarly the types $\alpha = [b : \beta]$, $\beta = [a = \alpha]$ would also be impossible to instantiate.

Therefore, when processing the abstract syntax tree (AST) of a program in SFJ, we do not allow the type of the class we are defining to be a type of its own field. Secondly we mark any classes with only basic types *(bool, int, float ...)* for the types of its fields as *resolved* and otherwise, if it contains a class type, we mark it as *unresolved*. After processing the whole AST, we perform the following algorithm to decide whether the type definitions in the program are valid.

```
do
    boolean resolutionOccured = false
    for class that is unresolved:
        boolean resolved = true
        for field 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
```

If the types in the program are finite trees finite trees whose leaves are constants with no cycles, then at each iteration of the algorithm, either all the types are going to be resolved, or we are going to be able to resolve at least one type. If we do not resolve at least one type and not all types are resolved, we know we have encountered a cycle in the type definition.

Given that we now know that all the types in the program are valid, we can create the subtyping relation which will be a map of types to a set of its subtypes. For every program, we assume an initial subtyping relation for all of our basic types. We would like to highlight that *Int* is not a subtype of *Float* as it cannot represent the whole domain of *Int* accurately and therefore the domain is not fully contained, however this is not the case for *Double* and *Int*. Similarly

so *Long* is not a subtype of *Double*.

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

Knowing that all our types are finite trees with leaves as constants and given this initial relation, we can create a subtyping relation for all class types using the following algorithm.

```
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
```

Continuing with the *Polygons* example, the algorithm would generate the following relation, including all base relations we defined before.

$$Polygon = \{Polygon, Triangle, Rectangle, Rumble\}$$
$$Triangle = \{Triangle\}\ Rectangle = \{Rectangle\}\ Diagonal = \{Diagonal\}$$

Given method *diagonal* from the example, we can see that the result of the set operation on its parameter type would be

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

Therefore if we try to pass an argument of type *Triangle* to this method, the type if not fully contained within the computed set, so it does not type check.

The above subtyping algorithm will find all nominal subtypes, as nominal subtypes inherit all fields and methods of their super class, so are always guaranteed to be a superset. Similarly, we can check every pair of types if one is a superset of the other, i.e. they share common fields and methods, and if so, we know it is a structural subtype of the other. For example, the Empty type, $empty = []$, will have all classes as structural subtypes. The above algorithm will without differentiating, find all nominal and structural subtypes as we always consider all other classes in the program to be potential subtype candidates.

The argument against structural subtyping comes up here, as a type $coord = [x : int, y : int, z : int]$ will be structurally similar to a type $colour = [x : int, y : int, z : int]$. While they can be used interchangeable in our type system, the meaning of the types are completely different so it means we ignore possible wrong behaviour in our program. We however argue that it is no more unsafe than nominal subtyping in Java where an overridden method can be defined to perform the opposite logic to what the super class expects. Therefore both methods leave an expectation on the programmer to check what they are doing is correct.

Furthermore, we have extended FJ with method types, demonstrated below to show another possible implementation of the Polygons example, where we only implement the *diagonal* method on polygons that support it, and not in the Triangle class.

class *Diagonal* **extends** *Object* {**int** *diagonal*(*(getDiagonal: void→int) p*){...}}

While we do not generate a relation for each method in all classes as this is computationally unnecessary, we can on encountering a method type generate the set of classes which implement this method and combine their subtype relations using the *or* operator. We can then use the same set inclusion to see if the argument passed to this function type checks.

In the implementation of SFJ, we are liberal in the use of fields and method which might not be implemented for all types in the subtyping relation rather than restricting only it to those which we know are shared by all types. Improper use of this allows for run-time exceptions however it more closely follows the idea of giving the programmer tools to create more expressive programs at the cost of checking program correctness such as with the use of structural subtypes.

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. Castagna *et al.* [7] define $\mathbb{C}\pi$, a variant of the asynchronous $\pi-$calculus [18], 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 [17] defines semantic subtyping for rewriting rules in the Whiley Rewrite Language and for a flow-typing calculus [16].

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 combined 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.

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/http://doi.acm.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/http://doi.acm.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/http://dx.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/http://doi.acm.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/http://doi.acm.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/http://doi.acm.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/http://doi.acm.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/http://doi.acm.org/10.1145/503502.503505
16. 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
17. 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
18. Sangiorgi, D., Walker, D.: The Pi-Calculus: A Theory of Mobile Processes. Cambridge University Press, New York, NY, USA (2003)
19. Usov, A., Dardha, O.: SFJ: An implementation of Semantic Featherweight Java