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 programming language at hand and an interpretation of its types as subsets of such model. Then, subtyping is defined as *set inclusion of types* denoting sets. An orthogonal issue, typical of object-oriented languages, is the *nominal* and *structural* subtyping. Dardha *et al.* [7,8] 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 only presented and exemplified at a theoretical level and not exploited practically. At last, in this work we have implemented SFJ—Semantic Featherweight Java, which features structural subtyping and boolean types as well as nominal subtyping, typical of FJ itself. The benefits of SFJ, also illustrated in the accompanied video, 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. In the latter, one starts by defining a model of the programming language at hand and an interpretation of its types as subsets of such model. The syntactic subtyping is more mainstream in programming languages design and implementation, and the relation is given by means of a set of formal deductive subtyping rules. The semantic subtyping is less know and the relation is given by inclusion of sets denoting types. One starts from a formal model of the programming language at hand and defines an interpretation, or semantics,

* Research 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 "BehAPI: Behavioural Application Program Interfaces".

of types as subsets of this model; subtyping naturally is defined as inclusion of these sets, hence the semantic adjective attached to it. Orthogonally, especially for object-oriented languages, there are two approaches to defining subtyping relation: the *nominal* and the *structural* approach. The nominal subtyping relation is based on *declarations* by the developer and is *name*-based, hance the adjective nominal: 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 relation 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 hence is more well-known than the structural subtyping which aligns well with the semantic approach.

Dardha *et al.* [7,8] define boolean types and semantic subtyping for *Featherweight Java* (FJ) [13], which is the functional fragment of the Java language. This approach to defining subtyping naturally allows for structural subtyping in FJ. On the other hand, nominal subtyping, typical of FJ, can be obtained in the framework exploiting boolean types and semantic subtyping, as shown in the extensive case studies in Dardha *et al.* [8, §8.4].

Example 1 (The Polygons). We will use the *Polygons* as our running example, to illustrate the benefits of the approach by Dardha *et al.* [7,8], both in the paper and in the tool video. 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 Rumble extends Other_Polygons {···}
```

Or by means of an interface:

```
public interface Diagonal {
    real diagonal(Polygon p);
}
class Polygon {···}
class Triangle extends Polygon {···}
class Rectangle extends Polygon  implements Diagonal {···}
class Rumble 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 exactly 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 the natural way to solve this in Java is by defining the method *diagonal* in the class *Polygon* and using an **instanceof**, for example inside a **try-catch** that throws an exception at run time, if the argument passed to the method is a triangle.

The approach proposed by Dardha *et al.* [7,8] allows for the class hierarchy to be defined in such a natural way without needing to change it, but on the other hand it does not require handling exceptions at run time, rather just static typechecking. To achieve this, the framework combines boolean types with semantic subtyping: 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* $\{\cdots\}$
class *Triangle* **extends** *Polygon* $\{\cdots\}$
class *Rectangle* **extends** *Polygon* $\{\cdots\}$
class *Rumble* **extends** *Polygon* $\{\cdots\}$
$\cdots$
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.

The combination of boolean types and semantic subtyping gives a more elegant solution, as illustrate by the *Polygons*—of course a toy example used only as a proof of concept of the approach—however, this approach is not yet practical as there are no boolean types or semantic subtyping in Java, hence its benefits unfortunately remained theoretical.

After a few years, finally we can now exploit the benefits of this approach at a practical level: in this paper we present SFJ—Semantic Featherweight Java § 2, an implementation of Featherweight Java with integrated boolean types and semantic subtyping. In SFJ the programmer has a larger and more expressive set of types, by using **and**, **not**, **or** boolean connectives. This added expressivity does not however introduce added complexity, but rather the opposite is true as the programmer has an easier, more compact and more elegant way of programming. Moreover, as demonstrated in Dardha *et al.* [8, §8.4], the added expressivity allows for standard programming constructs and features in Java, for e.g., lists, or overloading classes (via multimethods below), to be easily encoded in our framework. Another benefit of semantic subtyping is that it can allow a natural integration of structural and nominal subtyping which the programmer has the option to choose, making SFJ a conservative extension of FJ.

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*, [2], 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.* [7,8], 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** $\to$ **int** in A. However, in B it has type (**string** $\to$ **int**) $\wedge$ (**int** $\to$ **int**), which can be simplified as (**string** $\vee$ **int**) $\to$ **int**.

The technical developments behind semantic subtyping and its properties are not easy, however, they are completely transparent to the programmer. Due to space limit we will remove theoretical details from this submission,as they are not necessary to understand the implementation of SFJ which is the focus of this paper. We refer the reader to the relevant work [7,8] and will include the background on semantic subtyping in a possible full paper.

2 SFJ: Design and Implementation

Since we want our types to represent sets of values, 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, to not allow this, when processing the AST of a program, we do not allow the type of the class we are defining to be a field and we mark any classes with only basic types in it's 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.

```
boolean resolutionOccured = false
do
    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
```

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 the 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 therefore the domain is not fully contained in Float. Similarly so for Double and Long.

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

Knowing that all our types are finite trees with leaves as constants and given this initial relation, we can now confidently 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 tryAddSubtype(class, existingClass) is false:
            return false

        tryAddSubtype(existingClass, class)

        add class to its own subtype relation
```

```
function tryAddSubtype(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
```

The following bibliography provides a sample reference list with entries for journal articles [8].

3 Related Work and Conclusion

The first use of the semantic approach goes back to 15 years ago, as in [1,6]. Hosoya and Pierce have also adopted this approach in [10,12,11] to define XDuce, an XML-oriented language designed specifically to transform XML documents in other XML documents satisfying certain properties. The values of this language are fragments of XML documents; types are interpreted as sets of documents, more precisely as sets of values. The subtyping relation is established as inclusion of these sets. The type-system contains boolean types, product types and recursive types. There are no function types and no functions in the language.

Castagna et al. in [4,3,9] extend the XDuce language with first-class functions and arrow types and define and implement a higher-order language named $\mathbb{C}$Duce adopting the semantic approach to subtyping. The starting point of their framework is a higher-order $\lambda-$calculus with pairs and projections. The set of types is extended with intersection, union and negation types interpreted in a set-theoretic way.

The semantic approach can also be applied to the $\pi-$calculus [14,15]. Castagna, Varacca and De Nicola in [5] have used this technique to define the $\mathbb{C}\pi$ language, a variant of the asynchronous $\pi-$calculus, where channel types are augmented with boolean connectives interpreted in an obvious way.

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. 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`
3. Castagna, G.: Semantic subtyping: Challenges, perspectives, and open problems. In: ICTCS. pp. 1–20 (2005)
4. 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
5. 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
6. 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)
7. 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`
8. 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`
9. 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
10. 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
11. 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
12. 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
13. 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
14. Milner, R.: Communicating and mobile systems: the $\pi-$calculus. Cambridge University Press, New York, NY, USA (1999)

15. Sangiorgi, D., Walker, D.: The Pi-Calculus: A Theory of Mobile Processes. Cambridge University Press, New York, NY, USA (2003)