# Towards a Java Subtyping Operad*

Moez A. AbdelGawad
Informatics Research Institute, SRTA-City, Alexandria, Egypt
moez@cs.rice.edu

July 18, 2021

## Abstract

The subtyping relation in Java exhibits self-similarity. The self-similarity in Java subtyping is interesting and intricate due to the existence of wildcard types and, accordingly, the existence of three subtyping rules for generic types: covariant subtyping, contravariant subtyping and invariant subtyping. Supporting bounded type variables also adds to the complexity of the subtyping relation in Java and in other generic nominally-typed OO languages such as C# and Scala.

In this paper we explore defining an operad to model the construction of the subtyping relation in Java and in similar generic nominally-typed OO programming languages. Operads, from category theory, are frequently used to model self-similar phenomena. The Java subtyping operad, we hope, will shed more light on understanding the type systems of generic nominally-typed OO languages.

## 1 Introduction

The addition of generics to Java [24, 25] made the subtying relation in Java more intricate, particularly after adding wildcard types [49] that can be passed as type parameters to generic types.

Wildcard types in Java express so-called *usage-site* variance annotations. Due to supporting wildcard types, the subtyping relation between generic types in Java is governed by three rules: covariant subtyping, contravariant subtyping, and invariant subtyping. Covariant subtyping causes different generic types parameterized with types to be in the *same subtyping relation* with respect to each other as their type parameters are, using the wildcard parameter '`? extends Type`'. Contravariant subtyping causes different generic types parameterized with types to be in the *opposite subtyping relation* with respect to each other as their type parameters are (*i.e.*, generic types parameterized by *subtype* type parameters become *supertypes*), using the wildcard parameter '`? super Type`'. Invariant subtyping causes different generic types parameterized with types to be in *no subtyping relation* with respect to each other, regardless of the subtyping relation between their type parameters, using the type parameter '`Type`' (*i.e.*, in Java, invariant subtyping is the default subtyping rule between generic types when there is no '`? extends`' and '`? super`' annotations).

Further adding to the intricacy of the subtyping relation is the fact that type variables of a generic class[1] can have upper bounds, restricting the set of types that can be passed as type parameters in instantiations of the generic class. We explore in more detail the implications of the generic subtyping rules on the subtyping relation in Java in Section 2.1.

The subtyping relation in other industrial-strength generic nominally-typed OOP languages such as C# [1] and Scala [34] exhibits similar intricacy. C# and Scala support another kind of variance an-

*This paper has been accepted for publication at FTfJP'17 [6].

[1] In this paper Java interfaces are treated as similar to abstract classes.

notations (called *declaration-site* variance annotations) as part of their support of generic OOP. Both kinds—*i.e.*, usage-site and declaration-site variance annotations—have the same self-similarity effect on the generic subtyping relation in nominally-typed OOP languages.

As we describe in Section 5 in more detail, the introduction of wildcard types (and variance annotations, more generally) in mainstream OOP, even though motivated by earlier research, has generated much additional interest in researching generics and in having a good understanding of variance annotations in particular. In this paper we augment this research by presenting an operad for modeling the subtyping relation in Java, exhibiting and making explicit in our operad the self-similarity in the definition and construction of the relation, with the expectation that our operad will apply equally well to subtyping in other OO languages such as C# and Scala.

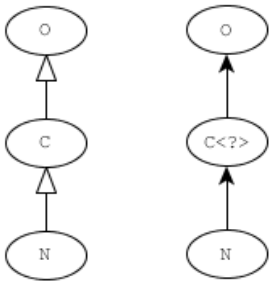
This paper is structured as follows. In Section 2 we present a demonstration of the intricacy and self-similarity of the subtyping relation in Java, followed by a brief introduction to operads. Then, in Section 3, we present $\mathcal{JSO}$, our operad for modeling the subtyping relation in Java and other generic nominally-typed OOP languages. In Section 4 and Appendix A we present examples of the application of $\mathcal{JSO}$ that demonstrate how it works. In Section 5 we discuss some research that is related to ours. We conclude in Section 6 by discussing some conclusions we made and discussing some research that can be built on top of this work.

# 2 Background

In this section we give a simple example of how the subtyping relation in a Java program can be constructed iteratively, based on the type and subtype declarations in the program. We follow that by a brief introduction to operads.

## 2.1 Subtyping in Java

To explore the intricacy of the subtyping relation in Java, let's consider a simple example—probably the most basic example—of a generic class declaration.




(a) Subclassing (b) Subtyping between rank 0 types

Figure 2.1

Assuming we have no classes or types declared other than class `Object` (whose name we later abbreviate to `O`), with a corresponding type that has the same name, then the generic class declaration

```
class C<T> extends Object {}
```

results in a *subclassing* relation in which class `C` is a subclass of `O`.

For subtyping purposes in Java, it is useful to also assume the existence of a special class `Null` (whose name we later abbreviate to `N`) that is a subclass of all classes in the program, and whose corresponding type (that is sometimes called `NullType` [24, 25][2], but which we also call the type `Null` for consistency), hence, is a subtype of all Java object types (*a.k.a.*, reference types). Accordingly, the full subclassing relation (based on the earlier declaration of class `C`) looks as in Figure 2.1a.

We now informally describe how the generic subtyping relation in our Java program can be constructed iteratively, based on the mentioned assumptions and the declaration of generic class `C`. For simplicity, we further assume that a generic class takes

[2] As of Java 8.0, the `Null` type is inexpressible in Java. It is needed, however, during type checking, particularly when checking the types of some expressions that involve polymorphic method type inference and wildcard types [24, 25].

only one type parameter, and that type variables of all generic classes have type `O` (*i.e.*, `Object`) as their (upper) bound.

Given that we have at least one generic class, namely `C`, to construct the generic subtyping relation we should note first that the relation will have an *infinite* number of types, since generic types can be arbitrarily nested. As such, to construct the infinite subtyping relation we go in iterations, where we start with a finite first approximation to the relation then, after each iteration, we get a step closer to the full relation.

The input to the first iteration of the construction process will be the subtyping relation between a finite set of initial types (*i.e.*, ones having rank 0) that are all defined directly using the subclassing relation, then the new types that get constructed in the first iteration are of rank 1, and they get fed as part of the input subtyping relation to the second iteration of the process, and so on.

To demonstrate the workings of this iterative process in more detail, we use the class `C` declaration above and assume, momentarily, the existence of the covariant subtyping rule only (*i.e.*, allow only type parameters of the form '`? extends Type`'). Later on we show how types constructed for the two other generic subtyping rules (contravariant, using '`? super Type`' type parameters, and invariant, using '`Type`' type parameters) and the resulting subtyping relations get incorporated in the construction process.

Types of rank 0—the input to the first iteration of the construction process—are defined immediately, using the subclassing relation, where each non-generic class has a type corresponding to the non-generic class with the same name as the class, while every generic class (only class `C` in our example) gets passed the default '`?`' type parameter to define its corresponding rank 0 type. The subtyping relation between rank 0 types will always be exactly *the same* as the subclassing relation between the classes used to define the types. As such, the diagram for the initial subtyping relation will look very similar to that of the subclassing relation, except that it is for the subtyping relation between object types rather than the subclassing relation between classes. (See Figure 2.1b.)

To construct the second, more accurate version of the full subtyping relation, we use the generic classes and the available subtyping rules (only covariant subtyping, for the moment) to construct new types. In relation to each other, these new types (of rank 1) will be in the new version of the subtyping relation based on the available subtyping rule(s).

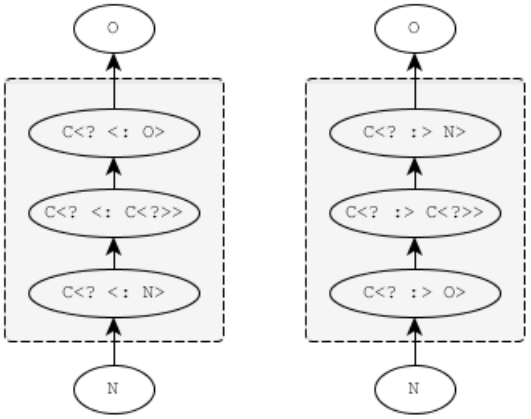
More concretely, using the generic class `C` and the covariant subtyping rule, in the first iteration of the construction process the rank 1 types

`C<? <: O>` and `C<? <: C<?>>` and `C<? <: N>`

(where we abbreviate `extends` to '`<:`'; we later also abbreviate `super` to '`:>`') are constructed, and these three types will have the *same* subtyping relation as that between their non-annotated type parameters (namely, `O`, `C<?>`, and `N`, respectively) in the first version of the subtyping relation. As such, the subtyping relation resulting from the first iteration of the construction process will be as in Figure 2.2a.

In order to appreciate the intricacy of the generic subtyping relation in Java, we should now—before things get more complex—notice the self-similarity that is getting evident in the relation, where the subtyping relation between types inside the dotted part of the diagram is the same as the relation between types in the input relation (the initial subtyping relation, in Figure 2.1b). As we demonstrate shortly, contravariant subtyping results in a "flipped" relation (*i.e.*, an opposite ordering relation) and invariant subtyping results in a "flattened" relation (*i.e.*, a discrete ordering relation). This observation will get reinforced, but will also get somewhat more blurred, as the construction process proceeds further. (If we have *only* the contravariant subtyping rule, the resulting subtyping relation will be as in Figure 2.2b, and if we have *only* the invariant subtyping rule, the resulting subtyping relation will be as in Figure 2.3.)

To construct the full second version of the subtyping relation, resulting from the first iteration of the construction of the relation *with all three* subtyping rules, the construction process merges the three earlier constructed relations by first identifying some types in them (*e.g.*, types `O` and `N`, and noting that for generic types the type parameter '`? :> N`' is the same as '`? <: O`', and '`? :> O`' is the same as '`O`') and




(a) Subtyping (between types of rank 0 and 1) due to *covariant* subtyping

(b) Subtyping (between types of rank 0 and 1) due to *contravariant* subtyping

Figure 2.2

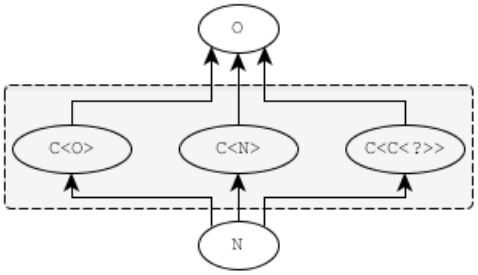



Figure 2.3: Subtyping (between types of rank 0 and 1) due to invariant subtyping

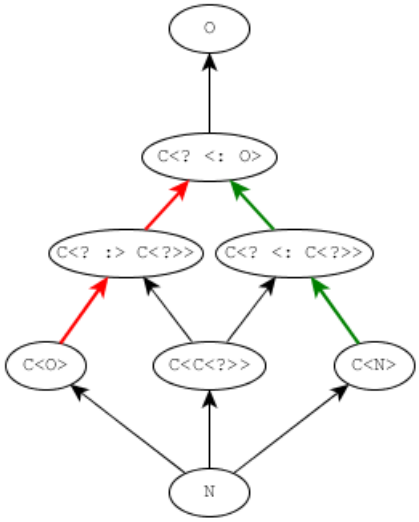



Figure 2.4: Subtyping between types of rank 0 and 1

consistently merging and defining all the subtype relations between the identified types and other types in the subtyping relations. Thus, with all three subtyping rules, the relation in Figure 2.4, as the proper merging of the relations in Figure 2.2a, Figure 2.2b and Figure 2.3, presents the second, more accurate version of the full subtyping relation.

To construct the full and most accurate version of the subtyping relation (*i.e.*, the whole infinite relation), it should now be clear that the process described is continued *ad infinitum*. The purpose of the Java subtyping operad we present in this paper is to formally model each iteration in this construction process, based on the intuitions we presented in the example above. We expand on this example in Section 4, after we present our subtyping operad in Section 3.

## 2.2 Operads

Operads, from category theory, are used frequently to model self-similar phenomena.[3] Informally, an op-

[3]In some category theory literature the notion of operads we use is called a 'colored operad' or a 'symmetric multicategory,' and the notion is strongly related to the notion of a 'monoidal

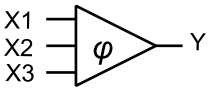



Figure 2.5: An Operad Morphism

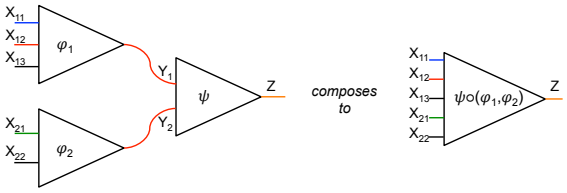



Figure 2.6: Operad Composition ((c) 2014 David Spivak)

erad is 'a category whose morphisms are *multiple-input* single-output morphisms'. Thus, operads, as generalizations of categories, embody in particular a generalization of the familiar notion of (single-input single-output) morphisms. As such, a morphism of an operad is usually depicted as in Figure 2.5.

An operad, however, has some features that make it particularly suited to model self-similarity [45], particularly the composition formula for its generalized notion of morphisms. Similar to regular categorical arrow/morphism composition, morphisms in operads can be composed to define new morphisms. Composition of operad morphisms is the source of most of the power of operads.

The idea behind composition in operads is straightforward (see Figure 2.6). The formal definition of operadic composition and of its associativity requirement, however, are a bit more complicated than for a category, since they involve much more variable indexing, so we do not present these here, depending instead on the intuitiveness and the straightforwardness of the idea behind them (the interested reader should consult [45, Sec. 7.4]). More details about operads, including plenty of operad examples, can be found in [45] and [33].

category'. In this paper we adopt the naming convention used by Spivak [45] and others, for reasons similar to theirs.

# 3 $\mathcal{JSO}$, A Simple Java Subtyping Operad

The Java subtyping operad, which we call $\mathcal{JSO}$, is somewhat similar to and has been inspired by the 'wiring diagrams' operad Spivak presents in [44, 45]. The objects of $\mathcal{JSO}$ are subtyping relations. Its morphisms include four relation transformation morphisms named *copy*, $flip$, $flat$, and *merge* (which we describe shortly) corresponding to the application of the three generic subtyping rules and the merging step we described in Section 2.1, an *identity* transformation (as part of the requirements for defining an operad), and the compositions of all composable (*i.e.*, composition-compatible) combinations of these transformations.

For each generic class `C`, in iteration $n$ of the construction process the *copy* morphism constructs types `C<? <: Type>` (of rank $n+1$) for each type `Type` of the (rank $n$ and lower) types in the input subtyping relation, and it defines the subtyping relation between these types the same as between the corresponding types in the input relation (hence the name *copy*). The *copy* morphism thus models covariant subtyping.

Similarly, for each generic class `C`, the $flip$ morphism constructs types `C<? :> Type>` for each input type, but reverses the input subtyping relation between these (effecting an ordering between the new types opposite to that between corresponding input types; hence the name $flip$) to model contravariant subtyping. The $flat$ morphism similarly constructs for each generic class `C` the types `C<Type>` with no subtyping relation between these new types (other than the trivial ordering due to reflexivity, effecting a discrete ordering between the new types; hence the name $flatten$, which we shorten to $flat$).

To produce their output subtyping relations, the three transformations then, for each generic class `C`, "embed" the new subtyping relation they defined between types constructed using `C` and the input relation, where the relation gets embedded in place of type `C<?>` in the initial subtyping relation (this embedding of a relation inside another relation can be precisely defined using operadic terms similar to ones

in the 'wiring diagrams' operad of Spivak [44, 45]).

The *merge* transformation is a little different than the three other main transformations. The three subtyping relations that are input to the *merge* morphism will be the output relations of the three morphisms *copy*, $flip$, and $flat$, as shown in Figure 3.1, *i.e.*, *merge* will be composed (operadically) with *copy*, $flip$, and $flat$. The *merge* morphism does not construct any new types but rather identifies any repeated new types in its three inputs while keeping their subtyping relations with other types (*i.e.*, *merge* performs a pushout/fibered coproduct of its input relations, quotienting over identified types and relations between them, effecting a "union" of its inputs). The *merge* morphism thus merges the three input subtyping relations into one output subtyping relation (hence the name *merge*).

For example, on input the relation in Figure 2.1b, the *copy* morphism will have the relation in Figure 2.2a as its output. Similarly, the morphisms $flip$ and $flat$ will have the relations in Figure 2.2b and Figure 2.3 as their output, respectively. On input the three relations in Figures 2.2a, 2.2b and 2.3, the *merge* morphism will have the relation in Figure 2.4 as its output. We further demonstrate $\mathcal{JSO}$ in Section 4.

Using the fact that subtyping relations can be viewed as DAGs (*i.e.*, directed acyclic graphs), confirming that $\mathcal{JSO}$ (*i.e.*, subtyping relations and the announced transformations) is an operad is tedious but relatively straightforward (similar to proving that directed graphs and homomorphisms over them define a category, the category **Graph**, but involving more variable-indexing).

Having defined $\mathcal{JSO}$, each iteration of the construction process of the Java subtyping relation (as described in Section 2.1) can be concisely described by a single $\mathcal{JSO}$ morphism, which we name $JSM$. As such, we define

$$JSM := merge \circ (copy, flip, flat)$$

where $\circ$ is operadic composition (see [45, Sec. 7.4], particularly Remark 7.4.1.2, for notation).

**Bounded Type Variables** In the description of $\mathcal{JSO}$ transformations we did not consider type variable bounds, assuming instead that all type variables have the default upper bound `O` (*i.e.*, `Object`). Bounded type variables can be handled in $\mathcal{JSO}$ using an additional *clip* morphism that takes the three output relations of *copy*, $flip$ and $flat$ and, for each generic class `C`, "throws away" (in an operadic way) the types (and their subtype relations) that are not within the bounds of the type variable of `C` (remember that, for simplicity, we are assuming a generic class has a single type variable). So far, however, it is not clear to us how to handle the case when the type variable "appears in its own bound" (*i.e.*, the case when we have 'F-bounded polymorphism'). Given the importance of handling this case, we thus leave the modeling of bounds of type variables altogether to future work.

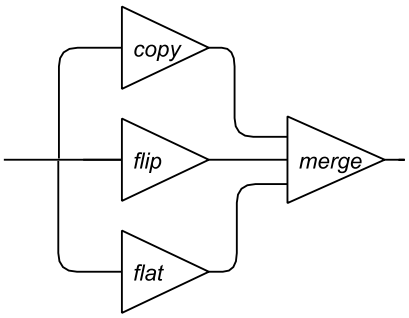



Figure 3.1: The main morphisms of $\mathcal{JSO}$, and how they are composed to define successively more accurate versions of the Java subtyping relation.

# 4 Examples of Applying $\mathcal{JSO}$

To demonstrate how $\mathcal{JSO}$ works, we present in this section two examples of how $\mathcal{JSO}$ defines the subtyping relation for two simple Java programs.

The first example furthers the construction of the subtyping relation for the simple program we presented in Section 2.1 to define the third version of the subtyping relation between types of rank 0, 1 and 2 for that program. This version of the relation will look as in Figure A.1.

The second Java program example, in addition to the declaration of class `C`, has a second generic class declaration

`class D<T> extends Object {}`.

The subclassing relation and the initial subtyping relation for this program will look as in Figure A.2, while the second version of the subtyping relation for this program will look as in Figure A.3. See Appendix A for more details on these examples.

# 5 Related Work

The addition of generics to Java has motivated much research on generic OOP and also on the type safety of Java and similar languages. Much of this research was done before generics were added to Java. For example, the work in [8, 7, 10, 16, 47] was mostly focused on researching OO generics, while the work in [18, 20, 19, 21] was focused on type safety. Some research on generics, and generic type inference, was also done after generics were added to Java, *e.g.*, [40, 50, 4, 26].

However, FJ/FGJ (Featherweight Java/Featherweight Generic Java) [28] is probably the most prominent work done on the type safety of Java, including generics. FJ/FGJ did not consider variance annotations (or wildcard type parameters) though.

Separately, probably as the most complex feature of Java generics, the addition of "wildcards" (*i.e.*, wildcard type parameters) to Java (in [49], which is based on the earlier research in [29]) also generated some research that is particularly focused on modeling wildcards and variance annotations [48, 12, 31, 11, 46]. This significant and substantial work points yet to the need for more research on wildcards and generics.

The use of category theory tools in computer science is well-known [35, 45], particularly in researching the semantics of programming languages [38, 39, 43, 27, 9, 23, 15]. After all, most of category theory has a constructive computational "feel" to it [37, 22]. Some category theory tools, particularly coalgebras, have also been used to research OOP [13, 17, 30, 36]. However, to the best of our knowledge, operads (or symmetric multicategories) [33] have not been used before in programming languages research.[4] Given the increasing awareness of its power and its wide array of practical scientific applications, it is currently expected that the use of category theory in computer science research (and other scientific research) will further increase [32, 45].

# 6 Discussion and Future Work

The simple operad $\mathcal{JSO}$ we presented in this paper seems to nicely capture some of the main features of the generic subtyping relation in Java and similar OO languages, particularly its self-similarity. Based on our development of $\mathcal{JSO}$ as a model of generic Java subtyping that particularly reveals its intricate self-similarity, we believe that, generally speaking, using category theory (which has a rich set of tools, including powerful notions such as operads) more may hold the key to having a better understanding of complex features of programming languages, such as wildcards and generics.

More specifically, we believe the self-similarity of the Java subtyping relation (revealed by $\mathcal{JSO}$) is obscured by three factors: first, the merging step (the work of the *merge* morphism of $\mathcal{JSO}$) in the construction of the subtyping relation, particularily its identification of some of the types constructed by other components of $\mathcal{JSO}$, thereby ambiguating the origin of these types. Second, we believe the self-similarity is also obscured by the inexpressibilty of the `Null` type in Java, which makes some of the types constructed by $\mathcal{JSO}$ involving `Null` (*e.g.*, type `C<N>` above), and their subtyping relations, sound unfamiliar.

In our opinion, the third reason for obscuring the self-similarity of the generic subtyping relation in Java is thinking about the relation in *structural-typing* terms rather than *nominal-typing* ones. Although it seems the polymorphic structural subtyp-

[4] As we mentioned earlier, what we call operads are called 'symmetric multicategories' or 'colored operads' in some category theory literature, where an operad in this literature is a one-object multicategory. In this paper we follow the naming convention of Spivak [45] and others.

ing relation (with variance annotations) exhibits self-similarity similar to the one $\mathcal{JSO}$ demonstrates for Java, but it should be noted that nominal typing in languages such as Java, C# and Scala, particularly the ensuing identification of type/contract inheritance with nominal subtyping [14, 2, 5], are used in $\mathcal{JSO}$, first, to define the initial version of the nominal subtyping relation in its iterative construction process (directly based on the subclassing/inheritance relation), and, second, in the embedding step (into the initial subtyping relation) in each iteration of the process. A simple and strong connection between type inheritance and subtyping does not exist when thinking about the Java subtyping relation in structural typing terms. It seems to us that not making this connection, keeping instead the subtyping relation separate and independent from the inheritance relation, makes it harder to see the self-similarity of generic nominal subtyping, its intricacies, and its connections to the subclassing/inheritance relation.

Having said that, more work on $\mathcal{JSO}$ is needed however, to make it model Java subtyping more accurately. Even though we presented how $\mathcal{JSO}$ can model the three generic subtyping rules in Java, and how they are combined to define the subtyping relation, $\mathcal{JSO}$ does not model bounded type variables in particular. We hinted earlier in this paper to how bounded type variables can be modeled, but the actual definition of $\mathcal{JSO}$ to include them remains to be done.

Also, $\mathcal{JSO}$ can be made more general—covering more features of the Java generic subtyping relation and also suggesting how generic subtyping in Java can be extended—if $\mathcal{JSO}$ includes a notion of *type intervals*, which roughly are 'intervals over the subtyping relation' [3], thereby supporting types other than `Object` and `Null` as upper and lower bounds, (1) firstly, for type variables [41, 40, 42], and, (2) secondly, for generic type parameters (as such, type intervals smoothly subsume and generalize wildcard types).

It may be useful also to extend $\mathcal{JSO}$ to model subtyping between generic types that have type variables in them (in this paper we modeled the subtyping relation between *ground* generic types, which have no type variables in them). Unifying the last two suggestions, we believe it may be useful if $\mathcal{JSO}$ more generally supports a notion of *nominal intervals* [4], where type variables are viewed as names for type intervals and where intervals with the same lower and upper bounds but with different names are considered unequal type intervals.

# References


[1] C# language specification, version 5.0. http://msdn.microsoft.com/vcsharp, 2015.

[2] Moez A. AbdelGawad. A domain-theoretic model of nominally-typed object-oriented programming. *Journal of Electronic Notes in Theoretical Computer Science (ENTCS), DOI: 10.1016/j.entcs.2014.01.002.*, 301:3–19, 2014.

[3] Moez A. AbdelGawad. Subtyping in Java with generics and wildcards is a fractal. Technical report, arXiv.org:1411.5166 [cs.PL], 2014.

[4] Moez A. AbdelGawad. Towards an accurate mathematical model of generic nominally-typed OOP (extended abstract). 2016.

[5] Moez A. AbdelGawad. Why nominal-typing matters in OOP. *Preprint available at http://arxiv.org/abs/1606.03809*, 2016.

[6] Moez A. AbdelGawad. Towards a Java subtyping operad. Proceedings of FTfJP'17, Barcelona, Spain, June 18-23, 2017. https://doi.org/10.1145/3103111.3104043

[7] Ole Agesen, Stephen N Freund, and John C Mitchell. Adding type parameterization to the Java language, 1997.

[8] Joseph A. Bank, Barbara Liskov, and Andrew C. Myers. Parameterized types and Java. Technical report, 1996.

[9] Andrej Bauer and Dana Scott. A new category for semantics. Technical report, Carnegie Mellon, 2001.

[10] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, volume 33, pages 183–200, Vancouver, BC, October 1998. ACM, ACM SIGPLAN.

[11] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A model for Java with wildcards. In *ECOOP'08*, 2008.

[12] Nicholas Cameron, Erik Ernst, and Sophia Drossopoulou. Towards an existential types model for Java wildcards. *$9^{th}$ Workshop on Formal Techniques for Java-like Programs,*, 2007.

[13] Peter S. Canning, William R. Cook, Walter L. Hill, J. Mitchell, and W. Olthoff. F-bounded polymorphism for object-oriented programming. In *Proc. of Conf. on Functional Programming Languages and Computer Architecture*, 1989.

[14] Robert Cartwright and Moez A. AbdelGawad. Inheritance *Is* subtyping (extended abstract). In *The $25^{th}$ Nordic Workshop on Programming Theory (NWPT)*, Tallinn, Estonia, 2013.

[15] Robert Cartwright, Rebecca Parsons, and Moez A. AbdelGawad. *Domain Theory: An Introduction.* Preprint available at http://arxiv.org/abs/1605.05858, 2016.

[16] Robert Cartwright and Jr. Steele, Guy L. Compatible genericity with run-time types for the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, volume 33, pages 201–215, Vancouver, BC, October 1998. ACM, ACM SIGPLAN.

[17] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *POPL'90 Proceedings*, 1990.

[18] Sophia Drossopoulou and Susan Eisenbach. Java is type safe–probably. In *ECOOP'97–Object-Oriented Programming*, pages 389–418. Springer, 1997.

[19] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the java type system sound? *TAPOS*, 5(1):3–24, 1999.

[20] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–183. ACM, 1998.

[21] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal syntax and semantics of Java*, pages 241–269. Springer, 1999.

[22] Maarten M. Fokkinga. A gentle introduction to category theory: The calculational approach. On web, Jun 1994. Book under preparation.

[23] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. W. Mislove, and D. S. Scott. *Continuous Lattices and Domains*, volume 93 of *Encyclopedia Of Mathematics And Its Applications*. Cambridge University Press, 2003.

[24] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2005.

[25] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. Addison-Wesley, 2014.

[26] Radu Grigore. Java generics are turing complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 73–85, New York, NY, USA, 2017. ACM.

[27] C. A. Gunter and Dana S. Scott. *Handbook of Theoretical Computer Science*, volume B, chapter 12 (Semantic Domains). 1990.

[28] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[29] Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. In *In ECOOP*, pages 441–469. Springer-Verlag, 2002.

[30] Bart Jacobs. Objects and classes, coalgebraically. In *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Acad. Publ, 1996.

[31] Andrew J. Kennedy and Benjamin C. Pierce. On decidability of nominal subtyping with variance. In *International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD)*, 2007.

[32] F William Lawvere and Stephen H Schanuel. *Conceptual mathematics: a first introduction to categories*. Cambridge University Press, 2009.

[33] T. Leinster. *Higher Operads, Higher Categories*. Higher Operads, Higher Categories. Cambridge University Press, 2004.

[34] Martin Odersky. The scala language specification, v. 2.9. http://www.scala-lang.org, 2014.

[35] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[36] Erik Poll. A coalgebraic semantics of subtyping. *Electronic Notes in Theoretical Computer Science*, 33:276 – 293, 2000. CMCS'2000, Coalgebraic Methods in Computer Science.

[37] David E. Rydeheard and Rod M. Burstall. *Computational Category Theory*. Prentice Hall International (UK) Ltd., 1988.

[38] Dana S. Scott. Data types as lattices. *SIAM Journal of Computing*, 5(3):522–587, 1976.

[39] Dana S. Scott. Domains for denotational semantics. Technical report, Computer Science Department, Carnegie Mellon University, 1983.

[40] Dan Smith and Robert Cartwright. Java type inference is broken: Can we fix it? *OOPSLA*, pages 505–524, 2008.

[41] Daniel Smith. Completing the Java type system. Master's thesis, Rice University, November 2007.

[42] Daniel Smith. *Designing Type Inference for Typed Object-Oriented Languages*. PhD thesis, Rice University, May 2010.

[43] M. B. Smyth and Gordon D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11:761–783, 1982.

[44] David I Spivak. The operad of wiring diagrams: Formalizing a graphical language for databases, recursion, and plug-and-play circuits. *Available at: http://arxiv.org/abs/1305.0297*, 2013.

[45] David I Spivak. *Category theory for the sciences*. MIT Press, 2014.

[46] Alexander J. Summers, Nicholas Cameron, Mariangiola Dezani-Ciancaglini, and Sophia Drossopoulou. Towards a semantic model for Java wildcards. *$10^{th}$ Workshop on Formal Techniques for Java-like Programs,*, 2010.

[47] Kresten Krab Thorup and Mads Torgersen. Unifying genericity. In *ECOOP 99–Object-Oriented Programming*, pages 186–204. Springer, 1999.

[48] Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. Wild FJ. In *Foundations of Object-Oriented Languages*, 2005.

[49] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *SAC*, 2004.

[50] Yizhou Zhang, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, and Andrew C. Myers. Lightweight, flexible object-oriented generics. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 436–445, New York, NY, USA, 2015. ACM.

# A Demonstrating the Java Subtyping Operad

In this appendix we present two examples demonstrating how $\mathcal{JSO}$, as defined in this paper, defines the construction of the subtyping relation in two sample Java programs with simple class declarations.

## A.1 Example 1: One Generic Class, and Types of Rank 0, 1 and 2

In this section we continue the example we started in Section 2.1. To shorten the type names, we use the numbers 0-7 as labels for the types in Figure 2.4, as presented in the upper-right corner graph of Figure A.1.

Using this labeling, the output subtyping relation resulting from applying the $JSM$ morphism of $\mathcal{JSO}$ to the relation in Figure 2.4 as its input will be as depicted in the main graph of Figure A.1. The color highlighting in the graph helps see part of the effect of the individual component morphisms of $JSM$—*copy* producing the relations in green, $flip$ producing the relations in red, $flat$ producing the flat relation at the bottom (*i.e.*, between the "atom types", right above type `N` in Figure A.1), and *merge* producing the whole third version of the subtyping relation for the sample Java program.

## A.2 Example 2: Two Generic Classes, and Types of Rank 0 and 1

Assuming another Java program that has the same class declaration for `C` but has a second generic class declaration

```
class D<T> extends Object {}
```

we will then have the subclassing and initial subtyping relations as in Figure A.2. Applying $JSM$ to the relation in Figure A.2, the resulting second version of the relation will be as in Figure A.3.

The reader is encouraged to confirm that she sees the relations in Figure A.1 and Figure A.3 as intuitive. She is also encouraged to construct the third version of the subtyping relation for the second program, which results from the application of $JSM$ to the relation in Figure A.3.

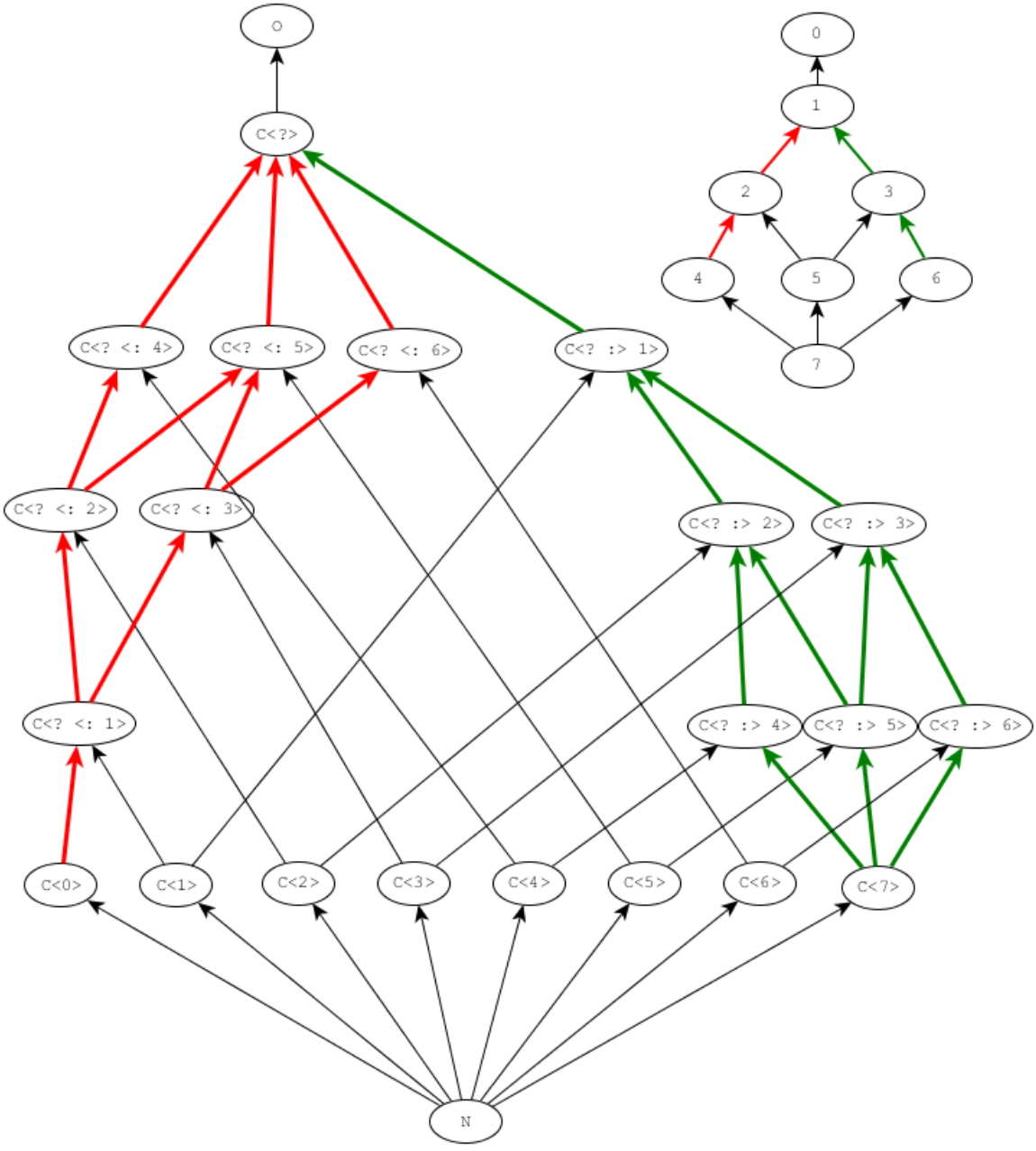



Figure A.1: Subtyping between rank 0, 1 and 2 types (with one generic class)

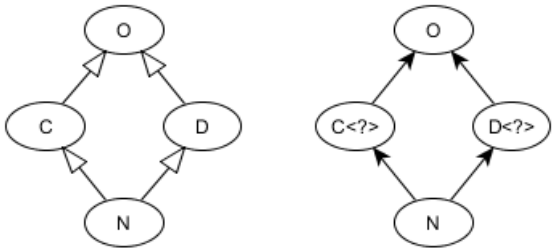



Figure A.2: Subclassing and subtyping between rank 0 types (with two generic classes)

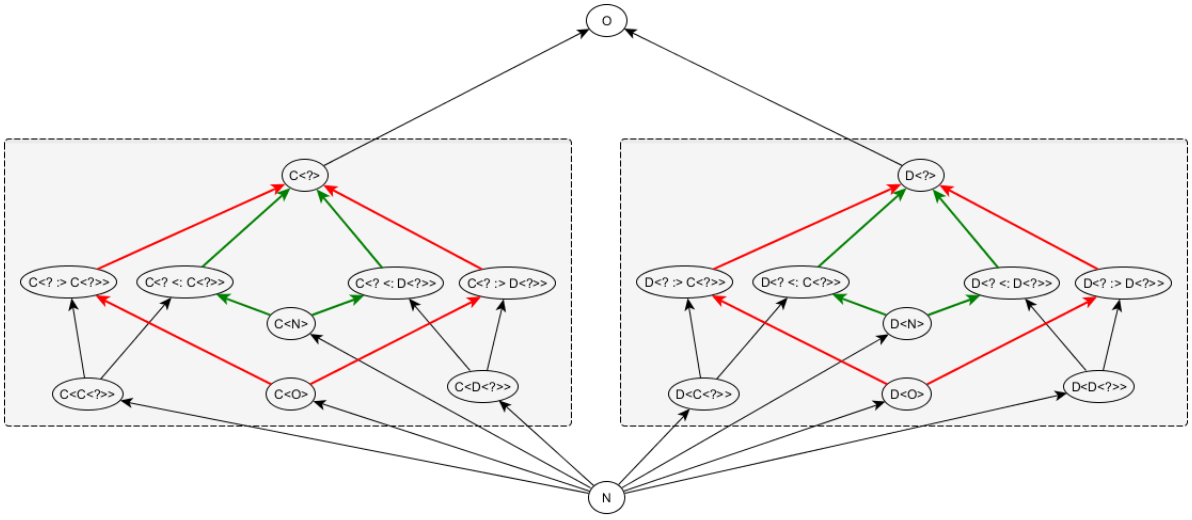



Figure A.3: Subtyping between rank 0 and 1 types (with two generic classes)