Towards an Order and Category Theoretic Model of Java Generics (extended version)

Moez Abdelgawad

June 17, 2020

Towards an Order and Category Theoretic Model of Java Generics (extended version)

Moez A. AbdelGawad[*][†]
Assistant Professor, Informatics Research Institute, SRTA-City, Alexandria, Egypt
moez@cs.rice.edu

Abstract

The mathematical modeling of generic type systems of mainstream object-oriented programming languages such as Java, C#, C++, Scala and Kotlin is a challenge. This is mainly due to these languages supporting features such as 'variance annotations,' 'F-bounded type parameters,' and 'type erasure.'

In this paper we present an order-theoretic and lattice-theoretic approach to modeling generics in nominally-typed OOP type systems that aims to build a simpler and more intuitive model than the extant "existentials-based" model. The approach also uses some elementary notions in category theory, as a generalization of order and lattice theory.

Our model, as constructed so far, reveals characteristics and relations underlying type systems of generic mainstream OOP languages—such as a Galois connection between subclassing and subtyping—that seem to have not been formalized and made explicit before. The model also suggests how support for generics in these languages may be simultaneously extended and simplified, including proposing features such as 'interval types,' 'doubly F-bounded generics,' 'default type arguments,' and 'cofree types.'

1 Introduction

In mainstream OOP languages such as Java [41, 43], C# [1], Scala [61], C++ [2], and Kotlin [3], classes, interfaces, and traits that are parameterized by one type parameter or more are, collectively, called *generic classes*[1], or simply *generics*. Generics are supported in these languages to enhance the expressiveness of their type systems [23, 33, 80]. In these languages, generics provide a counterpart of the 'parametric polymorphism' feature found in functional programming languages such as ML [60] and Haskell [58].

In the theory of functional programming languages, 'existential types' (also called *existentials*) are used to model abstract data types [30], which are common in these languages. Due to support for variance annotations (*e.g.*, Java wildcards) in particular, the most well-known model of generics in OOP languages is also based on existential types (*e.g.*, in the form of 'F-bounded existentials,' or 'coinductive F-bounded existentials' [79]), indicating thereby a strong influence by research on functional programming languages.

Existential types arise naturally in the context of functional programming. However, as illustrated by the results and discussions in volumes of research on modeling generics and wildcards (*e.g.*, [24, 25, 76, 79, 81, 82]), existentials, while elaborate, do not match well with mainstream OOP languages, since they—existentials—do not smoothly interact with OO inheritance and subtyping, which are fundamental notions in mainstream OOP languages. Accordingly, we believe that existential types are not a natural and intuitive basis for a model of generics in OO programming languages.

The mismatch between existential types and fundamental features of OOP has practical consequences. For example, generics-related diagnostic error messages emitted by compilers of nominally-typed OOP languages are usually cryptic messages (such as the notorious error messages containing "capture of ..." that are produced by `javac`, the standard Java compiler), and these messages are often minimally helpful to OO software developers in fixing errors in their code. As a consequence, many mainstream OOP developers shy away from making extensive or advanced use of generics in their code.

Further, as demonstrated by research on nominally-typed OO type systems, the centrality of generics to these type systems has hindered the progress of mainstream OOP languages, due to the complexity and unnaturalness of the extant existentials-based model of generics.[2]

Order theory is the branch of mathematics focused on the study of (partially or totally) ordered sets. In this work we demonstrate how an order-theoretic model of OO generics can be constructed that

1. directly models OO inheritance and OO subtyping (*i.e.*, is fundamentally object-oriented in its essence and nature),

[*] Also with the Computer Science Dept., Rice University, Houston TX, USA (Remote Visiting Scholar).

[†] Also with the Computer and Systems Engineering Department, Faculty of Engineering, Alexandria University, Alexandria, Egypt (Adjunct Assistant Professor, `moez@alexu.edu.eg`).

[1] In this work, interfaces (*e.g.*, in Java) and traits (*e.g.*, in Scala) are treated as abstract classes. Thus, the term 'class' in this paper refers to classes and other similar type-constructing constructs.

[2] Check, for example, [55], or the dense sections of the Java Language Specification (JLS) that specify crucial parts of its generic type system, *e.g.*, [43, §4.5 & §5.1.10].

2. includes all main features of OO generics (including the "complex" features mentioned earlier),
3. does not make explicit use of existential types, and thus can be a basis for significantly improving and simplifying the generics-related diagnostic messages of mainstream OOP compilers, and,
4. suggests how type systems of mainstream generic object-oriented programming languages can be smoothly extended to include other generics-related features.

Since every ordered set (*a.k.a.*, 'poset') is a category, category theory can be viewed as a generalization of order theory (which, in turn, is a generalization of lattice theory). Our approach, particularly when modeling F-bounded generics and type erasure, uses lattice theoretic concepts and notions that, historically, seem to have been studied more extensively as elementary notions in category theory.

As such, this paper is structured as follows. In §2 we present a summary of the mathematical prerequisites for constructing our order theoretic and category theoretic model of generic OOP. In §3 we present the model, constructing it and analyzing it using the tools presented in §2. In §4 we propose a number of extensions to current generic OOP type systems that are suggested by the model. In §5 we present some related work. We conclude in §6, where we present some discussion and final remarks, and also present potential future work.[3]

For the sake of concreteness, the presentation in this paper focuses on modeling the essential features of generics in Java in particular. The model presented herein, however, can readily model also the essential features of other generic nominally-typed OOP type systems.

2 Mathematical Background

In this section we briefly hint at the main mathematical notions we use in this paper. Some of the constructs are standard ones, which we either use "as is" or only give them names that make them more intuitive for use in an OO context, while others were defined for the purposes of constructing our order-theoretic model of generic OOP.

The most fundamental mathematical construct used in this work is that of a *poset, i.e.*, a partially-ordered set, which is simply a set provided with an ordering on elements of the set [35, 67]. Examples of posets most relevant to this work are the poset $\mathbb{C}$ of classes of an OO program ordered by the subclassing (*a.k.a.*, inheritance) relation ($\trianglelefteq$), and the poset $\mathbb{T}$ of (parameterized) types in an OO program ordered by the subtyping relation ($<:$).

Next, three operators that construct new posets given some input posets are also used in our approach. In particular, we use a *partial products* constructor $\ltimes$, a *wildcards* constructor $\triangle$, and an *intervals* constructor $\Updownarrow$. As its name indicates, operator $\ltimes$ constructs a *partial* product of two input posets relative to a subset of the first input poset. Operators $\triangle$ and $\Updownarrow$ construct intervals over an input poset, ordering these intervals by containment, where $\triangle$ constructs a restricted subset of the intervals (ones with $\top$ or $\bot$ as their upper- or lower-bound, resp.) while $\Updownarrow$ constructs all intervals. Operators $\ltimes$, $\triangle$, and $\Updownarrow$ are new poset constructors that we defined for purposes of use in this work. Their precise mathematical definitions are presented in Appendix A.

To model type erasure and F-bounded generics, we also make use of standard notions of order theory such as *Galois connections*, *pre-fixed points* and *post-fixed points*, where a monotonic *endofunction* F (*i.e.*, a monotonic "self-map" F) defined over some poset is used as a "points generator" in the poset. In category theory, where an *endofunctor* F plays the role of a generator, the three notions generalize to *adjunctions*, F*-algebras*, and F*-coalgebras*, respectively. In (power)set theory, *inductive/coinductive sets* correspond to pre-fixed/post-fixed points of a set generator [65, Ch.21]. Similarly, in functional programming theory, *inductive/coinductive types* are defined as inductive/coinductive sets, respectively, where type constructors get modeled as generators (over the inclusion/subtyping relation).

Similarly, in generic OOP theory we define F*-supertypes* and F*-subtypes* as the types corresponding to pre-fixed points and post-fixed points of a generic class F that is modeled as a type generator (over the generic OO subtyping relation).

Additionally, corresponding to *least pre-fixed points* and *greatest post-fixed points* in order theory (*initial algebras* and *final coalgebras* in category theory, resp.), we also define and use the notions of the *free type* (as the least F-supertype) and the *cofree type* (as the greatest F-subtype) corresponding to a (generic) class F[4].

3 An Order Theoretic Approach to Modeling Generic OOP

The subtyping relation between *ground* parameterized types (*i.e.*, ones that contain no type variables) is the basis for defining the full subtyping relation in generic OOP [47, 65]. Hence, in this section we focus on presenting the construction of the subtyping relation between ground types of generic OOP.[5]

A bird's-eye view of our order-theoretic approach to modeling generics summarizes the approach into the following four main steps:

- the first and most fundamental step in our approach is a conceptual one, namely, maintaining a strong and very clear distinction between *inheritance* (*a.k.a.*, *subclassing*), as an ordering relation defined on *classes*,

[3] Appendix A presents a more detailed summary of the mathematical notions used in constructing our model.

[4] An example of a free type is the Java type `C<?>`. Cofree types, *e.g.*, `C<!>`, are largely unsupported in generic OO type systems

[5] The construction of the full subtyping relation, in which type variables are included in subtyping rules, is left as future work. See §6.

on one hand, and *subtyping*, as an ordering relation defined on parameterized *types*[6], on the other hand, then,

- secondly (§3.1), describing—using tools from order theory—how the *infinite* subtyping relation between ground parameterized types (which defines a poset) can be constructed based only on the *finite* subclassing relation (another poset) and an auxiliary containment ordering relation (between wildcard type arguments; a 3$^{\text{rd}}$ poset[7]), then,
- thirdly (§3.2), analyzing the constructed subtyping relation, and its relation to the subclassing relation (using tools from order theory and category theory), and, accordingly, making some observations about generic OO type systems, and,
- fourthly (§4), suggesting some natural extensions to generic OO type systems that are motivated by the preceding construction and analysis steps.

3.1 Constructing The Subtyping Relation between Ground Parameterized Types

Given a finite subclassing poset $\mathbb{C}$ (of classes C ordered by subclassing $\trianglelefteq$), we construct the subtyping poset $\mathbb{T}$ (of ground parameterized types T ordered by subtyping <:) as follows.

First, we make a couple of assumptions regarding $\mathbb{C}$. We assume that a generic class in C (the universe of $\mathbb{C}$) takes exactly one type argument. We further assume, for simplicity, that if a generic class inherits from (*a.k.a.*, 'is a subclass of') another generic class then the superclass is passed the parameter of the subclass as its type argument (*e.g.*, as in the Java class declaration `class C<T> extends D<T>`, where `T`, the type parameter of `C`, is used directly as the type argument of the superclass `D`).[8] We also assume that C always has two distinct non-generic classes, say `Object` and `Null`, that play the role of the top ($\top$) and bottom ($\perp$) elements of the subclassing relation $\trianglelefteq$.

Next, if $G \subseteq C$ is the set of *generic* classes in $\mathbb{C}$, then operators $\ltimes$ and $\triangle$ (see §A) can be used to define the infinite subtyping relation $\mathbb{T}$ between ground parameterized types as the solution of the following recursive equation

$$\mathbb{T} = \mathbb{C} \ltimes_G \triangle(\mathbb{T}). \tag{1}$$

The subtyping poset $\mathbb{T}$, as the (least fixed point) solution of Equation (1), can then be constructed iteratively, using the derived iterative equation

$$\mathbb{T}_{i+1} = \mathbb{C} \ltimes_G \triangle(\mathbb{T}_i). \tag{2}$$

In words, Equation (2) specifies that, given $\mathbb{T}_i$ (a finite approximation of $\mathbb{T}$), operator $\triangle$ constructs the wildcard type arguments corresponding to $\mathbb{T}_i$, ordered by containment. Operator $\ltimes$, as a partial product operator, then constructs $\mathbb{T}_{i+1}$ by pairing generic classes (members of G) in $\mathbb{C}$ with the type arguments constructed by $\triangle$ then adding types that correspond to non-generic classes in $\mathbb{C}$ (check Definition A.9 in §A), thereby constructing the poset $\mathbb{T}_{i+1}$ of ground parameterized types ordered by subtyping (*i.e.*, the next approximation of $\mathbb{T}$). As is standard in mathematics (particularly in set theory, order/lattice theory, and domain theory [32, 35, 36, 38, 40, 73, 77]), the full infinite poset $\mathbb{T}$ is obtained as the *limit* of this iterative construction process.

We illustrate the order theoretic process of constructing subtyping from subclassing using a simple Java example.

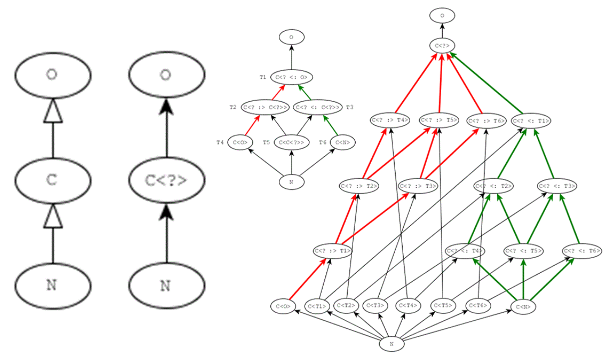


Figure 1. Constructing generic subtyping using subclassing.

Example 3.1. Assuming the Java class declaration
`class C<T> extends Object {...}`,
the posets $\mathbb{C}, \mathbb{T}_1, \mathbb{T}_2$ and $\mathbb{T}_3$ that correspond to this declaration can be represented by Hasse diagrams as in Figure 1 (where `O` and `N` are shorthands for classes `Object` and `Null` and their corresponding types, respectively, and where type names `T1` to `T6` stand for the six types in $\mathbb{T}_2$ other than `O` and `N` that are shortened for illustrative use in $\mathbb{T}_3$). The similarity of $\mathbb{T}_1$ to subsets of $\mathbb{T}_2$ and of $\mathbb{T}_2$ to subsets of $\mathbb{T}_3$ (subsets highlighted in red and green) should be noted.

Example 3.2. More figures illustrating examples of constructions of subtyping relations (posets $\mathbb{T}$) using more complex subclassing relations (*i.e.*, more complex posets $\mathbb{C}$) can be found in our earlier work [9, 12, 17].

We now move on to analyzing the relation between subclassing and subtyping, then, accordingly, we formalize a fundamental property of generic OOP type systems.

[6] In OOP literature, the expression *parameterized types* is sometimes used interchangeably with *object types*, *class types*, *reference types*, *generic types*, or even just *types*.

[7] These three ordering relations lie at the heart of all mainstream generic OO type systems.

[8] We keep a study of how these two assumptions can be relaxed to future work. (See §6.)

3.2 The Erasure Galois Connection (EGC), and Nominal Typing and Nominal Subtyping

Type erasure—where, intuitively, the type arguments of a parameterized type get "erased"—is a feature prominent in Java (and Java-based OO languages such as Scala and Kotlin), but that can also be defined and made explicit in other generic nominally-typed OOP languages.

In the order-theoretic approach to modeling generics (where a clear separation between types and classes is maintained), type erasure is modeled as a mapping, called *erasure*, from types to classes.[9] Further, as hinted at in §A, the 'most general (wildcard) instantiation' of a generic class is called the *free type* corresponding to the class.[10]

Crucially, by maintaining a clear separation between classes ordered by subclassing, on one hand, and types ordered by subtyping, on the other, the construction of the subtyping relation using the subclassing relation (as presented in §3.1) allows us to observe that, combined, the *erasure* mapping (from types to classes) and the *free type* mapping (from classes to types) define a Galois connection between the two fundamental relations of generic nominally-typed OOP (see §A for the definition of Galois connections).

More formally, if Er denotes the *erasure* mapping that maps a parameterized type to the class used to construct the type (*i.e.*, "erases" type arguments of the type)[11] and if Ft denotes the *free type* mapping that maps a class to its most general wildcard instantiation[12], then a direct expression of the Galois connection between subclassing and subtyping in generic OOP states that for all parameterized types t and classes c we have

$$Er\,(t) \trianglelefteq c \Leftrightarrow t <: Ft\,(c) \qquad (3)$$

where $\trianglelefteq$ is the subclassing relation between classes and $<:$ is the subtyping relation between parameterized types.

The Galois connection between subclassing and subtyping, which we call the Erasure Galois Connection (EGC), can be equivalently expressed, indirectly but in more familiar OOP terms, as stating that for all classes `C` and `D`, if `C` is a subclass of `D` then all instantiations of `C` are subtypes of `D<?>` (the most general instantiation of `D`) and vice versa—a statement that is intuitively familiar to all professional mainstream OO developers.

[9]To model *erased types*, the erasure mapping is then composed with a notion of a *default type* that maps each generic class to some corresponding parameterized type (*i.e.*, a particular instantiation of the class). As a proposed extension of extant generic OO type systems, we discuss *default type arguments* (DTAs) and *default types* in some more detail in §4.

[10]For example, a generic class `C` with one type parameter has the type `C<?>` as its corresponding free type. A *non*-generic class is mapped to the only type it constructs—a type typically homonymous to the class—as its corresponding free type.

[11]*E.g.*, Er (`List <Integer>`) = `List`.

[12]*E.g.*, Ft (`List`) = `List<?>`.

Formally, Equation (3) can be equivalently re-expressed as stating that, for all classes `C`, `D`, and for all types `T`, we have

$$\texttt{C} \trianglelefteq \texttt{D} \Leftrightarrow \texttt{C<T>} <: \texttt{D<?>} \qquad (4)$$

(note that, for all types `T`, class `C` is the *erasure* of `C<T>`, *i.e.*, Er (`C<T>`) = `C`, and that `D<?>` is the *free type* corresponding to class `D`, *i.e.*, Ft (`D`) = `D<?>`).

We illustrate EGC, the Galois connection between subtyping and subclassing in generic OOP, using an example.

Example 3.3. In Java, the statement

$$\texttt{LinkedList} \trianglelefteq \texttt{List} \Leftrightarrow \texttt{LinkedList<String>} <: \texttt{List<?>}$$

asserts that stating that class `LinkedList` is a subclass of `List` is logically equivalent to stating that `LinkedList<String>` is a subtype of the free type `List<?>`, which is an intuitively true statement in Java.[13]

More significantly however, it should be noted that the EGC (stated as either Equation (3) or Equation (4)) formally expresses a fundamental property of generic OOP that is a consequence of generic OO type systems being *nominally-typed* [6, 65], namely, that

inheritance is *the* source of subtyping

in generic OOP. As is clear by its formalization as the EGC, this fundamental property of generic OOP states, in one direction, that inheritance is *a* source of subtyping (*i.e.*, the subclassing relation, which is inherently nominal, causes subtyping between parameterized types in generic OOP) and, in the other direction, it states that inheritance/subclassing is *the only* source of subtyping (*i.e.*, subtyping between parameterized types comes from nowhere else other than from the inherently nominal subclassing relation, hence making subtyping in generic OOP languages a *nominal* relation too).[14]

We ponder over this fundamental property (and its formal expression as the EGC) and over the value of nominal typing and nominal subtyping in mainstream generic OOP type systems a little more in §6.

Beyond revealing the Galois connection between subtyping and subclassing, and allowing the precise formalization of a fundamental property of generic OOP, the value of the order theoretic approach to modeling generic OOP type systems is further illustrated by the approach suggesting some extensions to such type systems, which we present in the next section.

[13]In this statement, variables t and c in Equation (3) are instantiated to type `LinkedList<String>` and class `List`, respectively; or, equivalently, variables `C`, `D` and `T` in Equation (4) are instantiated to class `LinkedList`, class `List` and type `String`, respectively.

[14]It should be noted that this property of generic OOP—*i.e.*, that *inheritance is the only source of subtyping*—is the counterpart of the *inheritance is subtyping* property of non-generic nominally-typed OOP [7, 31].

4 Suggested Extensions of Generic OOP Type Systems

As presented in §3, constructing the generic subtyping relation, using tools from order theory, and noting the Galois connection that exists between subtyping and subclassing, unitedly suggest how generics in nominally-typed OOP languages can be extended in four specific directions.

4.1 Interval Types

Although type `Null` (the type that corresponds to the homonymous class `Null`) is explicitly supported only in few mainstream OOP languages, in §3 we have explicitly used type `Null` as the bottom element of the subtyping relation <: and, thus, also it was used as an explicit lowerbound of wildcard type arguments.[15] In Java, an (explicit) wildcard type argument must have type `Object` as its upperbound or type `Null` as its lowerbound. The order theoretic approach to modeling generic OOP, however, suggests how wildcard type arguments in Java can be extended to support having general lower bounds and upper bounds, simultaneously. In particular, it suggests how *interval type arguments* can be defined as a generalization of wildcard type arguments, and accordingly how *interval types* can be defined as a generalization of wildcard types (*i.e.*, parameterized types with top-level wildcard type arguments).

Formally, interval types and the subtyping relation between them can simply be constructed, using tools from order theory, by simply replacing the wildcards operator $\triangle$ in Equation (1) of §3.1 with the intervals operator $\Updownarrow$, which, given a subtyping relation as input, constructs interval type arguments and the containment relation between them (see §A and Appendix A for more on operators $\triangle$ and $\Updownarrow$.)

Similar to poset $\mathbb{T}$ in §3.1, the poset $\mathbb{S}$ of 'the subtyping relation on interval types' can also be constructed iteratively using the subclassing relation $\mathbb{C}$. Formally, poset $\mathbb{S}$ is defined as the solution of the recursive poset equation

$$\mathbb{S} = \mathbb{C} \ltimes_G \Updownarrow(\mathbb{S})\,. \tag{5}$$

Other than replacing $\triangle$ with $\Updownarrow$, the iterative construction of $\mathbb{S}$ proceeds in exactly the same manner as the construction of $\mathbb{T}$ presented in §3.1, namely using the equation

$$\mathbb{S}_{i+1} = \mathbb{C} \ltimes_G \triangle(\mathbb{S}_i)\,. \tag{6}$$

We illustrate the main difference between the construction of $\mathbb{T}$ (subtyping with wildcard types) and the construction of $\mathbb{S}$ (subtyping with interval types) using a simple Java example.

Example 4.1. Assuming the Java class declarations

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

the posets $\mathbb{C}$, $\mathbb{T}_1$, $\mathbb{T}_2$, $\mathbb{S}_1$ and $\mathbb{S}_2$ corresponding to these declarations can be represented by Hasse diagrams as in Figure 2.

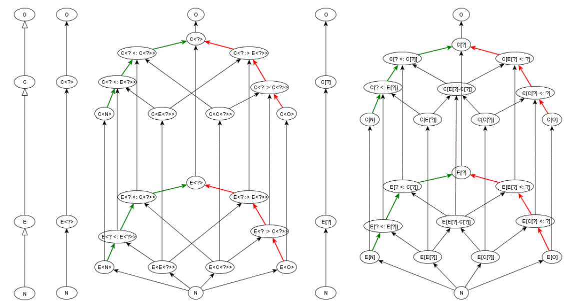


Figure 2. Constructing subtyping (w/ wildcard types and w/ interval types) using subclassing.

As can be seen by comparing the diagrams for $\mathbb{T}_2$ and $\mathbb{S}_2$ (the middle and rightmost Hasse diagrams in Figure 2, respectively), interval types are usually *more expressive* than (and always no less expressive than) wildcard types, which is a consequence of operator $\Updownarrow$ extending operator $\triangle$ (see Lemma A.17.) Given the intuitiveness of interval types, we suggest that type systems of generic OOP languages which support wildcard types only to consider supporting interval types to enhance their type expressiveness.

4.1.1 Subtyping Rules for Interval Types. In a generic OOP type system that supports interval types, the core subtyping rules of the type system that are related to interval types will be as follows.

$$\frac{\mathtt{Null}_c \trianglelefteq \mathtt{Object}_c}{\mathtt{Null}_t <: \mathtt{Object}_t}\,Sub\mathbb{S}_0 \qquad \frac{\mathtt{C} \trianglelefteq \mathtt{D} \quad \mathtt{I} \sqsubseteq \mathtt{J}}{\mathtt{C[I]} <: \mathtt{D[J]}}\,SubGG$$

$$\frac{T_1 <: T_2 \quad T_3 <: T_4}{[T_2 - T_3] \sqsubseteq [T_1 - T_4]}\,Cont$$

where Rule $Sub\mathbb{S}_0$ is for constructing $\mathbb{S}_0$[16], Rule *SubGG* (for subtyping between two parameterized types) corresponds to the fourth line in the definition of the ordering relation underlying $\ltimes$ (see Definition A.9)[17], and Rule *Cont* corresponds to Equation (7) in the definition of $\Updownarrow$ (see Definition A.14)[18]. Note that the circular dependency between relation $\sqsubseteq$ and relation <: (in Rules *SubGG* and *Cont*) necessitates the existence of "a base rule" (such as $Sub\mathbb{S}_0$) to break the circularity

[15] Type `Null`, called there '*the null type*,' is actually used inside the Java type system, but since it has no name in Java it can be used only implicitly in Java programs (*i.e.*, it cannot be explicitly specified in developers' code) [43, §4.1].

[16] And, accordingly (using $\Updownarrow$ and Rule $Cont$), for also constructing the interval type argument `[Null − Object]` = [?] (corresponding to the wildcard type argument <?>), which is then used (by $\ltimes$) to construct 'free types' and, thus, also construct $\mathbb{S}_1$.

[17] Note that Rule $SubGG$ makes use of our second assumption on $\mathbb{C}$ in §3.1, namely, the assumption that if a generic classes extends another generic class, then the type parameter of the subclass is passed "untouched" as a type argument to its superclass.

[18] Note that in Rule $Cont$, the condition $T_2 <: T_3$ need not be specified, since the condition is built in the definition of interval $[T_2 - T_3]$.

and get the iterative construction of the two relations kick-started. The intuitiveness and simplicity of these subtyping rules is worthy of noting.

4.2 Doubly F-bounded Generics

Similar to having general lower and upperbounds of *type arguments* to generic classes (*i.e.*, having interval types), the order theoretic approach to modeling generic OOP also suggests having general lower and upperbounds of *type parameters* of generic classes. Given the connections of order theory to category theory, the approach further suggests how a type parameter may have a lower or an upper *F-bound*, thereby proposing what we call 'doubly F-bounded generics,' or *dfbg* for short.

To illustrate *dfbg*, we present basic examples of how F-bounded generics—generic classes with a type parameter that has either a lower F-bound or an upper F-bound—may be declared (written in some hypothetical future version of Java). The examples are then followed by a discussion of how *dfbg* can be mathematically modeled using constructs from order and category theory.

Example 4.2. Consider the following Java class declarations

```
class C<T> { ... }
class D extends C<D> { ... }
class E<T extends C<T>> { ... }
class F<T extends F<T>> { ... }
class G extends F<G> { ... }
class A { ... }
class B<T> extends A { ... }
class H<T super B<T>> { ... } // Hypothetical
```

In these declarations, type parameter `T` of class `C` ranges over all types, class `D` defines a homonymous type `D` as a `C`-subtype (since `D <: C<D>`), and, since it occurs inside its own bound, type parameter `T` of class `E` is upper F-bounded and it ranges over `C`-subtypes (*i.e.*, the parameterized type `E<D>`, for example, is a valid instantiation of class `E`, since `D` is a `C`-subtype).

Further, like in the declaration of built-in class `Enum` in Java, type parameter `T` of class `F` is also upper F-bounded and it ranges over `F`-subtypes, making the declaration of class (and type) `G`, as extending `F<G>`, a valid declaration.

Also in the above, the declaration of class `B` makes type `A` a `B`-supertype (since `B<A> <: A`), then, finally, the (hypothetical) declaration of class `H` specifies that type parameter `T` of class `H` is *lower* F-bounded and that it ranges over `B`-supertypes, and thus that types `H<A>` and, implicitly, `H<Object>` (since `Object` is an implicit supertype of `A`) are valid parameterized types.[19]

4.2.1 Modeling (Doubly) F-bounded Generics: Inductive and Coinductive Types in Generic OOP.

While analyzing *dfbg* in [10], we used a *coinductive* logical argument[20] to prove that checking the validity of type arguments inside some particular bounds-declarations of generic classes is unnecessary. Also, in [79], Tate et al. conclude that Java wildcards are some form of '*coinductive* bounded existentials'.[21]

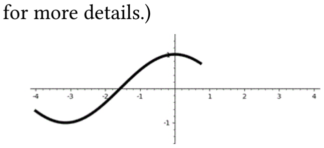
Combined, these two factors motivated us to consider, in some depth, the status of inductive and coinductive types in our order-theoretic approach [15], which led us to define the notions of 'F-subtypes' and 'F-supertypes' of a generic class F (see §A for the definitions of these notions), and, accordingly, to offer an order theoretic and category theoretic model of *dfbg*.

More formally, as hinted at in Example 4.2, the value of defining 'F-subtypes' and 'F-supertypes' in the modeling of *dfbg* is clear when we note that a type parameter, say `T`, with a *lower* F-bound ranges over the set of 'F-supertypes of the erasure of the lowerbound', and, dually, a type parameter with an *upper* F-bound ranges over the set of 'F-subtypes of the erasure of the upperbound.'

It should be immediately noted, however, that further investigation is needed on the practical value of having lower F-bounds[22], and on whether it is sensible to simultaneously have both a lower *and* an upper F-bound for the same type parameter `T`.

We keep that investigation to future work. Nevertheless, we believe *dfbg* should be considered as an extension of generic OO type systems that smoothly and intuitively goes hand-in-hand with considering supporting interval types, since we believe the general notion of intervals (*e.g.*, from analysis) is much simpler and more intuitive to mainstream OO developers than the notion of (bounded) existentials. Thus, for consistency and homogeneity purposes, if generic type arguments can have lower and upper bounds (*i.e.*, if

[19] It is worthy to note that our formulation of *dfbg* and our approach to modeling F-bounded generics got inspiration from considering functions in real analysis. For example, check the real function $\cos(x \le \cos(x))$ having a 'function-bounded domain' (illustrated below). Due to the *second* occurrence of cos (in the "f-bound" of parameter x), variable x (the first, binding occurrence of x) ranges only over real numbers that are post-fixed points of the cos function (hence the "left-handedness" of the graph of the function illustrated below). However, as is arguably intuitively clear (and as we coinductively prove in [10]), the second occurrence of cos, unlike the first occurrence, can be viewed as having *all* real numbers $\mathbb{R}$ as its domain, with no theoretical or technical difficulties resulting. (Interested readers may like to check [10, 17] for more details.)



[20] In mathematical logic, *coinductive reasoning* can, intuitively, be summarized as asserting that a statement is proven to be true if there is *no* (finite or "good") reason for the statement to *not* hold [16, 54].

[21] Given their historical origins [53, 77], induction and coinduction—and accordingly (co)inductive mathematical objects—are naturally best studied in lattice theory (as a sub-field of order theory).

[22] See [74] for some examples on the potential utility of having lower bounds for type parameters, and see [79] for some suggested restrictions on them.

interval types are supported) in generic OOP languages, then we conjecture that OO developers will expect type parameters to have lower and upper bounds, including F-bounds, too. Additionally, supporting *dfbg* may allow viewing interval type arguments (and thus also wildcard type arguments) as merely being special cases of doubly F-bounded type parameters—more specifically, viewing them as 'anonymous type parameters,' where the main difference between type arguments and type parameters, in such a view, will be that the former have no names (and thus cannot be referenced explicitly in any code) while the latter are named and, thus, can be referenced.

4.3 Default Type Arguments and Default Types

A third feature that is influenced by our order theoretic approach to modeling generics, if not directly suggested by it[23], is the notion of 'default type arguments' of a generic class, and the notion of the 'default type' corresponding to a generic class (which builds on the notion of default type arguments).

As we envision it, a *default type argument* (DTA) is a type that is specified explicitly in the declaration of a generic class (or is inferred[24]) as the type argument that should be used in instantiating the class (to define a parameterized type, as the *default type* corresponding to the class) when the class name *alone* is used in a context that needs a type (reference) rather than a class (reference).

We illustrate DTAs and default types using an example (again, coded in some hypothetical future version of Java).

Example 4.3. Consider the following Java class and variable declarations

```
class C<T=Object> { ... }
class D<T=Integer> { ... }

// Class names are used as type names
C c; // type of c is C<Object>
D d; // type of d is D<Integer>
```

In these declarations, the default type argument (DTA) for type parameter `T` of class `C` is specified, using the `T=Object` phrase, as type `Object`, while the DTA of type parameter `T` of class `D` is specified as type `Integer`. Thus, when the class names are used in a context where a type name is expected, as in the declarations of variables `c` and `d`, these variables have the default types resulting from instantiating the specified classes with their default type arguments, namely, `C<Object>` and `D<Integer>`, respectively.[25]

If default type arguments and default type are supported in a generic OOP language, then the *erased type* corresponding to a parameterized type (*i.e.*, the type resulting from the "erasure" of the parameterized type) can be defined as the composition of the *erasure* mapping Er (which maps the parameterized type to a class—see §3.2 for more on Er) with the *default type* mapping Dt (which maps a class to its corresponding default type).

Formally, we thus define the *erased type* mapping as $Et\,(\mathtt{T}) = Dt\,(Er\,(\mathtt{T}))$ for all parameterized types `T`, which, in point-free notation, can be expressed as

$$Et = Dt \circ Er.$$

In words, the definition of Et defines the erased type of a parameterized type as 'the default type of the erasure of the parameterized type.'

We believe supporting default types, and default type arguments upon which default types are based, is a simple extension of most if not all current generic OO type systems.[26]

4.4 Cofree Types

In §2 we briefly mentioned 'cofree types' as being the greatest (or largest) F-subtypes (*i.e.*, as the largest types that are subtypes of their own instantiations of a generic class `C`; or, formally, the greatest type `T` where `T <: C<T>`).

No generic OOP language that we know of explicitly supports cofree types so far, arguably for good reasons. Nevertheless, motivated by the earlier discussion of free types and of the EGC, we propose that generic OO type systems support cofree types using, for example, a syntax such as `C<!>` or `C[!]` to denote the cofree type corresponding to a generic class `C`. In accordance with the definition of cofree types (see §A), a cofree type, say `C<!>`, will be a subtype of all parameterized types that are instantiations of class `C`, and, semantically, `C<!>` will have the special value `null` as its *only* instance.[27] With such semantics, we will have (*e.g.*, in a future Java type system that supports cofree types) the following subtyping hierarchy

[23] Consideration of default types and default type arguments (DTAs) where urged on us, in particular, by our consideration of the 'free type' corresponding to a class and our consideration of the 'erasure Galois connection' (EGC). See §A and §3.2 for more on these.

[24] In the case of Java in particular, where legacy non-generic code of some extant generic classes (*e.g.*, the collection classes) may be available, legacy default type arugments of a "generified" class (with no default type arguments specified) can usually be guessed (*e.g.*, by static analysis tools) by doing a code comparison, *i.e.*, by comparing legacy non-generic code of the class with the generic code of the class.

[25] We also envision that if the default type argument of a generic class is not specified, then the default type argument of the generic class can be inferred (as hinted to earlier, if a legacy non-generic version of the class is available) or—in agreement with the current specification in Java [43] for erased types—the default type argument can be the upperbound of the type parameter [43, §4.6](*i.e.*, the upperbound of the parameter, in case no DTA is specified, will play the role of the "default default type argument!")

[26] In fact, based on their perceived simplicity, it is surprising to us that the two features seem to have not been suggested before for generic OO type systems.

[27] In other words, the type `C<!>`, when used in some context, will be associating a specific class with for the polymorphic value `null`, viewing `null` as an instance of class `C` in that context.

```
Null <: C<!> <: C<Ty> <: C<?> <: Object
```

for a generic class `C` and for all its valid type arguments `Ty` (or even for all parameterized types `Ty`, assuming the type system also supports admittible type arguments. See §6 for more on admittible versus valid type arguments).

While initially seeming mysterious and not quite useful, cofree types actually seem to be currently supported, indirectly, in Java. It should be noted, for example, that, when the free type `C<?>` is used as a *lower* bound of a wildcard type argument (lower bounds on type parameters are not currently supported in Java), the actual meaning of `C<?>` in this context is rather closer to the meaning (*i.e.*, semantics) of `C<!>` that we suggest than it is to the standard meaning of `C<?>` (*i.e.*, the meaning when `C<?>` is used as an *upper* bound, or when it is used as the type of a regular variable, for example). We thus conjecture that cofree types like `C<!>` may indeed be useful in generic OOP, at least as proper lower bounds of type arguments, and as lower bounds of type parameters (if *dfbg*—see §4.2—is supported).

We illustrate this potential use of cofree types using the following example.

Example 4.4. If cofree types (and *dfbg*) are supported in Java, and assuming `C` is some generic class, then the following is a generic class declaration that makes use of the cofree type `C<!>` as a lower type parameter bound.

```
class D<T super C<!> extends C<?>> { ... }
```

In this declaration, the use of the cofree type `C<!>` as a lowerbound of type variable `T`, combined with using the free type `C<?>` as its upperbound, specifies that type variable `T` ranges only over parameterized types that are instantiations of `C`, not of its subclasses (if any).

Example 4.5. Similarly, if interval types arguments are supported in Java, and if `C` and `E` are two already-declared generic classes, then the wildcard type

```
E<? super C<!> extends C<?>>
```

is a supertype of all instantiations of `E` that are themselves instantiations of class `C` only.

In spite of these possible uses of cofree types, we do acknowledge, however, that a more thorough analysis of how cofree types interact with the rest of a generic OOP type system may be needed before explicit support for them can be added to generic OOP languages.

5 Related Work

In this section we present a rough account of some earlier research that is somewhat closely related to the work we present in this paper.

The modeling of generic OO type systems based on existential types has its roots in the work of Igarashi and Viroli [48, 49], which is the first work to suggest using Cardelli and Wegner's bounded existential types [30][28] to model 'variant parametric types' (VPTs). Igarashi and Viroli developed VPTs to 'enhance the synergy between parametric and inclusion [*i.e.*, subtyping] polymorphism in object-oriented languages,' stating in [48] that VPTs were 'inspired by *structural virtual types* by Thorup and Torgersen' [80].

With *structural* typing—the standard form of typing in functional programming (FP) languages—clearly on their minds, Torgersen at al. then presented the first operational model of Java wildcards [81], viewing wildcard types as being 'a form of bounded existentials.'

Benjamin Pierce (with others) was one of the first to point to the significance of *nominal* typing in mainstream OOP, and one of the first to present operational models of nominally-typed OOP ([65, §19.3] and [47, 52]). Subsequent work highlighting the significance of the nominal subtyping versus structural subtyping distinction includes the work of Malayeri and Aldrich in [57], in which the authors attempt to provide a foundation for integrating both forms of subtyping.

The motivation behind the development of most operational models of generic OOP was to attempt proving the type soundness of generic OO type systems, to ponder over the decidability of type checking, and to suggest decidable "chunks" of the type systems. As such, in [79], Tate et al. propose the taming of Java wildcards by suggesting restrictions on the usage of wildcards. Tate et al.'s work was based on operational models such as FJ/FGJ [47] and on the earlier models presented in [24, 25, 76, 79, 81]. In their work, Tate at al. conclude that Java wildcards are 'best formalized as coinductive bounded existentials.' Later, in [78], Tate suggested adding support for declaration-site variance annotations to Java, supporting thereby 'mixed-site variance' to 'avoid the failings of wildcards.'

In spite of the prevalence of operational models in researching OOP type systems, in [68–70], particularly in [69], the authors develop an *untyped* denotational model of class-based (non-generic) OOP. Type information is largely ignored in this work (object methods and fields have no type signatures) and some nominal information is included with objects only to analyze OO dynamic dispatch. In 2011, the first domain-theoretic model of (non-generic) nominally-typed OOP (called *NOOP*) was constructed by AbdelGawad [4, 5] and it included nominal typing information in full. (*NOOP* and its construction are summarized in [7, 20].) The goal behind developing *NOOP* was to help move research on mainstream OO type systems, that are largely nominally-typed, to a more foundational, denotational level, rather than an operational one. Using *NOOP*, Cartwright and AbdelGawad

[28] Cardelli, in his pioneering models of OOP [27, 28], ignored the nominality of subtyping for the sake of simplifying his models (since, if class and type names and their matching [*i.e.*, subsumption] rules are modeled, 'many complex issues arise' [29, p.2].)

concluded that 'inheritance *is* subtyping' in (non-generic) nominally-typed OOP [31].

Attempts to use category theory in modeling generic (or, "polymorphic") OO type systems seem to predate using existential types in modeling them. To the best of our knowledge, Canning et al.[26, §5] seem to be the first to reference F-algebras in the context of modeling OOP, and to thus suggest F-bounded polymorphism as a model of polymorphic OO type systems. Even though Pierce, and others, have analyzed F-bounded polymorphism from a foundational perspective (again, with a focus on decidability issues) [22, 39, 44, 64], but the explicit reference to the category theoretic roots of F-bounded polymorphism seemed to have been missed in this later work on modeling generic OOP.

Other than this earlier work, it seems to us that the use of order/lattice theory and category theory to model generic OO type systems has not been pursued before.

Category theory, though, seems to have also been used in modeling OOP, not in modeling generic OO type systems, but rather in modeling the runtime termination behavior of OO software [50, 51] and even in modeling subtyping in *non*-generic Java using coalgebraic specifications [66]. Final coalgebras were seemingly also used to model infinite data structures (such as streams and infinite trees), unending processes, and "systems" with dynamic state [21, 72][29]. It seems to us this earlier work did not use category theory to model generic OO type systems because the work was carried out before the formal introduction of generics and wildcards to Java [41, 82] and other mainstream OOP languages.

Galois connections have been used before in studying the semantics of programming languages, in the context of static analysis (particularly in abstract interpretation [34]) and axiomatizing temporal logic [21, Ch.9], but seemingly not outside these contexts.

More recently, given that operads in category theory can be used to model self-similar phenomena [75], AbdelGawad has presented an outline of the Java Subtyping Operad (*JSO*) as an operad that models the iterative construction of the subtyping relation in generic Java. (This earlier work represented a significant step in the development of the work presented in the current paper.) Most recently, it is worthy of mention that an extended abstract of the approach presented in this paper was accepted for poster presentation at ACT'19 [17, 19].[30]

[29]In fact it seems that [21], even though it makes no mention of inclusion/subtyping type polymorphism or generics and little mention of parameteric type polymorphism, is to date one of the best presentations of applications of order theory and category theory—both of main interest to us in this paper—to computer science, particularly their applications in the field of (automated) construction of programs from their (algebraic) specifications.

[30]An updated version of [17] is available at http://eng.staff.alexu.edu.eg/~moez/research/OOP/gen-act19.pdf.

6 Discussion, Concluding Remarks, and Future Work

In this paper we presented an order-theoretic approach to modeling generic OO type systems. The approach, as presented, demonstrates that in generic OO type systems:

- The infinite subtyping relation between ground parameterized types can be constructed (using tools from order theory) exclusively based only on the finite, nominal[31], explicitly-specified subclassing (*i.e.*, inheritance) relation between classes,
- Type erasure can be modeled as a mapping from parameterized types ordered by subtyping to classes ordered by subclassing,
- Due to the nominality of subclassing and the nominality of subtyping (since subtyping is based on subclassing), the erasure of parameterized types (*i.e.*, the class used to construct a parameterized type) and the free types corresponding to classes (*i.e.*, the greatest F-subtype of a class), together, define a Galois connection between subtyping and subclassing,
- The Galois connection between subtyping and subclassing formally expresses the fundamental property of generic OOP that 'inheritance is the only source of subtyping,' *i.e.*, that inheritance (*a.k.a.*, subclassing, between classes) is the only source of subtyping (between parameterized types),
- Wildcard type arguments can be modeled intuitively as intervals over the subtyping relation, ordered by interval containment,
- Generic classes and type constructors can be modeled as generators over the subtyping relation, *i.e.*, as mappings that take in type arguments (ordered by containment) and construct parameterized types (ordered by subtyping),
- Upper F-bounded type variables[32] range over F-subtypes, which can be modeled as post-fixed points or coinductive types, while lower F-bounded type variables range over F-supertypes, which can be modeled as pre-fixed points or inductive types,
- Using the containment relation between generic (*i.e.*, wildcard or interval) type arguments, the complex `open` and `close` operations (*i.e.*, capture conversion; see [42, 43, §5.1.10, p.113]) are *not* needed in the definition of the subtyping relation between ground parameterized types,
- During type checking (*e.g.*, while compiling of a generic OO program), it may be not necessary to check for the

[31]Since it is always explicitly declared using class *names*, inheritance/subclassing is an inherently nominal relation.

[32]*E.g.*, type variable `T` in the Java class declaration `class D<T extends C<T>>`.

validity of the type argument of the bound of an F-bounded type parameter[33],

- Deriving the subtyping relation from the subclassing relation implies that properties of the infinite and intricate generic subtyping relation can be derived exclusively from properties of the finite and simpler subclassing relation. As such, errors in an OO program related to generic subtyping can be explained in terms of the subclassing/inheritance relation. Additionally, given that the order-theoretic approach to modeling generics does not explicitly use existential types, these explications (*e.g.*, in compiler error messages) can make use of no concepts related to existential types (*e.g.*, "capturing"), and that,
- Noting that the inheritance relation is directly and explicitly specified by OO developers, and that existential types and notions related to them (*e.g.*, `opening`, `closeing`, and "capturing" them) are unfamiliar to most OO developers, it can be conjectured that using the explicitly-specified inheritance relation when reasoning about generics, as well as avoiding the explicit use of any notions related to existential types, may strongly motivate OO developers to make better and more confident use of generics while developing their OO software.

Additionally, we observe that extant structural (*i.e.*, non-nominal) models of generic OOP largely ignore the nominal subclassing relation—explicitly declared by OO software developers—when interpreting the generic subtyping relation and other features of generic OOP. As discussed in §5 on related work, those models—influenced by their origins in functional programming—depend instead on concepts and tools[34] developed for structural typing and structural subtyping.

As such, we conclude that the inclusion of nominality in any models of generic OO type systems is key to constructing a simpler, more intuitive and more natural model of such type systems, which, as we demonstrated in this paper, is due to the reliance of the definition of the infinite subtyping relation on the finite inherently-nominal subclassing relation.

Even though a full order-theoretic model of generic OO type systems has not been constructed yet (we discuss below plenty of the future work that remains to construct such a model), yet, given the insights into generic OO type systems presented above, we believe that the order-theoretic approach to modeling these type systems has the potential to offer a model of generic OOP that is more in the spirit of nominally-typed OO type systems than the existing models (based on existential types) are, and that is also significantly simpler and more intuitive than extant models. Due to the centrality of generics to modern OO type systems, we also conjecture that having a simple model of generics, like the one promised by the order theoretic approach to modeling generics, will enable researching and progressing mainstream nominally-typed OOP languages on firmer grounds.

6.1 Future Work

As expressed in the title of this paper, and as is evident from our presentation of the approach in this paper, we believe the order and category theoretic approach to modeling generic OO type systems we present in this paper is far from being finished or complete (without this incompleteness contradicting in any way the potential the approach has; in fact quite the opposite). In particular, we believe a complete order and category theoretic model of generic OO type systems cannot be constructed unless most (if not all) of the following issues are properly addressed and the missing details in the model are appropriately filled in.

Firstly, throughout this paper we have been, intentionally, unclear about whether we assume type parameters of generic classes to have general lower and upper bounds, or whether (like some other research on generics does, *e.g.*, for simplicity) we assume that type parameters are "unbounded" (more accurately, are upper bounded by type `Object` and lower bounded by type `Null`). In fact we believe both!

In particular, motivated by our consideration of what we may call "self F-bounded type parameters" (*e.g.*, like in the declaration **`class`** `Enum<T` **`extends`** `Enum<T>>`, where not only is type parameter `T` used in defining its own bound, thereby making `T` be F-bounded, but *also* the class being defined, namely `Enum`, is itself used in defining the bound—see §4.2 for more details), we got into deliberating that there are *two* related kinds of type arguments that a generic class may be passed (*i.e.*, to instantiate it, so as to define a parameterized type), namely, *admittible type arguments* and *valid type arguments*, where valid type arguments are a proper subset of admittible ones.

To illustrate the main difference between the two kinds of type arguments, we envision that the set of *all* parameterized types (*i.e.*, all members of poset $\mathbb{T}$ in §3) defines the set of *admittible* type arguments to a generic class, *i.e.*, that can be passed to the class as parameters *regardless* of them satisfying the bounds specified on type parameter(s) of the class. On the other hand, *valid* type arguments of a generic class will then be defined as the subset of all parameterized types (*i.e.*, of admittible type arguments) that, in addition, satisfy the bounds specified on the type parameters of the class. (For example, according to these definitions, type `Object` is an admittible type argument to class `Enum`, but it is not a valid type argument [since type `Object` cannot be a subtype of

[33] Since, based on a coinductive logical argument, we can mathematically prove that a definition of a function-bounded function that uses the defined function itself in the bound specification defines the *same* function as one that uses the unbounded function in the bound specification. See §4.2 and [10] for more details.

[34] Such as existentials, abstract datatypes, and the opening/closing of type "packages."

any other type but itself].) As such, valid type arguments are class-specific (*i.e.*, vary by class), while admittible ones are not since, for all classes, they are simply the set of all (syntactically well-formed) parameterized types.

However, we further noted that distinguishing admittible type arguments versus valid ones, while intuitive and warranted, also necessitates defining *admittible (parameterized) types* (*e.g.*, types `Enum<Object>` and `Enum<Color>`) and *valid (parameterized) types* (`Enum<Color>` is valid, but `Enum<Object>` is not since `Object` is not a valid type argument), and, thus, also defining *admittible subtyping relations* (that involve at least one admittible parameterized type, *e.g.*, `Enum<Object> <: Enum<? extends Object>` and `Enum<Color> <: Enum<? extends Object>`) versus *valid subtyping relations* (involving only two valid types, *e.g.*, `Color <: Enum<Color>`). Given that we concluded that the 'admittible versus valid' distinction will permeate any model of generics, we decided (for our current effort) to stop at this point, and to keep the pursuit of the distinction, in full extent, to some future work.

Secondly, in §4.1.1 we presented the core subtyping rules for a generic OO type system that supports interval types (as a generalization of wildcard types). Including type variables in the subtyping rules, then defining the syntax, typing rules and evaluation rules of some tiny generic OO language that (along the lines of FGJ [47], for example) contains core features of generic OO type systems can be used to prove the type soundness of such a tiny generic OOP language. Due to time (and space) considerations, we've also decided that including type variables in the subtyping rules, then proving type soundness of a tiny generic OO language, will also have to be a venture that can be embarked upon in some future work.

Thirdly, a somewhat technical concern that arises for the order theoretic approach is that the third line in the definition of the mapping f of wildcards (modeling wildcard type arguments) to intervals (modeling interval type arguments)—*i.e.*, the third line of Equation (8) in §A—involves using a wildcard *twice*, both as an upper bound and a lower bound, in the definition of the corresponding interval type argument, which has the potential to cause subtyping relations for wildcards (ones based on existentials) to differ from those for corresponding intervals (based on interval containment).

We believe that this concern, while not addressed in our main work, can be addressed, if substantiated, in some future work either by: (1) modifying the definition of intervals and/or containment rules for intervals, (2) by adding a notion of *nominal intervals* (by which an interval can have a name, and where accordingly two nominal intervals—*i.e.*, intervals with names—are considered equal only if they have equal bounds and also have the same name. Anonymous intervals—with no name—will be compared based only on their bounds. See [8] for some more details on nominal intervals), or, (3) by abandoning (or deprecating) the existentials-based wildcard containment rules (considering these rules as "a failure" [78], at the risk of possibly thereby causing some backward compatibility issues to Java and extant generic OOP languages[35]) and supporting the simpler and more intuitive intervals-based containment rules instead.[36] For simplicity, we have chosen to pursue neither of these alternative options in this work however. Yet we believe this concern should be properly addressed—using either of the three suggested means, or some other ones—in any future work that builds on the work presented in this paper.

Fourthly, in §3.1 we explicitly assumed that generic classes pass their type parameter "as is" to their generic superclasses. While this is a common inheritance pattern among generic classes, but the assumption in our model precludes some other inheritance patterns (*e.g.*, `class C<T> extends D<F<T>>`). Based on first defining admittible and valid type arguments (the first future work suggested above), we believe that other more complex inheritance patterns can simply be modeled by defining a partial product poset constructor $\ltimes$ that is more complex than the straightforward one we defined in this paper.[37] In particular, such a new poset constructor will *not* construct new parameterized types (*i.e.*,

[35] For such languages, to avoid any backward compatibility issues (if they turn out to be expensive), it may be lesser expensive to support *both* containment rules: the standard-but-deprecated existentials-based wildcard containment rules, using the standard <> syntax for type arguments, together with the new similar-but-more-intuitive intervals-based rules, using the `[]` syntax, for example.

[36] If interval types (and thus also *type intervals*, which we called 'interval type arguments' in this paper) are supported in a generic OO type system, there will be contexts in a generic OO program that expect types (contexts such as field types, local variable types, method argument types, and method return types) and other contexts that expect type intervals (such as "type" arguments to other classes). Contexts where a *type* is expected can be divided into either covariant, contravariant, or invariant type contexts. For the type system to be *type safe*, we envision that when a type interval is provided (*e.g.*, as a type variable) in a context that expects a type (rather than a type interval) then the *lowerbound* of the provided type interval (which is a type, not an interval) can be used in contravariant type contexts (*e.g.*, method argument types) while the *upperbound* of the provided type interval (also a type) can be used in covariant contexts (*e.g.*, method return types). For invariant contexts (*e.g.*, field or variable types), we envision making them into ones that expect an interval instead, then require that when a field or variable is *read from* (*i.e.*, is used in an "r-position" or a read context, *e.g.*, on the right-hand side of an assignment statement) then the upperbound of the type interval of the field or variable is used (in type checking) while requiring that the lowerbound is used when the field or variable is written into (*i.e.*, is used in an "l-position" or a write context, *e.g.*, on the left-hand side of an assignment statement). While this suggestion is plausible (*e.g.*, it agrees with "views" of APIs of generic classes with declaration-site or use-site variance annotations), we believe it is in need for a more thorough investigation in some future work, *e.g.*, in the context of some tiny generic OO type system (our second suggested future work above).

[37] A main purpose behind the simple definition of $\ltimes$ used in this paper is to make evident the involvement of a partial product (of posets) in the definition and construction of parameterized types (see Equation (1), Equation (2), Equation (5), and Equation (6) in §3.1), even if at the price of not handling all generic class inheritance patterns except the most straightforward ones.

ones that were not constructed by $\ltimes$ in the solution of Equation (1) in §3.1 or of Equation (5) in §4.1) but will only affect the ordering relation between the constructed types (based on the declared inheritance patterns).[38]

Further, in §3.1 we assume that a generic class takes only one type parameter. Based on the relative simplicity of using a list of type parameters in place of a single type parameter, we believe this simplifying assumption can be easily relaxed, in any future work that builds on the work presented in this paper, to allow generic classes to have multiple type parameters.

Next, it is well-known that subtyping relations with circular (*i.e.*, self-referential, involving direct self-references) justifications can be viewed as infinitely-justified subtyping relation and can thus be modeled by a *coinductive* interpretation of the subtyping relation [52]. In light of the use of coinductive types (*i.e.*, F-subtypes) and the use of a coinductive logical argument in analyzing *dfbg* (§4.2), circular subtyping relations may motivate some future work to consider defining poset $\mathbb{T}$ (of parameterized types and the subtyping relation between them) *coinductively*, *i.e.*, as the greatest fixed point of Equation (1) in §(3.1), thereby allowing $\mathbb{T}$ to contain parameterized types that are possibly infinite (*i.e.*, infinitely nested) as well as subtyping relations that are possibly infinitely-justified.

While direct self-references are common in OOP, even more common are *indirect* self-references, where, for example, some class may refer to itself only indirectly by it referring to some other class (which may refer to a third class, and so on) that eventually refers back to the first class[39]. Such definitions and dependencies are sometimes called *mutually circular definitions* or *reciprocal dependencies*. In similitude to how direct self-references can by modeled using coinduction and coinductive objects, indirect self-references may be modeled using *mutual coinduction* and *mutually coinductive* objects.

While *mutual coinduction* is not widely used (as of yet) in the semantics of programming languages, indirect self-references in OOP and observing the mutual dependency between the definition of the subtyping relation on parameterized types and the definition of the containment relation on wildcard/interval type arguments (see §3 and §4), motivated us to define an order-theoretic notion of *mutual (co)induction* (rather than a power set theoretic one [62][40]) to allow studying least and greatest fixed point solutions of mutually-recursive definitions in a more abstract order-theoretic context [18]. Some future work may then consider pursuing the order theoretic treatment of mutual (co)induction further than we did, then use it as a model of indirect self-references between classes in generic OO software and, equally importantly, also use it in modeling the mutual dependency between the containment and the subtyping relations in generic OO type systems.[41]

Finally, concerning the use of category theory, in the order theoretic approach to modeling generics we brought up some concepts and tools from category theory (such as adjunctions, monads, F-(co)algebras, initial algebras, final coalgebras) that can be used to generalize the order-theoretic model of generics, and to situate it further in the context of category theory, yet, for simplicity, we also tried to somewhat keep the role of category theory in the approach to a minimum. But in fact doing so may not be necessary, or even recommended, in any future work that builds on the work presented in this paper.

That's because we believe that order theory, while definitely simpler and more intuitive than category theory, may not be as unifying and powerful (*i.e.*, revealing of underlying commonalities) as category theory is. For example, we believe it is possible to present the approach, first (order theoretically) under the more general umbrella of 'closure operators' (where, for example, free types will be 'closed types' since they are fixed points of the closure operator defined by composing the *free type* map of the EGC with its *erasure* map. See §3.2), then generalizing the presentation to use *monads* of category theory (as generalizations of closure operators). We conjecture that a monadic, category-theoretic treatment of generics, in some future work, may allow revealing even further structures and insights underlying generic OOP type systems.

Also related to applying category theory, based on the earlier presentation of the outline of *JSO* as an operad for modeling the construction of the self-similar generic subtyping relation in Java (as discussed in §5), some possible future work may also consider joining the work presented in this paper with the *JSO* operad to possibly present a deeper category theoretic model of generic OOP type systems—one that may possibly use the language of higher operads and higher categories [56].

[38] Due to the possibility of nesting type variables, a notion of 'rank' may be also useful in the definition of such a constructor.

[39] *E.g.*, assuming the absence of primitive types in Java (*i.e.*, if Java is "purely OO"), the definitions of classes `Object` and `Boolean` in Java are *mutually dependent* on each other. That is because class `Boolean`, like all classes in Java, extends class `Object`, thereby depending on it; and reciprocally, without a primitive type `bool`, the fundamental `equals()` method inside the definition (and the public interface) of class `Object` will return a `Boolean`, making the definition of class `Object` thereby depend on class `Boolean`.

[40] In agreement with what Priestley states in [21, Ch.2] (same as [67]), we believe that 'Powersets are too nice! Programs built on pure set models cannot capture all the behaviours that one might wish. Ordered set models are richer.'

[41] Doing so will allow for full clarity on asserting, accurately, that generic classes are passed 'wildcard/interval type arguments' rather than 'parameterized types' as their proper arguments (*i.e.*, will settle, conclusively, the debate on whether generic classes are parameterized by types or by wildcards/intervals, by taking the side of wildcards/intervals).

In conclusion, in spite of its volume, we do not believe the amount of suggested future work that can possibly build on the approach presented in this paper, some of which we already acknowledge may need to be performed to fully validate the approach, diminishes in any way the potential the approach has—in fact quite the opposite. That's because most of the suggested future work points to concrete and specific suggestions as to how to implement the work and how to address particular issues. As such we believe the approach shows its potential, even further, by it suggesting how potential roadblocks may be handled and addressed appropriately, fully within the standard mathematical frameworks of order theory and category theory.

References

[1] 2015. *C# Language Specification, Version 5.0.*

[2] 2017. *International Standard ISO/IEC 14882:2017(E) - Programming Language C++.*

[3] 2018. *Kotlin Language Documentation, v. 1.2.*

[4] Moez A. AbdelGawad. 2012. *NOOP: A Mathematical Model of Object-Oriented Programming.* Ph.D. Dissertation. Rice University.

[5] Moez A. AbdelGawad. 2013. *NOOP: A Nominal Mathematical Model Of Object-Oriented Programming.* Scholar's Press.

[6] Moez A. AbdelGawad. 2013. An Overview of Nominal-Typing versus Structural-Typing in Object-Oriented Programming (with code examples). *arXiv:cs.PL/1309.2348* (2013).

[7] Moez A. AbdelGawad. 2014. A Domain-Theoretic Model of Nominally-Typed Object-Oriented Programming. *Electronic Notes in Theoretical Computer Science* 301 (2014), 3–19. https://doi.org/10.1016/j.entcs.2014.01.002

[8] Moez A. AbdelGawad. 2016. Towards an Accurate Mathematical Model of Generic Nominally-Typed OOP (Extended Abstract). *arXiv:cs.PL/1610.05114* (2016).

[9] Moez A. AbdelGawad. 2017. Towards a Java Subtyping Operad. *Proceedings of FTfJP'17, Barcelona, Spain (Extended version available at arXiv:cs.PL/1706.00274)* (2017). https://doi.org/10.1145/3103111.3104043

[10] Moez A. AbdelGawad. 2018. Doubly F-bounded Generics. *arXiv:cs.PL/1808.06052* (2018).

[11] Moez A. AbdelGawad. 2018. Induction, Coinduction, and Fixed Points: A Concise Comparative Survey. *arXiv:cs.LO/1812.10026* (2018).

[12] Moez A. AbdelGawad. 2018. Java Subtyping as an Infinite Self-Similar Partial Graph Product. *arXiv:cs.PL/1805.06893* (2018).

[13] Moez A. AbdelGawad. 2018. Partial Cartesian Graph Product. *arXiv:cs.PL/1805.07155* (2018).

[14] Moez A. AbdelGawad. 2018. Towards Taming Java Wildcards and Extending Java with Interval Types. *arXiv:cs.PL/1805.10931* (2018).

[15] Moez A. AbdelGawad. 2019. Induction, Coinduction, and Fixed Points in PL Type Theory. *arXiv:cs.LO/1903.05126* (2019).

[16] Moez A. AbdelGawad. 2019. Induction, Coinduction, and Fixed Points: Intuitions and Tutorial. *arXiv:cs.LO/1903.05127* (2019).

[17] Moez A. AbdelGawad. 2019. Modeling Object-Oriented Generics: A Lattice- and Category-Theoretic Approach. *Poster @ Applied Category Theory 2019 (ACT'19), Oxford University, London, UK* (2019).

[18] Moez A. AbdelGawad. 2019. Mutual Coinduction. *arXiv:cs.LO/1903.06514* (2019).

[19] Moez A. AbdelGawad. 2019. Using Category Theory in Modeling Generics in Object-Oriented Programming (Outline). *arXiv:cs.PL/1906.04925* (2019).

[20] Moez A. AbdelGawad and Robert Cartwright. 2018. NOOP: A Domain-Theoretic Model of Nominally-Typed Object-Oriented Programming. *arXiv:cs.PL/1801.06793* (2018).

[21] Roland Carl Backhouse, Roy L. Crole, and Jeremy Gibbons (Eds.). 2002. *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, International Summer School and Workshop, Oxford, UK, April 10-14, 2000, Revised Lectures.* Lectures Notes in Computer Science, Vol. 2297. Springer.

[22] Paolo Baldan, Giorgio Ghelli, and Alessandra Raffaeta. 1999. Basic Theory of F-bounded Polymorphism. *Information and Computation* 153, 1 (1999), 173–237.

[23] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. 1998. Making The Future Safe For The Past: Adding Genericity to The Java Prog. Lang.. In *OOPSLA'98*, Vol. 33.

[24] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. 2008. A Model for Java with Wildcards. In *ECOOP'08.*

[25] Nicholas Cameron, Erik Ernst, and Sophia Drossopoulou. 2007. Towards an existential types model for Java wildcards. *FTfJP'07* (2007).

[26] Peter S. Canning, William R. Cook, Walter L. Hill, J. Mitchell, and W. Olthoff. 1989. F-bounded Polymorphism for Object-Oriented Programming. In *Proc. of Conf. on Functional Programming Languages and Computer Architecture.*

[27] Luca Cardelli. 1984. A semantics of multiple inheritance. In *Proc. of the internat. symp. on semantics of data types* (Sophia-Antipolis, France), Vol. 173. Springer-Verlag, 51–67. http://portal.acm.org/citation.cfm?id=1096.1098

[28] Luca Cardelli. 1988. A Semantics of Multiple Inheritance. *Inform. and Comput.* 76 (1988), 138–164.

[29] Luca Cardelli. 1988. Structural Subtyping and the Notion of Power Type. In *ACM Proceedings of POPL.*

[30] Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys* 17, 4 (December 1985), 471–522.

[31] Robert Cartwright and Moez A. AbdelGawad. 2013. Inheritance *Is* Subtyping (Extended Abstract). In *The 25^{th} Nordic Workshop on Programming Theory (NWPT).* Tallinn, Estonia.

[32] Robert Cartwright, Rebecca Parsons, and Moez A. AbdelGawad. 2016. *Domain Theory: An Introduction.* arXiv:cs.PL/1605.05858.

[33] Robert Cartwright and Jr. Steele, Guy L. 1998. Compatible Genericity with Run-time Types for the Java Prog. Lang.. In *OOPSLA'98*, Vol. 33.

[34] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of 4th Symp. on Principles of Prog. Lang. (POPL).* 238–252.

[35] B. A. Davey and H. A. Priestley. 2002. *Introduction to Lattices and Order* (2nd ed.). Cambridge University Press.

[36] Herbert B. Enderton. 1977. *Elements of Set Theory.* Academic Press, New York.

[37] Brendan Fong and David Spivak. 2018. *Seven Sketches in Compositionality: An Invitation to Applied Category Theory.* Draft.

[38] Thomas Forster. 2003. *Logic, Induction, and Sets.* Cambridge University Press.

[39] Vladimir Gapeyev, Michael Y. Levin, and Benjamin C. Pierce. 2002. Recursive Subtyping Revealed. *Journal of Functional Programming* (2002).

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

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

[42] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. 2018. *The Java Language Specification.* Addison-Wesley.

[43] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. 2019. *The Java Language Specification.* Addison-Wesley.

[44] Ben Greenman, Fabian Muehlboeck, and Ross Tate. 2014. Getting F-bounded Polymorphism Into Shape. In *PLDI'14*.
[45] Paul R. Halmos. 1960. *Naive Set Theory.* D. Van Nostrand Company, Inc.
[46] Egbert Harzheim. 2005. *Ordered Sets.* Springer.
[47] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. on Prog. Languages and Systems* 23, 3 (May 2001), 396–450.
[48] Atsushi Igarashi and Mirko Viroli. 2002. On variance-based subtyping for parametric types. In *In ECOOP*. Springer-Verlag, 441–469.
[49] Atsushi Igarashi and Mirko Viroli. 2006. Variant Parametric Types: A Flexible Subtyping Scheme for Generics. *ACM Trans. on Prog. Lang. and Sys.* 28, 5 (September 2006), 795–847.
[50] Bart Jacobs. 1996. Objects and Classes, Coalgebraically. In *Object-Orientation with Parallelism and Persistence.* Kluwer Acad. Publ, 83–103.
[51] Bart Jacobs and Erik Poll. 2002. Coalgebras and Monads in the Semantics of Java. *Preprint submitted to Elsevier Science* (2002).
[52] Andrew J. Kennedy and Benjamin C. Pierce. 2007. On Decidability of Nominal Subtyping with Variance. In *International Workshop on Foundations and Developments of Object-Oriented Languages.*
[53] B. Knaster. 1928. Un Théorème sur Les Fonctions d'Ensembles. *Ann. Soc. Polon. Math.* 6 (1928), 133–134.
[54] Dexter Kozen and Alexandra Silva. 2016. Practical Coinduction. *Mathematical Structures in Computer Science* 27, 7 (2016), 1132–1152. https://doi.org/10.1017/S0960129515000493
[55] Angelika Langer. 2015. The Java Generics FAQ. http://www.angelikalanger.com/GenericsFAQ/.
[56] T. Leinster. 2004. *Higher Operads, Higher Categories.* Cambridge University Press.
[57] Donna Malayeri and Jonathan Aldrich. 2008. Integrating nominal and structural subtyping. In *ECOOP 2008–Object-Oriented Programming.* Springer, 260–284.
[58] Simon Marlow. 2010. *Haskell 2010 Language Report.*
[59] Andrew McLennan. 2018. *Advanced Fixed Point Theory for Economics.* Springer.
[60] R. Milner, M. Tofte, R. Harper, and D. MacQueen. 1997. *The Definition of Standard ML (Revised).* MIT Press.
[61] Martin Odersky. 2014. The Scala Language Specification, v. 2.9.
[62] Lawrence C. Paulson. 1995. Set Theory For Verification: II Induction and Recursion. *Journal of Automated Reasoning* 15, 2 (1995), 167–215.
[63] Benjamin C. Pierce. 1991. *Basic Category Theory for Computer Scientists.* MIT Press.
[64] Benjamin C. Pierce. 1994. Bounded Quantification is Undecidable. *Information and Computation* 112, 1 (1994), 131–165.
[65] Benjamin C. Pierce. 2002. *Types and Prog. Languages.* MIT Press.
[66] Erik Poll. 2000. A Coalgebraic Semantics of Subtyping. *Electronic Notes in Theoretical Computer Science* 33 (2000), 276 – 293. https://doi.org/10.1016/S1571-0661(05)80352-4 CMCS'2000, Coalgebraic Methods in Computer Science.
[67] Hilary A. Priestley. 2002. Ordered Sets and Complete Lattices: A Primer for Computer Science. In *Alg. and Coalg. Methods in the Math. of Program Construction.* Springer, Chapter 2, 21–78.
[68] Bernhard Reus. 2002. Class-based versus Object-based: A Denotational Comparison. *Algebraic Methodology And Software Technology, Lecture Notes in Computer Science* 2422 (2002), 473–488.
[69] Bernhard Reus. 2003. Modular Semantics and Logics of Classes, In Computer Science Logic. *Computer Science Logic (CSL'03), LNCS* 2803 (2003), 456–469.
[70] Bernhard Reus and Thomas Streicher. 2002. Semantics and Logics of Objects. *Proceedings of the 17th Symp. on Logic in Computer Science (LICS 2002)* (2002), 113–122.
[71] Steven Roman. 2008. *Lattices and Ordered Sets.* Springer.
[72] J.J.M.M. Rutten. 2000. Universal Coalgebra: A Theory of Systems. *Theoretical Computer Science* (2000).
[73] Dana S. Scott. 1976. Data Types as Lattices. *SIAM Journal of Computing* 5, 3 (1976), 522–587.
[74] Dan Smith and Robert Cartwright. 2008. Java Type Inference is Broken: Can We Fix It? *OOPSLA* (2008), 505–524.
[75] David Spivak. 2014. *Category theory for the sciences.* MIT Press.
[76] Alexander J. Summers, Nicholas Cameron, Mariangiola Dezani-Ciancaglini, and Sophia Drossopoulou. 2010. Towards a Semantic Model for Java Wildcards. *FTfJP'10* (2010).
[77] Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* 5 (1955), 285–309.
[78] Ross Tate. 2013. Mixed-Site Variance. In *FOOL '13: Informal Proceedings of the 20th International Workshop on Foundations of Object-Oriented Languages* (Indianapolis, IN, USA).
[79] Ross Tate, Alan Leung, and Sorin Lerner. 2011. Taming Wildcards in Java's Type System. *PLDI'11, June 4–8, San Jose, CA, USA.* (2011).
[80] Kresten Krab Thorup and Mads Torgersen. 1999. Unifying genericity. In *ECOOP 99–Object-Oriented Programming.* Springer, 186–204.
[81] Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. 2005. Wild FJ. In *Foundations of Object-Oriented Languages.*
[82] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. 2004. Adding Wildcards to the Java Programming Language. In *SAC.*

A Mathematical Background

In this section we present the main mathematical notions we use in constructing an order and category theoretic model of generics. Some of the constructs presented in this section are standard constructs in order theory and category theory, which we either use "as is" or only give them names that make them more intuitive for use in an OO context, while others were defined for the purposes of constructing our model. (For readers interested in more details, we present some relevant references in the text.)

A.1 Posets and Poset Constructors

Definition A.1 (Posets)**.** A *poset* (partially-ordered set) $\mathbb{P}$ is a pair of a set P, called the *universe* of $\mathbb{P}$, and denoted by $|\mathbb{P}|$, and a binary relation over P that is reflexive, transitive and antisymmetric (called the 'underlying ordering relation' of $\mathbb{P}$, and usually denoted $\leq_{\mathbb{P}}$) [35, 46, 71].

Example A.2. A familiar example of a *total* ordering relation (also called a *chain*, or a *linear order*) is the $\leq$ (less than or equals) relation on the set of integers $\mathbb{Z}$. (Relation $\leq$ is a *total* ordering because for all $m, n \in \mathbb{Z}$, either $m \leq n$ or $n \leq m$.)

Example A.3. Familiar examples of *partial* ordering relations underlying posets include the descendants ('is descendant of') and ancestors ('is ancestor of') relations on sets of humans (the existence of siblings forces both relations to be only partial orderings not total ones), as well as the scheduling relation between subtasks of a process (*e.g.*, when run on a multiprocessor or parallel computer).

Example A.4. Posets with underlying partial ordering relations that are of most relevance to this work include, first, the poset $\mathbb{C}$ of the *inheritance* relation (sometimes also called the *subclassing* relation, and is sometimes denoted by $\trianglelefteq$) on the set C, the universe of $\mathbb{C}$, of classes of an OO program, and second, the poset $\mathbb{T}$ of the *subtyping* relation (usually denoted by <:) on the set T, the universe of $\mathbb{T}$, of types of an OO program.

Definition A.5 (Tagging)**.** For a set X, define the +-tagged ('plus-tagged') set X_+ as

$$X_+ = \{(x, '+') \,|x \in X\},$$

and the −-tagged ('minus-tagged') set X_- as

$$X_- = \{(x, '-') \,|x \in X\}.$$

Example A.6. Sets X_+ are used to model "covariant wildcard type arguments" (*e.g.*, type arguments that, in Java, are defined using the '`? extends`' clause), while sets X_- are used to model "contravariant wildcard type arguments" (*e.g.*, those defined in Java using '`? super`').

In the following definitions, let $\mathbb{P}$ and $\mathbb{Q}$ denote two posets with universes P and Q and underlying ordering relations $\leq_{\mathbb{P}}$ and $\leq_{\mathbb{Q}}$, respectively. Further, let $S \subseteq P$ and let $S' = P \backslash S$. Operators +, ×, and \ denote the standard 'sum' (*a.k.a.*, 'disjoint union'), 'product[42],' and 'difference' operations on sets [36, 45], respectively. (See Definition A.11 and §3.1 below.)

Definition A.7 (Poset Products)**.** The *product* of $\mathbb{P}$ and $\mathbb{Q}$, denoted $\mathbb{P} \times \mathbb{Q}$, is the poset having the set

$$|\mathbb{P} \times \mathbb{Q}| = P \times Q$$

as its universe, and, for $(a, b), (c, d) \in P \times Q$, the ordering relation $\leq$ underlying $\mathbb{P} \times \mathbb{Q}$ is defined by

$$(a, b) \leq (c, d) \Leftrightarrow a \leq_{\mathbb{P}} c \wedge b \leq_{\mathbb{Q}} d.$$

Example A.8 (Tagging as a Product)**.** For a set X, the +-tagged set X_+ can be defined as the product of X with the singleton set $\{'+'\}$, *i.e.*, as $X_+ = X \times \{'+'\}$. Similarly, the −-tagged set X_- can be defined as $X_- = X \times \{'-'\}$.

Now we present three new, more advanced poset constructors.

Definition A.9 (Partial Products)**.** The *partial product* of $\mathbb{P}$ and $\mathbb{Q}$ relative to $S \subseteq P$, denoted $\mathbb{P} \ltimes_S \mathbb{Q}$, is the poset having

$$|\mathbb{P} \ltimes_S \mathbb{Q}| = S' + S \times Q$$

as its universe (*i.e.*, all elements of P that are in the subset S are paired with all elements of Q, then the pairs are added, via a disjoint union, to elements of S' to define the universe of $\mathbb{P} \ltimes_S \mathbb{Q}$). For elements $T_1, T_2 \in |\mathbb{P} \ltimes_S \mathbb{Q}|$, if we let $R = S \times Q$ then the ordering relation $\leq:$ underlying $\mathbb{P} \ltimes_S \mathbb{Q}$ is defined, according to the forms of T_1 and T_2, by

$$\begin{cases} T_1 \leq: T_2 \Leftrightarrow T_1 \leq_{\mathbb{P}} T_2 & T_1, T_2 \in S' \\ T_1 \leq: (c, d) \Leftrightarrow T_1 \leq_{\mathbb{P}} c & T_1 \in S', T_2 = (c, d) \in R \\ (a, b) \leq: T_2 \Leftrightarrow a \leq_{\mathbb{P}} T_2 & T_1 = (a, b) \in R, T_2 \in S' \\ (a, b) \leq: (c, d) \Leftrightarrow a \leq_{\mathbb{P}} c \wedge b \leq_{\mathbb{Q}} d & T_1 = (a, b), T_2 = (c, d) \in R \end{cases}$$

Example A.10. As we present in §3, operator $\ltimes$ can be used to model the pairing (in generic OOP type systems) of *some* classes (generic classes, in particular) with type arguments to construct types.

Definition A.11 (Wildcards)**.** The *wildcards poset* of a bounded poset $\mathbb{P}$ (*i.e.*, one with a top element $\top \in P$ and a bottom element $\perp \in P$), denoted $\triangle(\mathbb{P})$, is the poset having the set

$$|\triangle(\mathbb{P})| = P + (P \backslash \{\perp\})_+ + (P \backslash \{\top, \perp\})_-$$

as its universe, and for elements $W_1, W_2 \in |\triangle(\mathbb{P})|$, the ordering relation $\sqsubseteq_w$ (wildcard *containment*) underlying $\triangle(\mathbb{P})$ is defined by

$$\begin{cases} (T_1, '+') \sqsubseteq_w (T_2, '+') \Leftrightarrow T_1 \leq_{\mathbb{P}} T_2 & W_1 = (T_1, '+'), W_2 = (T_2, '+') \\ (T_1, '-') \sqsubseteq_w (T_2, '-') \Leftrightarrow T_2 \leq_{\mathbb{P}} T_1 & W_1 = (T_1, '-'), W_2 = (T_2, '-') \\ T_1 \sqsubseteq_w (T_2, '+') \Leftrightarrow T_1 \leq_{\mathbb{P}} T_2 & W_1 = T_1, W_2 = (T_2, '+') \\ T_1 \sqsubseteq_w (T_2, '-') \Leftrightarrow T_2 \leq_{\mathbb{P}} T_1 & W_1 = T_1, W_2 = (T_2, '-') \end{cases}$$

Example A.12. The (wildcard) type argument `Number` is *contained* in each of the following wildcard type arguments: '`? extends Number`', '`? extends Object`', '`? super Number`', and '`? super Float`'.

Definition A.13 (Notation)**.** For convenience, in the following definition (and in the rest of this paper) we use the notation $[l - u]$ to denote the pair (l, u), where l, u are elements of some poset $\mathbb{P}$. The $[\cdot - \cdot]$ notation (called the 'interval notation') will particularly be used to denote pairs that represent intervals over $\mathbb{P}$ whose *lowerbound* is l (the first component of the pair) and whose *upperbound* is u (the pair's second component), *i.e.*, where $l \leq_{\mathbb{P}} u$.

Definition A.14 (Intervals)**.** The *intervals poset* of a poset $\mathbb{P}$ (not necessarily bounded), denoted $\Updownarrow(\mathbb{P})$, is the poset having the set

$$|\Updownarrow(\mathbb{P})| = \{[p - q] \,|p, q \in P \wedge p \leq_{\mathbb{P}} q\}$$

as its universe, and for elements $[p_1 - q_1], [p_2 - q_2] \in |\Updownarrow(\mathbb{P})|$, the ordering relation $\sqsubseteq$ (interval *containment*) underlying $\Updownarrow(\mathbb{P})$ is defined by

$$[p_1 - q_1] \sqsubseteq_i [p_2 - q_2] \Leftrightarrow p_2 \leq_{\mathbb{P}} p_1 \wedge q_1 \leq_{\mathbb{P}} q_2. \quad (7)$$

Definition A.15 (Isomorphic Posets)**.** Posets $\mathbb{P}$ and $\mathbb{Q}$ are *isomorphic* if there exists a (set theoretic) isomorphism $f : P \rightarrow Q$ and, additionally, $p_1 \leq_{\mathbb{P}} p_2 \Leftrightarrow f(p_1) \leq_{\mathbb{Q}} f(p_2)$ for all $p_1, p_2 \in P$.

[42]Of pairings of *all* elements of the first set with all elements of the second.

Definition A.16 (Subposets)**.** Poset $\mathbb{Q}$ is a *subposet* (or *suborder*) of $\mathbb{P}$ if $Q \subseteq P$ and $\leq_{\mathbb{Q}}$ is a subset of $\leq_{\mathbb{P}}$ (*i.e.*, if $\leq_Q \subseteq \leq_P$). Equivalently, $\mathbb{Q}$ is a subposet of $\mathbb{P}$ if $Q \subseteq P$ and for all $q_1, q_2 \in Q$, $q_1 \leq_{\mathbb{Q}} q_2 \Rightarrow q_1 \leq_{\mathbb{P}} q_2$ (*i.e.*, elements of $\mathbb{Q}$ are related by $\leq_{\mathbb{Q}}$ only if, as elements of P, they are related by $\leq_{\mathbb{P}}$ in $\mathbb{P}$).

Further, poset $\mathbb{Q}$ is a *full subposet* of $\mathbb{P}$ if $Q \subseteq P$ and $\leq_{\mathbb{Q}}$ is equal to the restriction of $\leq_{\mathbb{P}}$ to elements of Q, or, equivalently, if for all $q_1, q_2 \in Q$, $q_1 \leq_{\mathbb{Q}} q_2 \Leftrightarrow q_1 \leq_{\mathbb{P}} q_2$ (*i.e.*, elements of $\mathbb{Q}$ are related if and only if they are related in $\mathbb{P}$).

Lemma A.17 ($\Updownarrow$ extends $\triangle$)**.** *For a bounded poset $\mathbb{P}$, the poset $\triangle(\mathbb{P})$ is isomorphic to a full subposet of $\Updownarrow(\mathbb{P})$.*

Proof. Let $w \in |\triangle(\mathbb{P})|$, and let $f : \triangle(\mathbb{P}) \to \Updownarrow(\mathbb{P})$ be the function that maps wildcards of $\mathbb{P}$ to intervals of $\mathbb{P}$ defined by[43]

$$f(w) = \begin{cases} [T - \top] & w = (T, '-') \\ [\bot - T] & w = (T, '+') \\ [T - T] & w = T \end{cases} . \tag{8}$$

First, we check, using the definitions of f, $\sqsubseteq_w$ and $\sqsubseteq_i$, that f preservingly maps the wildcard containment relation underlying $\triangle(\mathbb{P})$ to the interval containment ordering relation underlying $\Updownarrow(\mathbb{P})$, *i.e.*, that for every pair of elements $w_1, w_2 \in |\triangle(\mathbb{P})|$, we have $w_1 \sqsubseteq_w w_2 \Leftrightarrow f(w_1) \sqsubseteq_i f(w_2)$ (function f is then called an 'order-embedding' [35]).

Next, by inspecting its definition, function f is clearly an injection (a one-to-one function). If the codomain of f is restricted to its image $f(\triangle(\mathbb{P}))$, then f (with its codomain restricted to its range) is also a surjection (*i.e.*, an onto function), and thus is reversible, and hence an isomorphism from $\triangle(\mathbb{P})$ to $f(\triangle(\mathbb{P}))$. Furthermore, the domain of the inverse function f^{-1} is the set $f(\triangle(\mathbb{P}))$. As such, the poset $\triangle(\mathbb{P})$ of wildcards over $\mathbb{P}$ is isomorphic to its image poset $f(\triangle(\mathbb{P}))$ of intervals.

Finally, using the definitions of $\triangle$ and $\Updownarrow$, it is straightforward to confirm that the image $f(\triangle(\mathbb{P}))$ is a full subposet of $\Updownarrow(\mathbb{P})$, by confirming that $|f(\triangle(\mathbb{P}))| \subseteq |\Updownarrow(\mathbb{P})|$, then confirming that for all $w_1, w_2 \in |\triangle(\mathbb{P})|$, if $i_1 = f(w_1)$ and $i_2 = f(w_2)$ then we have

$$i_1 \sqsubseteq_i |_{|f(\triangle(\mathbb{P}))|} i_2 \Leftrightarrow i_1 \sqsubseteq_i i_2 \Leftrightarrow w_1 \sqsubseteq_w w_2$$

i.e., that any two elements of $|f(\triangle(\mathbb{P}))|$ are related (by interval containment) in $f(\triangle(\mathbb{P}))$ if and only if they are related (by interval containment) in $\Updownarrow(\mathbb{P})$, hence—by the first proof step—if and only if their two preimages are related (by wildcard containment) in $\triangle(\mathbb{P})$. □

Definition A.18 (Note on Lemma)**.** Given Lemma A.17, operator $\Updownarrow$ is said to *extend* operator $\triangle$. While preserving all elements of $\triangle(\mathbb{P})$, and the relations between them, the poset $\Updownarrow(\mathbb{P})$ of intervals (of $\mathbb{P}$) has no less elements than the poset $\triangle(\mathbb{P})$ of wildcards (of $\mathbb{P}$), and the elements of $\Updownarrow(\mathbb{P})$ are related in full consistence with relations between elements of $\triangle(\mathbb{P})$.

Example A.19. Operators $\ltimes$, $\triangle$ and $\Updownarrow$ are poset constructors that we newly defined for the purpose of constructing an order theoretic model of generics.[44]

Having presented posets and some order theoretic tools useful in constructing them, we next present the definitions of some notions that are useful in analyzing posets and relations between them. As we will see in §3 and §4, these following notions are also useful in suggesting extensions to generic OOP type systems.

A.2 Galois Connections, Pre-Fixed Points, and Post-Fixed Points

Like before, in the following definitions let $\mathbb{P}$ and $\mathbb{Q}$ denote two posets with universes P and Q and with underlying ordering relations $\leq_{\mathbb{P}}$ and $\leq_{\mathbb{Q}}$, respectively.

Definition A.20 (Galois Connections)**.** Two mappings $F : \mathbb{P} \to \mathbb{Q}$ and $E : \mathbb{Q} \to \mathbb{P}$ (sometimes written as $F : \mathbb{P} \rightleftarrows \mathbb{Q} : E$) define a *Galois connection* between posets $\mathbb{P}$ and $\mathbb{Q}$ if and only if for all $p \in P$, $q \in Q$

$$F(p) \leq_{\mathbb{Q}} q \Leftrightarrow p \leq_{\mathbb{P}} E(q) .$$

Mapping F is then called the *lower adjoint* (or, sometimes, the *left* or *free adjoint*) of the connection, while E is called its *upper adjoint* (or, sometimes, the *right* or *forgetful adjoint*).

Example A.21. Under the standard orderings of integers, $\mathbb{Z}$, and real numbers, $\mathbb{R}$, the mapping $e : \mathbb{Z} \to \mathbb{R}$ that embeds ("upcasts") integers into reals (*e.g.*, mapping 3 to 3.0 and 4 to 4.0) is a lower adjoint of the floor function $\lfloor\cdot\rfloor : \mathbb{R} \to \mathbb{Z}$ and is an upper adjoint of the ceiling function $\lceil\cdot\rceil : \mathbb{R} \to \mathbb{Z}$ [37, 67, 75].

Definition A.22 (CT [Category Theory] Terminology)**.** In the context of category theory, Galois connections (between two posets) are generalized to *adjunctions* (between two categories) [37, 63, 75], and lower/upper adjoints are called left/right adjoints, respectively.

Definition A.23 (Pre-/Post-Fixed Points)**.** A mapping $F : \mathbb{P} \to \mathbb{P}$ is called an *endomap* over poset $\mathbb{P}$ (since it maps $\mathbb{P}$ into itself). An element $p \in P$ is called a *fixed point* of an endomap $F : \mathbb{P} \to \mathbb{P}$ if $F(p) = p$. If $F(p) \leq_{\mathbb{P}} p$, then p is called a *pre-fixed point* of F. If $p \leq_{\mathbb{P}} F(p)$, then p is called a *post-fixed point* of F.[45]

[43] In an OOP context, function f will map wildcard type arguments to interval type arguments.

[44] For more details on operators $\ltimes$, $\triangle$ and $\Updownarrow$, check [12–14]. In [12–14], the three operators $\ltimes$, $\triangle$ and $\Updownarrow$ are not defined directly over posets but are defined, rather, over directed (acyclic) graphs (DAGs)—*i.e.*, over the 'Hasse diagrams' of posets.

[45] A fixed point of an endomap is necessarily both a pre-fixed point and a post-fixed point of the map. A pre-fixed point, though, may not necessarily be a fixed point, nor, dually, may a post-fixed point be a fixed point of the map. As clear from the definitions, a point that is simultaneously a pre-fixed point and a post-fixed point of some endomap is also a fixed point of the map.

Definition A.24 (CT Terminology)**.** In the context of category theory, where *endofunctors* generalize order theoretic endomaps, pre-fixed points generalize to *algebras* of an endofunctor F (*a.k.a.*, F-*algebras*) while post-fixed points generalize to *coalgebras* of endofunctor F (*a.k.a.*, F-*coalgebras*).

Definition A.25 (Set Theory Terminology)**.** In the context of (power) set theory, if $F : \wp(U) \to \wp(U)$ is a *monotonic* function over subsets of a set U ordered by inclusion $\subseteq$ (*i.e.*, if for all $X, Y \subseteq U$, $X \subseteq Y \implies F(X) \subseteq F(Y)$), then F is called a *generator* over U, and the pre-fixed points of F are then called (F-)*inductive sets* while the post-fixed points of F are called (F-)*coinductive sets* [65, Ch. 21] and [39].)[46]

Definition A.26 (OOP Terminology)**.** In the context of generic OOP, a generic class $G : \mathbb{T} \to \mathbb{T}$ maps parameterized types (ordered by subtyping, <:) to parameterized types (*i.e.*, maps types into themselves). As such, in homage to terminology used in category theory, we call a type $t \in |\mathbb{T}|$ a G-*subtype* if $t <: G(t)$, while t is called a G-*supertype* if $G(t) <: t$.[47]

Example A.27. For a very simple, almost trivial example, type `Object` is a G-supertype of any generic class G in a generic Java program.[48] In fact, since it is the top element of the subtyping relation, type `Object` is the *greatest* (sometimes also called the *largest*) G-*supertype*.

Definition A.28 (Free Objects and Cofree Objects)**.** In the context of order theory, *least pre-fixed points* and *greatest post-fixed points*, if they exist, are usually of special significance. In a complete lattice $\mathbb{L}$ (a fundamental object of study in lattice theory [35, 40, 67, 71, 73, 77]), these special points are guaranteed to exist for any monotonic endomap F defined over $\mathbb{L}$, and these points correspond precisely to *lfp*s—least fixed points—and *gfp*s—greatest fixed points—of the lattice [77].[49]

Similarly, in category theory, *initial algebras* and *final coalgebras* (of a functor F), if they do exist in some category, are usually objects that play a special role in the category. If two functors $L : \mathcal{A} \rightleftarrows \mathcal{B} : R$ define an adjunction between two categories $\mathcal{A}$ and $\mathcal{B}$, then initial algebras in the "richer" right category $\mathcal{B}$ (*i.e.*, of the composite endofunctor $L \circ R : \mathcal{B} \to \mathcal{B}$, if they exist) define what are called *free objects*. For example, the 'free monoid' corresponding to a set X is the monoid of finite sequences of elements of X, while a *quiver* is the 'free category' corresponding to a directed graph [75].

Similar to initial algebras defining free objects, we stipulate that final coalgebras, if they exist in some category (in the context of some adjunction), define the *cofree objects* of the category.

Definition A.29 (Free Types and Cofree Types)**.** In the context of generic OOP, again in homage to terminology used in category theory, we define the *free type* corresponding to a (generic) class G as the 'most general instantiation' of class G, or, equivalently, as the least G-supertype, and define the *cofree type* corresponding to G, if it at all exists, as the 'least general instantiation' of G, or, equivalently, as the greatest G-subtype. (The motivation behind the definition of these new OOP notions is made clearer in §§3.2 and 4.2, where we discuss the Erasure Galois Connection and doubly F-bounded generics.)

Example A.30. In generic Java, the free type corresponding to a generic class `C` (informally, sort of "the free type that comes with" the generic class) is the wildcard type `C<?>`. As to cofree types, however, no OOP language that we know of directly supports them so far. (See §4.4 for further discussion of cofree types.)

[46] For more on correspondences between order theory, power set theory, and category theory, check [11].

[47] Thus, G-supertypes and G-subtypes in generic OOP precisely correspond to algebras and coalgebras in category theory, and to pre-fixed and post-fixed points in order/lattice theory, respectively.

[48] Here, it is assumed that G has an *unbounded* type parameter; or that this statement about type `Object` applies to *admittible* type arguments that may not necessarily be *valid* type arguments (to class G). Even though a detailed treatment of admittible versus valid type arguments is kept for future work, §6 includes a brief discussion.

[49] Beyond mathematics and computer science, (least and greatest) fixed points have applications in many scientific fields, *e.g.* in economics and finance [59]. Fixed points also have strong affinity with *eigenvectors* (and *eigenvalues*) of operators (*i.e.*, matrices) in linear algebra. Like fixed points, eigenvectors and eigenvalues also have many applications outside mathematics, *e.g.*, in machine learning algorithms, and in quantum physics and quantum computing.