Lightweight, Flexible Object-Oriented Generics

Yizhou Zhang* Matthew C. Loring* Guido Salvaneschi†
Barbara Liskov‡ Andrew C. Myers*

*Cornell University, USA †TU Darmstadt, Germany ‡MIT, USA
yizhou@cs.cornell.edu mcl83@cornell.edu salvaneschi@cs.tu-darmstadt.de
liskov@csail.mit.edu andru@cs.cornell.edu

Abstract

The support for generic programming in modern object-oriented programming languages is awkward and lacks desirable expressive power. We introduce an expressive genericity mechanism that adds expressive power and strengthens static checking, while remaining lightweight and simple in common use cases. Like type classes and concepts, the mechanism allows existing types to model type constraints retroactively. For expressive power, we expose *models* as named constructs that can be defined and selected explicitly to witness constraints; in common uses of genericity, however, types implicitly witness constraints without additional programmer effort. Models are integrated into the object-oriented style, with features like model generics, model-dependent types, model enrichment, model multimethods, constraint entailment, model inheritance, and existential quantification further extending expressive power in an object-oriented setting. We introduce the new genericity features and show that common generic programming idioms, including current generic libraries, can be expressed more precisely and concisely. The static semantics of the mechanism and a proof of a key decidability property can be found in an associated technical report.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism, Constraints; D.3.2 [*Language Classifications*]: Object-oriented languages; F.3.2 [*Semantics of Programming Languages*]

Keywords Genus; generic programming; constraints; models

1. Introduction

Generic programming provides the means to express algorithms and data structures in an abstract, adaptable, and interoperable form. Specifically, genericity mechanisms allow polymorphic code to apply to different types, improving modularity and reuse. Despite decades of work on genericity mechanisms, current OO languages still offer an unsatisfactory tradeoff between expressiveness and usability. These languages do not provide a design that coherently integrates desirable features—particularly, retroactive extension and dynamic dispatch. In practice, existing genericity mechanisms force developers to circumvent limitations in expressivity by using awkward, heavyweight design patterns and idioms.

The key question is how to expose the operations of type parameters in a type-safe, intuitive, and flexible manner within the OO paradigm. The following somewhat daunting Java signature for method `Collections::sort` illustrates the problem:

```
<T extends Comparable<? super T>> void sort(List<T> l)
```

The subtyping constraint constrains a type parameter `T` using the `Comparable` interface, ensuring that type `T` is comparable to itself or to one of its supertypes. However, `sort` can only be used on a type `T` if that type argument is explicitly declared to implement the `Comparable` interface. This restriction of nominal subtyping is alleviated by structural constraints as introduced by CLU [22, 23] and applied elsewhere (e.g., [10, 12]), but a more fundamental limitation remains: items of type `T` cannot be sorted unless `T` has a `compareTo` operation to define the sort order. That limitation is addressed by *type classes* in Haskell [41]. Inspired by Haskell, efforts have been made to incorporate type classes into OO languages with language-level support [33, 37, 39, 43] and the Concept design pattern [30]. However, as we argue, these designs do not fully exploit what type classes and OO languages have to offer when united.

This paper introduces a new genericity mechanism, embodied in a new extension of Java called Genus. The genericity mechanism enhances expressive power, code reuse, and static type safety, while remaining lightweight and intuitive for the programmer in common use cases. Genus supports *models* as named constructs that can be defined and selected explicitly to witness constraints, even for primitive type arguments; however, in common uses of genericity, types implicitly witness constraints without additional programmer effort. The key novelty of models in Genus is their deep integration into the OO style, with features like model generics, model-dependent types, model enrichment, model multimethods, constraint entailment, model inheritance, and existential quantification further extending expressive power in an OO setting.

The paper compares Genus to other language designs; describes its implementation; shows that Genus enables safer, more concise code through experiments that use it to reimplement existing generic libraries; and presents performance measurements that show that a naive translation from Genus to Java yields acceptable performance and that with simple optimizations, Genus can offer very good performance. A formal static semantics for a core version of Genus is available in the technical report [44], but omitted here due to lack of space; there we show that termination of default model resolution holds under reasonable syntactic restrictions.

2. The Need for Better Genericity

Prior work has explored various approaches to constrained genericity: subtyping constraints, structural matching, type classes, and design patterns. Each of these approaches has significant weaknesses.

Copyright © ACM, 2015. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *PLDI'15*, June 13–17, 2015, Portland, OR, USA. http://dx.doi.org/10.1145/2737924.2738008.

```
class AbstractVertex
  <EdgeType extends
      AbstractEdge<EdgeType, ActualVertexType>,
   ActualVertexType extends
      AbstractVertex<EdgeType, ActualVertexType>> {...}
class AbstractEdge
  <ActualEdgeType extends
      AbstractEdge<ActualEdgeType, VertexType>,
   VertexType extends
      AbstractVertex<ActualEdgeType, VertexType>> {...}
```

Figure 1: Parameter clutter in generic code.

```
class TreeSet<T> implements Set<T> {
  TreeSet(Comparator<? super T> comparator) {...} ...
}
interface Comparator<T> { int compare(T o1, T o2); }
```

Figure 2: Concept design pattern.

The trouble with subtyping. Subtyping constraints are used in Java [5], C# [14, 21], and other OO languages. In the presence of nominal subtyping, subtyping constraints are too inflexible: they can only be satisfied by classes explicitly declared to implement the constraint. Structural subtyping and matching mechanisms (e.g., [10, 12, 23, 26]) do not require an explicit declaration that a constraint is satisfied, but still require that the relevant operations exist, with conformant signatures. Instead, we want retroactive modeling, in which a *model* (such as a type class instance [41]) can define how an existing type satisfies a constraint that it was not planned to satisfy ahead of time.

Subtyping constraints, especially when F-bounded [7], also tend to lead to complex code when multiple type parameters are needed. For example, Figure 1 shows a simplification of the signatures of the classes `AbstractVertex` and `AbstractEdge` in the FindBugs project [15]. The vertex and the edge types of a graph have a mutual dependency that is reflected in the signatures in an unpleasantly complex way (See Figure 3 for our approach).

Concept design pattern. Presumably because of these limitations, the standard Java libraries mostly do *not* use constraints on the parameters of generic classes in the manner originally envisioned [5]. Instead, they use a version of the Concept design pattern [27] in which operations needed by parameter types are provided as arguments to constructors. For instance, a constructor of `TreeSet`, a class in the Java collections framework, accepts an object of the `Comparator` class (Figure 2). The `compare` operation is provided by this object rather than by `T` itself.

This design pattern provides missing flexibility, but adds new problems. First, a `comparator` object must be created even when the underlying type has a comparison operation. Second, because the model for `Comparator` is an ordinary (first-class) object, it is hard to specialize or optimize particular instantiations of generic code. Third, there is no static checking that two `TreeSet`s use the same ordering; if an algorithm relies on the element ordering in two `TreeSet`s being the same, the programmer may be in for a shock.

In another variant of the design pattern, used in the C++ STL [25], an extra parameter for the class of the comparator distinguishes instantiations that use different models. However, this approach is more awkward than the `Comparator` object approach. Even the common case, in which the parameter type has exactly the needed operations, is just as heavyweight as when an arbitrary, different operation is substituted.

Type classes and concepts. The limitations of subtyping constraints have led to recent research on adapting type classes to OO languages to achieve retroactive modeling [37]. However, type classes have limitations: first, constraint satisfaction must be uniquely witnessed, and second, their models define how to adapt a single type, whereas in a language with subtyping, each adapted type in general represents all of its subtypes.

No existing approach addresses the first limitation, but an attempt is made by JavaGI [43] to fit subtyping polymorphism and dynamic dispatch into constrained genericity. As we will argue (§5.1), JavaGI's limited dynamic dispatch makes certain constraints hard to express, and interactions between subtyping and constraint handling make type checking subject to nontermination.

Beyond dynamic dispatch, it is important for OO programming that extensibility applies to models as well. The essence of OO programming is that new behavior can be added later in a modular way; we consider this post-factum *enrichment* of models to be a requirement.

Goals. What is wanted is a genericity mechanism with multiple features: retroactive modeling, a lightweight implicit approach for the common case, multiparameter type constraints, non-unique constraint satisfaction with dynamic, extensible models, and model-dependent types. The mechanism should support modular compilation. It should be possible to implement the mechanism efficiently; in particular, an efficient implementation should limit the use of wrapper objects and should be able to specialize generic code to particular type arguments—especially, to primitive types. Genus meets all of these goals. We have tried not only to address the immediate problems with generics seen in current OO languages, but also to take further steps, adding features that support the style of programming that we expect will evolve when generics are easier to use than they are now.

3. Type Constraints in Genus

3.1 Type Constraints as Predicates

Instead of constraining types with subtyping, Genus uses explicit *type constraints* similar to type classes. For example, the constraint

```
constraint Eq[T] {
  boolean equals(T other);
}
```

requires that type `T` have an `equals` method.[1] Although this constraint looks like a Java interface, it is really a predicate on types, like a (multiparameter) type class in Haskell [32]. We do not call constraints "type classes" because there are differences and because the name "class" is already taken in the OO setting.

Generic code can require that actual type parameters satisfy constraints. For example, here is the `Set` interface in Genus (simplified):

```
interface Set[T where Eq[T]]  { ... }
```

The *where clause* "`where Eq[T]`" establishes the ability to test equality on type `T` within the scope of `Set`. Consequently, an instantiation of `Set` needs a witness that `Eq` is satisfied by the type argument. In Genus, such witnesses come in the form of *models*. Models are either implicitly chosen by the compiler or explicitly supplied by the programmer.

Multiparameter constraints. A constraint may be a predicate over multiple types. Figure 3 contains an example in which a constraint `GraphLike[V,E]` declares graph operations that should be satisfied by any pair of types `[V,E]` representing vertices and edges of a graph. In a multiparameter constraint, methods must explicitly declare receiver types (`V` or `E` in this case). Every operation in this constraint mentions both `V` and `E`; none of the operations really belongs to any single type. The ability to group related types and operations into a single constraint leads to code that is more modular and more readable than that in Figure 1.

[1] We denote Genus type parameters using square brackets, to distinguish Genus examples from those written in other languages (especially, Java).

```
// A multiparameter constraint
constraint GraphLike[V,E] {
  Iterable[E] V.outgoingEdges();
  Iterable[E] V.incomingEdges();
  V E.source();
  V E.sink();
}
```

```
constraint OrdRing[T]
  extends Comparable[T]
{
  // static methods
  static T T.zero();
  static T T.one();
  T T.plus(T that);
  T T.times(T that);
}
```

Figure 3: Constraints `GraphLike` and `OrdRing`.

```
Map[V,W] SSSP[V,E,W](V s)
    where GraphLike[V,E], Weighted[E,W],
          OrdRing[W], Hashable[V] {
  TreeMap[W,V] frontier = new TreeMap[W,V]();
  Map[V,W] distances = new HashMap[V,W]();
  distances.put(s, W.one()); frontier.put(W.one(), s);
  while (frontier.size() > 0) {
    V v = frontier.pollFirstEntry().getValue();
    for (E vu : v.outgoingEdges()) {
      V u = vu.sink();
      W weight = distances.get(v).times(vu.weight());
      if (!distances.containsKey(u) ||
          weight.compareTo(distance.get(u)) < 0) {
        frontier.put(weight, u);
        distances.put(u, weight);
      }}} return distances; }
```

Figure 4: A highly generic method for Dijkstra's single-source shortest-path algorithm. Definitions of `Weighted` and `Hashable` are omitted. Ordering and composition of distances are generalized to an ordered ring. (A more robust implementation might consider using a priority queue instead of `TreeMap`.)

Prerequisite constraints. A constraint can have other constraints as its *prerequisites*. For example, `Eq[T]` is a prerequisite constraint of `Comparable[T]`:

```
constraint Comparable[T] extends Eq[T] {
  int compareTo(T other);
}
```

To satisfy a constraint, its prerequisite constraints must also be satisfied. Therefore, the satisfaction of a constraint *entails* the satisfaction of its prerequisites. For example, the Genus version of the `TreeSet` class from Figure 2 looks as follows:

```
class TreeSet[T where Comparable[T]] implements Set[T] {
   TreeSet() { ... } ...
}
```

The type `Set[T]` in the definition of `TreeSet` is well-formed because its constraint `Eq[T]` is entailed by the constraint `Comparable[T]`.

Static constraint members. Constraints can require that a type provide static methods, indicated by using the keyword `static` in the method declaration. In Figure 3, constraint `OrdRing` specifies a static method (`zero`) that returns the identity of the operation `plus`.

All types `T` are also automatically equipped with a static method `T.default()` that produces the default value for type `T`. This method is called, for instance, to initialize the elements of an array of type `T[]`. The ability to create an array of type `T[]` is often missed in Java.

3.2 Prescribing Constraints Using Where Clauses

Where-clause constraints enable generic algorithms, such as the version of Dijkstra's shortest-path algorithm in Figure 4, generalized to ordered rings.[2] The where clause of `SSSP` requires only that the type arguments satisfy their respective constraints—no subtype relationship is needed.

[2] The usual behavior is achieved if `plus` is $\min$, `times` is $+$, and `one` is 0.

Where-clause constraints endow typing contexts with assumptions that the constraints are satisfied. So the code of `SSSP` can make method calls like `vu.sink()` and `W.one()`. Note that the where clause may be placed after the formal parameters as in CLU; this notation is just syntactic sugar for placing it between the brackets.

Unlike Java `extends` clauses, a where clause is not attached to a particular parameter. It can include multiple constraints, separated by commas. Each constraint requires a corresponding model to be provided when the generic is instantiated. To allow models to be identified unambiguously in generic code, each such constraint in the where clause may be explicitly named as a *model variable*.

Another difference from Java `extends` clauses is that a where clause may be used without introducing a type parameter. For example, consider the `remove` method of `List`. Expressive power is gained if its caller can specify the notion of equality to be used, rather than requiring `List` itself to have an intrinsic notion of equality. Genus supports this genericity by allowing a constraint `Eq[E]` to be attached to `remove`:

```
interface List[E] {
  boolean remove(E e) where Eq[E]; ...
}
```

We call this feature *model genericity*.

3.3 Witnessing Constraints Using Models

As mentioned, generic instantiations require witnesses that their constraints are satisfied. In Genus, witnesses are provided by models. Models can be inferred—a process we call *default model resolution*—or specified explicitly, offering both convenience in common cases and expressivity when needed. We start with the *use* of models and leave the *definition* of models until §4.

Using default models. It is often clear from the context which models should be used to instantiate a generic. For instance, the `Set[T]` interface in the `TreeSet` example (§3.1) requires no further annotation to specify a model for `Eq[T]`, because the model can be uniquely resolved to the one promised by `Comparable[T]`.

Another common case is that the underlying type already has the required operations. This case is especially likely when classes are designed to support popular operations; having to supply models explicitly in this case would be a nuisance. Therefore, Genus allows types to *structurally conform* to constraints. When the methods of a type have the same names as the operations required by a constraint, and also have conformant signatures, the type automatically generates a *natural model* that witnesses the constraint. For example,[3] the type `Set[String]` means a `Set` that distinguishes strings using `String`'s built-in `equals` method. Thus, the common case in which types provide exactly the operations required by constraints is simple and intuitive. In turn, programmers have an incentive to standardize the names and signatures of popular operations.

Genus supports using primitive types as type arguments, and provides natural models for them that contain common methods. For example, a natural model for `Comparable[int]` exists, so types like `TreeSet[int]` that need that model can be used directly.

Default models can be used to instantiate any generic—not just generic classes. For example, consider this sort method:

```
void sort[T](List[T] l) where Comparable[T] { ... }
```

The call `sort(x)`, where `x` is a `List[int]`, infers `int` both as the type argument and as the default model. Default model resolution, and more generally, type and model inference, are discussed further in §4.4 and §4.7.

[3] We assume throughout that the type `String` has methods "`boolean equals(String)`" and "`int compareTo(String)`."

Using named models. It is also possible to explicitly supply models to witness constraints. To do so, programmers use the `with` keyword followed by models for each of the where-clause constraints in the generic. These models can come from programmer-defined models (§4) or from model variables declared in where clauses (§3.2). For example, suppose model `CIEq` tests `String` equality in a case-insensitive manner. The type `Set[String with CIEq]` then describes a `Set` in which all strings are distinct without case-sensitivity. In fact, the type `Set[String]` is syntactic sugar for `Set[String with String]`, in which the `with` clause is used to explicitly specify the natural model that `String` automatically generates for `Eq[String]`.

A differentiating feature of our mechanism is that different models for `Eq[String]` can coexist in the same scope, allowing a generic class like `Set`, or a generic method, to be instantiated in more than one way in a scope:

```
Set[String] s0 = ...;
Set[String with CIEq] s1 = ...;
s1 = s0; // illegal assignment: different types.
```

The ordering that an instantiation of `Set` uses for its elements is *part of the type*, rather than a purely dynamic argument passed to a constructor as in the Concept pattern. Therefore, the final assignment statement is a static type error. The type checker catches the error because the different models used in the two `Set` instantiations allow `Set`s using different notions of equality to be distinguished. The use of models in types is discussed further in §4.5.

It is also possible to express types using *wildcard models*; the type `Set[String with ?]` is a supertype of both `Set[String]` and `Set[String with CIEq]`. Wildcard models are actually syntactic sugar for existential quantification (§6).

4. Models

Models can be defined explicitly to allow a type to satisfy a constraint when the natural model is nonexistent or undesirable. For example, the case-insensitive string equality model `CIEq` can be defined concisely:

```
model CIEq for Eq[String] {
  bool equals(String str) {
    return equalsIgnoreCase(str);
  }
}
```

Furthermore, a model for case-insensitive `String` ordering might be defined by reusing `CIEq` via *model inheritance*, to witness the prerequisite constraint `Eq[String]`:

```
model CICmp for Comparable[String] extends CIEq {
  int compareTo(String str) {
    return compareToIgnoreCase(str);
  }
}
```

It is also possible for `CICmp` to satisfy `Eq` by defining its own `equals` method. Model inheritance is revisited in §5.3.

Models are immutable: they provide method implementations but do not have any instance variables. Models need not have global scope; modularity is achieved through the Java namespace mechanism. Similarly, models can be nested inside classes and are subject to the usual visibility rules.

4.1 Models as Expanders

Operations provided by models can be invoked directly, providing the functionality of *expanders* [42]. For example, the call `"x".(CIEq.equals)("X")` uses `CIEq` as the expander to test equality of two strings while ignoring case. Natural models can similarly be selected explicitly using the type name: `"x".(String.equals)("X")`.

```
constraint Cloneable[T] { T clone(); }
model ArrayListDeepCopy[E] for Cloneable[ArrayList[E]]
      where Cloneable[E] {
  ArrayList[E] clone() {
    ArrayList[E] l = new ArrayList[E]();
    for (E e : this) { l.add(e.clone()); }
    return l;
}}
```

Figure 5: A parameterized model.

```
model DualGraph[V,E] for GraphLike[V,E]
      where GraphLike[V,E] g {
  V E.source() { return this.(g.sink)(); }
  V E.sink() { return this.(g.source)(); }
  Iterable[E] V.incomingEdges() {
    return this.(g.outgoingEdges)(); }
  Iterable[E] V.outgoingEdges() {
    return this.(g.incomingEdges)(); }
}
void SCC[V,E](V[] vs) where GraphLike[V,E] g { ...
  new DFIterator[V,E with g]() ...
  new DFIterator[V,E with DualGraph[V,E with g]]() ...
}
class DFIterator[V,E] where GraphLike[V,E] {...}
```

Figure 6: Kosaraju's algorithm. Highlighted code is inferred if omitted.

Using models as expanders is an integral part of our genericity mechanism: the operations promised by where-clause constraints are invoked using expanders. In Figure 4, if we named the where-clause constraint `GraphLike[V,E]` with model variable `g`, the call `vu.sink()` would be sugar for `vu.(g.sink)()` with `g` being the expander. In this case, the expander can be elided because it can be inferred via default model resolution (§4.4).

4.2 Parameterized Models

Model definitions can be generic: they can be parameterized with type parameters and where-clause constraints. For example, model `ArrayListDeepCopy` (Figure 5) gives a naive implementation of deep-copying `ArrayList`s. It is generic with respect to the element type `E`, but requires `E` to be cloneable.

As another example, we can exploit model parameterization to implement the transpose of any graph. In Figure 6, the `DualGraph` model is itself a model for `GraphLike[V,E]`, and is parameterized by another model for `GraphLike[V,E]` (named `g`). It represents the transpose of graph `g` by reversing its edge orientations.

4.3 Non-Uniquely Witnessing Constraints

Previous languages with flexible type constraints, such as Haskell, JavaGI, and $\mathcal{G}$, require that witnesses be unique at generic instantiations, whether witnesses are scoped globally or lexically. By contrast, Genus allows multiple models witnessing a given constraint instantiation to coexist in the same context. This flexibility increases expressive power.

For example, consider Kosaraju's algorithm for finding strongly connected components in a directed graph [2]. It performs two depth-first searches, one following edges forward, and the other on the transposed graph, following edges backward. We would like to reuse the same generic depth-first-search algorithm on the same graph data structure for both traversals.

In Figure 6, the where clause of `SCC` introduces into the context a model for `GraphLike[V,E]`, denoted by model variable `g`. Using the `DualGraph` model, the algorithm code can then perform both forward and backward traversals. It instantiates `DFIterator`, an iterator class for depth-first traversal, twice, with the original graph model `g` and with the transposed one. Being able to use two different models to witness the same constraint instantiation in `SCC` enables more code

reuse. The highlighted `with` clauses can be safely elided, which brings us to default model resolution.

4.4 Resolving Default Models

In Genus, the omission of a `with` clause triggers default model resolution. Default model resolution is based on the following four ways in which models are *enabled* as potential default choices. First, types automatically generate natural models when they structurally conform to constraints. Natural models, when they exist, are always enabled as default candidates. Second, a where-clause constraint enables a model within the scope of the generic to which the where clause is attached. For example, in method `SCC` in Figure 6 the where clause enables a model as a default candidate for `GraphLike[V,E]` within `SCC`. Third, a `use` declaration, e.g.,

```
use ArrayListDeepCopy;
```

enables the specified model as a potential default way to clone `ArrayLists` in the compilation unit in which the declaration resides. Fourth, a model itself is enabled as a potential default model within its definition.

Default model resolution works as follows:

1. If just one model for the constraint is enabled, it becomes the default model.
2. If more than one model is enabled, programmer intent is ambiguous. In this case, Genus requires that programmers make their intent explicit using a `with` clause. Omitting the `with` clause is a static error in this case.
3. If *no* model is explicitly enabled, but there is in scope a single model for the constraint, that model becomes the default model for the constraint.

Resolution for an elided expander in a method call works similarly. The only difference is that instead of searching for a model that witnesses a constraint, the compiler searches for a model that contains a method applicable to the given call. In typical use, this would be the natural model.

These rules for default models make generics and expanders easy to use in the common cases; in the less common cases where there is some ambiguity about which model to use, they force the programmer to be explicit and thereby help prevent hard-to-debug selection of the wrong model.

Letting each compilation unit choose its own default models is more flexible and concise than using Scala implicits, where a type-class instance can only be designated as implicit at the place where it is defined, and implicit definitions are then imported into the scope, with a complex process used to find the most specific implicit among those imported [29]. We aim for simpler rules.

Genus also achieves the conciseness of Haskell type classes because uniquely satisfying models are allowed to witness constraints without being enabled, just as a unique type class instance in Haskell satisfies its type class without further declarations. But natural models make the mechanism lighter-weight than in Haskell, and the ability to have multiple models adds expressive power (as in the `SCC` example in Figure 6).

4.5 Models in Types

Section 3.3 introduced the ability to instantiate generic types with models, which become part of the type (i.e., model-dependent types). Type safety benefits from being able to distinguish instantiations that use different models.

The `addFromSorted` method in `TreeSet` (Figure 7) adds all elements in the source `TreeSet` to `this` one. Its signature requires that the source `TreeSet` and `this` one use the same ordering. So a `TreeSet` with a different ordering cannot be accidentally passed to this method, avoiding a run-time exception.

```
class TreeSet[T] implements Set[T with c]
    where Comparable[T] c {
  TreeSet() {...}
  void addAll(Collection[? extends T] src) {
    if (src instanceof TreeSet[? extends T with c]) {
      addFromSorted((TreeSet[? extends T with c])src);
    } else {...}
  }
  void addFromSorted(TreeSet[? extends T with c] src) {
    ... // specialized code in virtue of the same
        // ordering in src and this
  } ... }
```

Figure 7: `TreeSet` in Genus. Highlighted code is inferred if omitted.

Including the choice of model as part of the type is unusual, perhaps because it could increase annotation burden. Models are not part of types in the Concept design pattern (e.g., as realized in Scala [30]), because type class instances are not part of instantiated types. $\mathcal{G}$ [37] allows multiple models for the same constraint to be defined in one program (albeit only one in any lexical scope), yet neither at compile time nor at run time does it distinguish generic instantiations with distinct models. This raises potential safety issues when different modules interoperate.

In Genus, the concern about annotation burden is addressed by default models. For example, the type `TreeSet[? extends T]` in Figure 7 is implicitly instantiated with the model introduced by the where clause (via constraint entailment, §5.2). By contrast, Scala implicits work for method parameters, but *not* for type parameters of generic classes.

4.6 Models at Run Time

Unlike Java, whose type system is designed to support implementing generics via erasure, Genus makes models and type arguments available at run time. Genus allows testing the type of an object from a parameterized class at run time, like the `instanceof` test and the type cast in Figure 7.

Reifiability creates opportunities for optimization. For example, consider `TreeSet`'s implementation of the `addAll` method required by the `Collection` interface. In general, an implementation cannot rely on seeing the elements in the order expected by the destination collection, so for each element in the source collection, it must traverse the destination `TreeSet` to find the correct position. However, if both collections use the same ordering, the merge can be done in a more asymptotically efficient way by calling the specialized method `addFromSorted`.

4.7 Default Model Resolution: Algorithmic Issues

Recursive resolution of default models. Default model resolution is especially powerful because it supports recursive reasoning. For example, the `use` declaration in §4.4 is syntactic sugar for the following parameterized declaration:

```
use [E where Cloneable[E] c] ArrayListDeepCopy[E with c]
  for Cloneable[ArrayList[E]];
```

The default model candidacy of `ArrayListDeepCopy` is valid for cloning objects of any instantiated `ArrayList` type, provided that the element type satisfies `Cloneable` too. Indeed, when the compiler investigates the use of `ArrayListDeepCopy` to clone `ArrayList[Foo]`, it creates a subgoal to resolve the default model for `Cloneable[Foo]`. If this subgoal fails to be resolved, `ArrayListDeepCopy` is not considered as a candidate.

Recursive resolution may not terminate without additional restrictions. As an example, the declaration "`use DualGraph;`" is illegal because its recursive quest for a model of the same constraint causes resolution to cycle. The issue is addressed in §9 and the technical report [44] by imposing syntactic restrictions.

When a `use` declaration is rejected by the compiler for violating the restrictions, the programmer always has the workaround of explicitly selecting the model. By contrast, the inability to do so in Haskell or JavaGI makes it impossible to have a model like `DualGraph` in these languages.

Unification vs. default model resolution. Since Genus uses models in types, it is possible for models to be inferred via unification when they are elided. This inference potentially raises confusion with default model resolution.

Genus distinguishes between two kinds of where-clause constraints. Constraints for which the model is required by a parameterized type, such as `Eq[T]` in the declaration `void f[T where Eq[T]](Set[T] x)`, are called *intrinsic constraints*, because the `Set` must itself hold the corresponding model. By contrast, a constraint like `Printable[T]` in the declaration `void g[T where Printable[T]](List[T] x)` is *extrinsic* because `List[T]` has no such constraint on `T`.

Inference in Genus works by first solving for type parameters and intrinsic constraints via unification, and only then resolving default models for extrinsic constraints. To keep the semantics simple, Genus does not use default model availability to guide unification, and it requires extrinsic where-clause constraints to be written to the right of intrinsic ones. Nevertheless, it is always possible for programmers to explicitly specify intent.

4.8 Constraints/Models vs. Interfaces/Objects

The relationship between models and constraints is similar to that between objects and interfaces. Indeed, the Concept pattern can be viewed as using objects to implement models, and JavaGI extends interfaces to encode constraints. In contrast, Genus draws a distinction between the two, treating models as second-class values that cannot be stored in ordinary variables. This design choice has the following basis:

- Constraints are used in practice very differently from "ordinary" types, as evidenced by the nearly complete separation between *shapes* and *materials* seen in an analysis of a very large software base [18]. In their parlance, interfaces or classes that encode multiparameter constraints (e.g., `GraphLike`) or constraints requiring binary operations (e.g., `Comparable`) are shapes, while ordinary types (e.g., `Set`) are materials. Muddling the two may give rise to nontermination (§9).
- Because models are not full-fledged objects, generic code can easily be specialized to particular using contexts.
- Because model expressions can be used in types, Genus has dependent types; however, making models second-class and immutable simplifies the type system and avoids undecidability.

5. Making Models Object-Oriented

5.1 Dynamic Dispatching and Enrichment

In OO programs, subclasses are introduced to specialize the behavior offered by their superclasses. In Genus, models define part of the behavior of objects, so models too should support specialization. Therefore, a model in Genus may include not only method definitions for the base type, but also methods defining more specific behavior for subtypes. These methods can be dispatched *dynamically* by code both inside and outside model declarations. Dynamic dispatch takes place not only on the receiver, but also on method arguments of the manipulated types. The expressive power of dynamic dispatch is key to OO programming [3], and multiple dispatch is particularly important for binary operations, which are typically encoded as constraints. Our approach differs in this way from $\mathcal{G}$ and Scala, which do not support dynamic dispatch on model operations.

For example, model `ShapeIntersect` in Figure 8 gives multiple definitions of `intersect`, varying in their expected argument types.

```
constraint Intersectable[T] { T T.intersect(T that); }
model ShapeIntersect for Intersectable[Shape] {
  Shape Shape.intersect(Shape s) {...}
  // Rectangle and Circle are subclasses of Shape:
  Rectangle Rectangle.intersect(Rectangle r) {...}
  Shape Circle.intersect(Rectangle r) {...} ...
}
enrich ShapeIntersect {
  Shape Triangle.intersect(Circle c) {...} ...
}
```

Figure 8: An extensible model with multiple dispatch.

In a context where the model is selected, a call to `intersect` on two objects statically typed as `Shape` will resolve at run time to the most specific method definition in the model. In JavaGI, multiple dispatch on `intersect` is impossible, because its dispatch is based on "self" types [6], while the argument types (including receiver) as well as the return type of an `intersect` implementation do not necessarily have to be the same.

Existing OO type hierarchies are often extended with new subclasses in ways not predicted by their designers. Genus provides *model enrichment* to allow models to be extended in a modular way, in sync with how class hierarchies are extended; here we apply the idea of open classes [11] to models. For example, if `Triangle` is later introduced to the `Shape` hierarchy, the model can then be separately enriched, as shown in the `enrich` declaration in Figure 8.

Model multimethods and model enrichment create the same challenge for modular type checking that is seen with other extensible OO mechanisms. For instance, if two modules separately enrich `ShapeIntersect`, these enrichments may conflict. Like Relaxed MultiJava [24], Genus can prevent such errors with a load-time check that there is a unique best method definition for every method invocation, obtaining mostly modular type checking and fully modular compilation. The check can be performed soundly, assuming load-time access to the entire program. If a program loads new code dynamically, the check must be performed at the time of dynamic loading.

5.2 Constraint Entailment

As seen earlier (§3.1), a constraint entails its prerequisite constraints. In general, a model may be used as a witness not just for the constraint it is declared for, but also for any constraints entailed by the declared constraint. For example, a model for `Comparable[Shape]` can be used to witness `Eq[Shape]`.

A second way that one constraint can entail another is through *variance* on constraint parameters. For example, since in constraint `Eq` the type parameter only occurs in contravariant positions, a model for `Eq[Shape]` may also be soundly used as a model for `Eq[Circle]`. It is also possible, though less common, to use a model to witness constraints for supertypes, via covariance. Variance is inferred automatically by the compiler, with bivariance downgraded to contravariance.

A model enabled for some constraint in one of the four ways discussed in §4.4 is also enabled for its prerequisite constraints and constraints that can be entailed via contravariance. Accommodating subtyping extends the expressivity of default model resolution, but poses new challenges for termination. In the technical report [44], we show that encoding "shape" types (in the sense of Greenman et al. [18]) as constraints helps ensure termination.

5.3 Model Inheritance

Code reuse among models can be achieved through model inheritance, signified by an `extends` clause (e.g., model `CICmp` in §4). Unlike an `extends` clause in a class or constraint definition, which creates an *is-a* relationship between a subclass and its superclass or a constraint and its prerequisite constraint, an `extends` clause in a

model definition is *merely* for code reuse. The inheriting model inherits all method definitions *with compatible signatures* available in the inherited model. The inheriting model can also override these inherited definitions.

Model inheritance provides a means to derive models that are otherwise rejected by constraint entailment. For example, the model `ShapeIntersect` (Figure 8) soundly witnesses the same constraint for `Rectangle`, because the selected method definitions have compatible signatures, even though `Intersectable` is invariant with respect to its type parameter. The specialization to `Rectangle` can be performed succinctly using model inheritance, with the benefit of a more precise result type when two rectangles are intersected:

```
model RectangleIntersect for Intersectable[Rectangle]
    extends ShapeIntersect { }
```

6. Use-Site Genericity

Java's wildcard mechanism [40] is in essence a limited form of existential quantification. Existentials enable genericity at *use sites*. For example, a Java method with return type `List<? extends Printable>` can be used by generic calling code that is able to print list elements even when the type of the elements is unknown to the calling code. The use-site genericity mechanism of Genus generalizes this idea while escaping some limitations of Java wildcards. Below we sketch the mechanism.

6.1 Existential Types

Using subtype-bounded existential quantification, the Java type `List<? extends Printable>` might be written more type-theoretically as ∃U≤`Printable.List[U]`. Genus extends this idea to constraints. An existential type in Genus is signified by prefixing a quantified type with type parameters and/or where-clause constraints. For example, if `Printable` is a constraint, the Genus type corresponding to the Java type above is `[some U where Printable[U]]List[U]`. The initial brackets introduce a use-site type parameter `U` and a model for the given constraint, which are in scope in the quantified type; the syntax emphasizes the connection between existential and universal quantification.

The presence of prefixed parameters in existential types gives the programmer control over the existential binding point, in contrast to Java wildcard types where binding is always at the generic type in which the wildcard is used as a type argument. For example, no Java type can express ∃`U.List[List[U]]`, meaning a *homogeneous* collection of lists in which each list is parameterized by the *same* unknown type. This type is easily expressed in Genus as `[some U]List[List[U]]`.

Genus also offers convenient syntactic sugar for common uses of existential types. A single-parameter constraint can be used as sugar for an existential type: e.g., `Printable`, used as a type, is sugar for `[some U where Printable[U]]U`, allowing a value of any printable type. The wildcard syntax `List[?]` represents an existential type, with the binding point the same as in the Java equivalent. The type with a wildcard model `Set[String with ?]` is sugar for `[some Eq[String] m]Set[String with m]`.

Subtyping and coercion. Genus draws a distinction between *subtyping* and *coercion* involving existential types. Coercion may induce extra computation (i.e., existential packing) and can be context-dependent (i.e., default model resolution), while subtyping cannot. For example, the `return` expression in Figure 9 typechecks not because `ArrayList[String]` is a subtype of the existential return type, but because of coercion, which works by packing together a value of type `ArrayList[String]` with a model for `Comparable[String]` (in this case, the natural model) into a single value. The semantics of subtyping involving where-clause-quantified existential types is designed in a way that makes it easy for programmers to reason about subtyping and joining types.

```
[some T where Comparable[T]]List[T] f () {
  return new ArrayList[String]();
}
1  sort(f());
2  [U] (List[U] l) where Comparable[U] = f(); // bind U
3  l.first().compareTo(l.last());    // U is comparable
4  U[] a = new U[64];      // use run-time info about U
5  l = new List[U]();               // new list, same U
```

Figure 9: Working with existential quantification.

Capture conversion. In Java, wildcards in the type of an expression are instantiated as fresh identifiers when the expression is type-checked, a process called *capture conversion* [17]. Genus extends this idea to constraints: in addition to fresh type variables, capture conversion generates fresh models for where-clause constraints, and enables them in the current scope.

For example, at line 1 in Figure 9, when the call to `sort` (defined in §3.3) is type-checked, the type of the call `f()` is capture-converted to `List[#T]`, where `#T` is the fresh type variable that capture conversion generates for `T`, and a model for `Comparable[#T]` becomes enabled in the current context. Subsequently, the type argument to `sort` is inferred as `#T`, and the default model for `Comparable[#T]` resolves to the freshly generated model.

6.2 Explicit Local Binding

Capture conversion is convenient but not expressive enough. Consider a Java object typed as `List<? extends Comparable>`. The programmer might intend the elements of this homogeneous list to be comparable to one another, but comparisons to anything other than `null` do not type-check.

The awkwardness is addressed in Genus by *explicit local binding* of existentially quantified type variables and where-clause constraints, giving them names that can be used directly in the local context. An example of this mechanism is found at line 2 in Figure 9. The type variable `U` can be used as a full-fledged type in the remainder of the scope.

As its syntax suggests, explicit local binding can be viewed as introducing an inlined generic method encompassing subsequent code. Indeed, it operates under the same rules as universally quantified code. For example, the where clause at line 2 enables a new model so that values of type `U` can be compared at line 3. Also, locally bound type variables are likewise reifiable (line 4). Moreover, the binding at line 2 is type-checked using the usual inference algorithm to solve for `U` and for the model for `Comparable[U]`: per §4.7, the former is inferred via unification and the latter via default model resolution—it is an extrinsic constraint. Soundness is maintained by ensuring that `l` is initialized upon declaration and that assignments to the variable preserve the meaning of `U`.

7. Implementation

We have built a partial implementation of the Genus language in Java. The implementation consists of about 23,000 lines of code, extending version 2.6.1 of the Polyglot compiler framework [28]. We have an essentially complete implementation of all type checking and inference features. Code generation works by translating to Java 5 code, relying on a Java compiler as a back end. Code generation is less complete than type checking but also less interesting; however, the compiler can compile the benchmarks of §8, which use generics in nontrivial ways. The current compiler implementation type-checks but does not generate code for multimethod dispatch or for existentials; it does not yet specialize instantiations to particular type arguments.

```
/* Source code in Genus */
class ArrayList[T] implements List[T] {
  T[] arr;
  ArrayList() { arr = new T[INITIAL_SIZE]; } ... }

/* Target code in Java */
class ArrayList<T,A$T> implements List<T,A$T> {
  A$T arr;
  ObjectModel<T,A$T> T$model; // run-time info about T
  ArrayList(ObjectModel<T,A$T> T$model) {
    this.T$model = T$model;
    arr = T$model.newArray(INITIAL_SIZE);
  } ... }
```

Figure 10: Translating the Genus class `ArrayList` into Java.

7.1 Implementing Constraints and Models

Constraints and models in Genus code are translated to parameterized interfaces and classes in Java. For example, the constraint `Comparable[T]` is translated to a parameterized Java interface `Comparable<T,A$T>` providing a method `compareTo` with the appropriate signature: `int compareTo(T,T)`. The extra type parameter `A$T` is explained in §7.3. Models are translated to Java classes that implement these constraint interfaces.

7.2 Implementing Generics

Parameterized Genus classes are translated to correspondingly parameterized Java classes. However, type arguments and models must be represented at run time. So extra arguments carrying this information are required by class constructors, and constructor bodies are extended to store these arguments as fields. For example, class `ArrayList` has a translated constructor with the signature shown in Figure 10. Parameterized methods and models are translated in a similar way by adding extra arguments representing type and model information.

7.3 Supporting Primitive Type Arguments

A challenge for efficient generics, especially with a JVM-based implementation, is how to avoid uniformly wrapping all primitives inside objects when primitive types are used as type arguments. Some wrapping is unavoidable, but from the standpoint of efficiency, the key is that when code parameterized on a type `T` is instantiated on a primitive type (e.g., `int`), the array type `T[]` should be represented exactly as an array of the primitive type (e.g., `int[]`), rather than a type like `Integer[]` in which every array element incurs the overhead of individualized memory management.

Our current implementation uses a *homogeneous* translation to support this efficiency; the model object (e.g., `T$model` in Figure 10) for a type parameter `T` provides all operations about `T[]`. The model object has the interface type `ObjectModel<T,A$T>`, which specifies, via `A$T`, the operations for creating and accessing arrays of (unboxed) `T`. For example, the type of the model object used to create an `ArrayList[double]` implements `ObjectModel<Double,double[]>`. All interfaces representing single-parameter constraints implicitly extend `ObjectModel<T,A$T>`, so an `ObjectModel` argument is usually needed only on generics that do not otherwise constrain their type parameters.

A more efficient approach to supporting primitive type arguments is to generate specialized code for primitive instantiations, as is done in C#. The design of Genus makes it straightforward to implement particular instantiations with specialized code.

8. Evaluation

8.1 Porting Java Collections Framework to Genus

To evaluate how well the language design works in practice, we ported all 10 general-purpose implementations in the Java collections framework (JCF) as well as relevant interfaces and abstract implementations, to Genus. The result is a safer, more precise encoding and more code reuse with little extra programmer effort.

The single most interesting constrained generic in JCF is probably `TreeSet` (and `TreeMap`, which backs it). In its Java implementation, elements are ordered using either the element type's implementation of `Comparable` or a comparator object passed as a constructor argument, depending on which constructor is used to create the set. This ad hoc choice results in error-prone client code. In Genus, by contrast, the ordering is part of the `TreeSet` type, eliminating 35 occurrences of `ClassCastException` in `TreeSet`'s and `TreeMap`'s specs.

Genus collection classes are also more faithful to the semantics of the abstractions. Unlike a `Set[E]`, a `List[E]` should not necessarily be able to test the equality of its elements. In Genus, collection methods like `contains` and `remove` are instead parameterized by the definition of equality (§3.2). These methods cannot be called unless a model for `Eq[E]` is provided.

More powerful genericity also enables increased code reuse. For example, the `NavigableMap` interface allows extracting a descending view of the original map. In JCF, `TreeMap` implements this view by defining separate classes for each of the ascending and descending views. In contrast, Genus expresses both views concisely in a single class parameterized by a model that defines how to navigate the tree, eliminating 160 lines of code. This change is made possible by retroactive, non-unique modeling of `compareTo()`.

Thanks to default models—in particular, implicit natural models, for popular operations including `toString`, `equals`, `hashCode` and `compareTo`—client and library code ordinarily type-check without using `with` clauses. When `with` clauses are used, extra expressive power is obtained. In fact, the descending views are the *only* place where `with` clauses are needed in the Genus collection classes.

8.2 Porting the Findbugs Graph Library to Genus

We ported to Genus the highly generic Findbugs [15] graph library (∼1000 non-comment LoC), which provides graph algorithms used for the intermediate representation of static analyses. In Findbugs, the entities associated with the graph (e.g., `Graph`, `Vertex`, `Edge`) are represented as Java interfaces; F-bounded polymorphism is used to constrain parameters. As we saw earlier (§2), the resulting code is typically more cumbersome than the Genus version.

We quantified this effect by counting the number of parameter types, concrete types and keywords (`extends`, `where`) in each type declaration, ignoring modifiers and the name of the type. Across the library, Genus reduces annotation burden by 32% yet increases expressive power. The key is that constraints can be expressed directly without encoding them into subtyping and parametric polymorphism; further, prerequisite constraints avoid redundancy.

8.3 Performance

The current Genus implementation targets Java 5. To explore the overhead of this translation compared to similar Java code, we implemented a small Genus benchmark whose performance depends heavily on the efficiency of the underlying genericity mechanism, and hence probably exaggerates the performance impact of generics. The benchmark performs insertion sort over a large array or other ordered collection; the actual algorithm is the same in all cases, but different versions have different degrees of genericity with respect to the element type and even to the collection being sorted. Element type `T` is required to satisfy a constraint `Comparable[T]` and type `A` is required to satisfy a constraint `ArrayLike[A,T]`, which requires `A` to act like an array of `T`'s. Both primitive values (`double`) and ordinary object types (`Double`) are sorted.

The results from sorting collections of 100k elements are summarized in Table 1. Results were collected using Java 7 on a MacBook Pro with a 2.6GHz Intel Core i7 processor. All measurements

Table 1: Comparing performance of Java and Genus

	data structure	Java (s)	Genus (s) [spec.]
Non-generic sort	`double[]`	1.3	
	`Double[]`	3.8	
	`ArrayList[double]`	—	5.4 [4.0]
	`ArrayList[Double]`	9.6	14.5 [8.3]
Generic sort: `Comparable[T]`	`double[]`	—	19.3 [1.3]
	`Double[]`	7.7	10.0 [3.8]
	`ArrayList[double]`	—	6.7 [4.0]
	`ArrayList[Double]`	9.8	17.9 [8.3]
Generic sort: `ArrayLike[A,T]`, `Comparable[T]`	`double[]`	—	17.0 [1.3]
	`Double[]`	12.8	12.4 [3.8]
	`ArrayList[double]`	—	24.6 [4.0]
	`ArrayList[Double]`	12.8	24.8 [8.3]

are the average of 10 runs, with an estimated relative error always within 2%. For comparison, the same (non-generic) algorithm takes 1.1s in C (with `gcc -O3`). The Java column leaves some entries blank because Java does not allow primitive type arguments.

To understand the performance improvement that is possible by specializing individual instantiations of generic code, we used hand translation; as mentioned above, the design of Genus makes such specialization easy to do. The expected performance improvement is shown in the bracketed table entries. Specialization to primitive types is particularly useful for avoiding the high cost of boxing and unboxing primitive values, but the measurements suggest use of primitive type arguments can improve performance even without specialization (e.g., Genus `ArrayList[double]` is usually faster than Java `ArrayList<Double>`).

9. Formalization and Decidability

We have formalized the key aspects of the Genus type system, in the style of Featherweight Java [20]. Importantly, inference rules for subtyping, constraint entailment, and well-formedness (including model–constraint conformance in the presence of multimethods) are given. For lack of space, the formalization is provided in the technical report [44]. We are not aware of any unsoundness in the type system, but leave proving soundness to future work.

Default model resolution is an integral part of the formalization, matching the description in §4.4, §4.7 and §5.2. It is formalized as a translation from one calculus into another—the source calculus allows default models while the target is default-model-free.

Syntactic restrictions for decidable resolution of type class instances [38] and decidable subtyping with variance [18] have been separately proposed. We formulate our termination condition for default model resolution by synthesizing these restrictions, and to the best of our knowledge, give the first termination proof for such resolution when coupled with variance.

10. Related Work

Much prior work on parametric genericity mechanisms (e.g., [1, 5, 8, 10, 21, 23, 26, 34]) relies on constraint mechanisms that do not support retroactive modeling. We focus here on more recent work that follows Haskell's type classes in supporting retroactive modeling, complementing the discussion in previous sections.

The C++ community developed the Concept design pattern, based on templates, as a way to achieve retroactive modeling [4]. This pattern is used extensively in the STL and Boost libraries. Templates are not checked until instantiation, so developers see confusing error messages, and the lack of separate compilation makes compilation time depend on the amount of generic library code. The OO language $\mathcal{G}$ [37], based on System F^G [36], supports separate compilation but limits the power of concept-based overloading. By contrast, C++ Concepts [19] abandon separate compilation to fully support concept-based overloading. It was not adopted by the C++11 standard [35], however. Concept-based overloading is orthogonal to the other Genus features; it is not currently implemented but could be fully supported by Genus along with separate compilation, because models are chosen modularly at compile time.

In Scala, genericity is achieved with the Concept design pattern and *implicits* [30]. This approach is expressive enough to encode advanced features including associated types [9] and generalized constraints [14]. Implicits make using generics less heavyweight, but add complexity. Importantly, Scala does not address the problems with the Concept pattern (§2). In particular, it lacks model-dependent types and also precludes the dynamic dispatch that contributes significantly to the success of object-oriented programming [3].

JavaGI [43] generalizes Java interfaces by reusing them as type classes. Like a type class instance, a JavaGI `implementation` is globally scoped, must uniquely witness its interface, and may only contain methods for the type(s) it is declared with. Unlike in Haskell, a call to an interface method is dynamically dispatched across all `implementation`s. Although dispatch is not based entirely on the receiver type, within an `implementation` all occurrences of an implementing type for `T` must coincide, preventing multiply dispatching `intersect` across the `Shape` class hierarchy (cf. §5.1).

Approaches to generic programming in recent languages including Rust [33] and Swift [39] are also influenced by Haskell type classes, but do not escape their limitations.

Type classes call for a mechanism for implicitly and recursively resolving evidence of constraint satisfaction. The implicit calculus [31] formalizes this idea and extends it to work for all types. However, the calculus does not have subtyping. Factoring subtyping into resolution is not trivial, as evidenced by the reported stack overflow of the JavaGI compiler [18].

No prior work brings type constraints to use sites. The use of type constraints as types [39, 43] is realized as existentials in Genus. "Material–Shape Separation" [18] prohibits types such as `List<Comparable>`, which do find some usage in practice. Existentials in Genus help express such types in a type-safe way.

Associated types [9, 27] are type definitions required by type constraints. Encoding functionally dependent type parameters as associated types helps make certain type class headers less verbose [16]. Genus does not support associated types because they do not arise naturally as in other languages with traits [30, 33] or module systems [13] and because Genus code does not tend to need as many type parameters as in generic C++ code.

11. Conclusion

The Genus design is a novel and harmonious combination of language ideas that achieves a high degree of expressive power for generic programming while handling common usage patterns simply. Our experiments with using Genus to reimplement real software suggests that it offers an effective way to integrate generics into object-oriented languages, decreasing annotation burden while increasing type safety. Our benchmarks suggest the mechanism can be implemented with good performance. Future work includes proving type safety of Genus and exploring more efficient implementations.

Acknowledgments

We thank Owen Arden and Chinawat Isradisaikul for their help with implementation problems, Ross Tate, Doug Lea, and Sophia Drossopolou for their suggestions about the design, and Tom Magrino for suggesting the name Genus.

This work was supported by grant N00014-13-1-0089 from the Office of Naval Research, by MURI grant FA9550-12-1-0400, by a grant from the National Science Foundation (CCF-0964409), by the

German Federal Ministry of Education and Research (BMBF), grant 01IC12S01V, by the European Research Council, grant 321217, and by Quanta. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of any sponsor.

References

[1] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parameterization to the Java language. In *Proc. 12th OOPSLA*, pages 49–65, 1997.

[2] A. V. Aho, J. E. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1983.

[3] J. Aldrich. The power of interoperability: Why objects are inevitable. In *Proc. ACM Int'l Symp. on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, pages 101–116, 2013.

[4] M. H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[5] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. 13th OOPSLA*, Oct. 1998.

[6] K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *Proc. 12th European Conf. on Object-Oriented Programming*, number 1445 in Lecture Notes in Computer Science, pages 523–549, July 1998.

[7] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded polymorphism for object-oriented programming. In *Proc. Conf. on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[8] R. Cartwright and G. L. Steele Jr. Compatible genericity with run-time types for the Java programming language. In *Proc. 13th OOPSLA*, pages 201–215, Oct. 1998.

[9] M. M. T. Chakravarty, G. Keller, S. Peyton-Jones, and S. Marlow. Associated types with class. In *Proc. 32nd POPL*, pages 1–13, 2005.

[10] C. Chambers. Object-oriented multi-methods in Cecil. In O. L. Madsen, editor, *Proc. 20th European Conf. on Object-Oriented Programming*, volume 615, pages 33–56, 1992.

[11] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proc. 15th OOPSLA*, pages 130–145, 2000.

[12] M. Day, R. Gruber, B. Liskov, and A. C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proc. 10th OOPSLA*, pages 156–168, Oct. 1995. ACM SIGPLAN Notices 30(10).

[13] D. Dreyer, R. Harper, M. M. T. Chakravarty, and G. Keller. Modular type classes. In *Proc. 34th POPL*, pages 63–70, 2007.

[14] B. Emir, A. Kennedy, C. Russo, and D. Yu. Variance and generalized constraints for C# generics. In *Proc. 20th European Conf. on Object-Oriented Programming*, pages 279–303, 2006.

[15] findbugs-release. Findbugs. `http://findbugs.sourceforge.net/`.

[16] R. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A comparative study of language support for generic programming. In *Proc. 18th OOPSLA*, pages 115–134, 2003.

[17] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 3rd edition, 2005. ISBN 0321246780.

[18] B. Greenman, F. Muehlboeck, and R. Tate. Getting F-bounded polymorphism into shape. In *PLDI*, pages 89–99, 2014.

[19] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *Proc. 21st OOPSLA*, pages 291–310, 2006.

[20] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[21] A. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. In *PLDI*, pages 1–12, 2001.

[22] B. Liskov, A. Snyder, R. Atkinson, and J. C. Schaffert. Abstraction mechanisms in CLU. *Comm. of the ACM*, 20(8):564–576, Aug. 1977. Also in S. Zdonik and D. Maier, eds., *Readings in Object-Oriented Database Systems*.

[23] B. Liskov, R. Atkinson, T. Bloom, J. E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*. Springer-Verlag, 1984. Also published as Lecture Notes in Computer Science 114, G. Goos and J. Hartmanis, Eds., Springer-Verlag, 1981.

[24] T. Millstein, M. Reay, and C. Chambers. Relaxed MultiJava: Balancing extensibility and modular typechecking. In *Proc. 18th OOPSLA*, pages 224–240, 2003.

[25] D. R. Musser, G. J. Derge, and A. Saini. *The STL Tutorial and Reference Guide*. Addison-Wesley, 2nd edition, 2001. ISBN 0-201-37923-6.

[26] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *Proc. 24th POPL*, pages 132–145, Jan. 1997.

[27] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, 7(5), June 1995.

[28] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: an extensible compiler framework for Java. In *Proc. 12th Int'l Conf. on Compiler Construction (CC'03)*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152, 2003.

[29] M. Odersky. *The Scala Language Specification*. EPFL, 2014. Version 2.9.

[30] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *Proc. 25th OOPSLA*, pages 341–360, 2010.

[31] B. C. Oliveira, T. Schrijvers, W. Choi, W. Lee, and K. Yi. The implicit calculus: A new foundation for generic programming. In *PLDI*, pages 35–44, 2012.

[32] S. Peyton-Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, 1997.

[33] Rust. Rust programming language. http://doc.rust-lang.org/1.0.0-beta, 2015.

[34] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An introduction to Trellis/Owl. In *Proc. 1st OOPSLA*, Sept. 1986.

[35] J. G. Siek. The C++0x concepts effort. Arxiv preprint arXiv:1201.0027, Dec. 2011.

[36] J. G. Siek and A. Lumsdaine. Essential language support for generic programming. In *PLDI*, pages 73–84, 2005.

[37] J. G. Siek and A. Lumsdaine. A language for generic programming in the large. *Science of Computer Programming*, 76(5):423–465, 2011.

[38] M. Sulzmann, G. J. Duck, S. Peyton-Jones, and P. J. Stuckey. Understanding functional dependencies via constraint handling rules. *J. Funct. Program.*, 17(1):83–129, Jan. 2007.

[39] Swift. Swift programming language. https://developer.apple.com/swift/resources/, 2014.

[40] M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahé, G. Bracha, and N. Gafter. Adding wildcards to the Java programming language. In *Proc. 2004 ACM Symposium on Applied Computing*, SAC '04, pages 1289–1296, 2004.

[41] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. 16th POPL*, pages 60–76, 1989.

[42] A. Warth, M. Stanojević, and T. Millstein. Statically scoped object adaptation with expanders. In *Proc. 21st OOPSLA*, Oct. 2006.

[43] S. Wehr and P. Thiemann. JavaGI: The interaction of type classes with interfaces and inheritance. *ACM Trans. Prog. Lang. Syst.*, 33(4):12:1–12:83, July 2011.

[44] Y. Zhang, M. C. Loring, G. Salvaneschi, B. Liskov, and A. C. Myers. Genus: Making generics object-oriented, expressive, and lightweight. Technical Report http://hdl.handle.net/1813/39910, Cornell University, Apr. 2015.