Language Support for Generic Programming in Object-Oriented Languages: Peculiarities, Drawbacks, Ways of Improvement

Julia Belyakova

I. I. Vorovich Institute of Mathematics, Mechanics and Computer Science
Southern Federal University, Rostov-on-Don, Russia
julbel@sfedu.ru
http://mmcs.sfedu.ru/~juliet/

Abstract. Earlier comparative studies of language support for generic programming (GP) have shown that mainstream object-oriented (OO) languages such as C# and Java provide weaker support for GP as compared with functional languages such as Haskell or SML. But many new object-oriented languages have appeared in recent years. Have they improved the support for generic programming? And if not, is there a reason why OO languages yield to functional ones in this respect? In this paper we analyse language constructs for GP in seven modern object-oriented languages. We demonstrate that all of these languages follow the same approach to constraining type parameters, which has a number of inevitable problems. However, those problems are successfully lifted with the use of the another approach. Several language extensions that adopt this approach and allow to improve GP in OO languages are considered. We analyse the dependencies between different language features, discuss the features' support using both approaches, and propose which approach is more expressive.

Keywords: generic programming, object-oriented languages, programming language design, type parameters, constraints, interfaces, concepts, type classes, Concept pattern, multi-type constraints, multiple models, C#, Java, Scala, Ceylon, Kotlin, Rust, Swift, Haskell

1 Introduction

Most of the modern programming languages provide language support for generic programming (GP) [13]. As was shown in earlier comparative studies [4,7,8,14], some languages do it better than others. For example, Haskell is generally considered to be one of the best languages for generic programming [4,7], whereas mainstream object-oriented (OO) languages such as C# and Java are much less expressive and have many drawbacks [1,3]. But several new object-oriented languages have appeared in recent years, for instance, Rust, Swift, Kotlin. Have

The final publication is available at Springer via
http://dx.doi.org/10.1007/978-3-319-45279-1_1

```
interface IPrintable { string Print(); }

void PrintArr(IPrintable[] xs)
{ foreach (var x in xs) Console.WriteLine("{0}\n", x.Print()); }

string InParens<T>(T x) where T : IPrintable
{ return "(" + x.Print() + ")"; }
```

Fig. 1. An ambiguous role of C# interfaces

they improved the support for generic programming? To answer this question, we analyse seven modern OO languages with respect to their support for GP. It turns out that all of these languages follow the same approach to constraining type parameters, which we call the "Constraints-are-Types" approach. This approach is specific to object-oriented languages and has several inevitable limitations. The approach and its drawbacks are discussed in Sec. 2.

Sec. 3 provides a survey of the existing extensions [2, 3, 17, 24, 25] for object-oriented languages that address the limitations of OO languages [1] and improve the support for generic programming: all of them add new language constructs for constraining type parameters. We call the respective approach "Constraints-are-Not-Types". The advantages and shortcomings of this approach as compared with the basic one used in OO languages are discussed; yet we outline the design issues that need further investigation.

In conclusion, we argue that the "Constraints-are-Not-Types" approach is more expressive than the "Constraints-are-Types" one. Table 1 is a modified version of the well-known table [7, 8] showing the levels of language support for generic programming. It provides information on all of the object-oriented languages and extensions considered, introduces some new features, and demonstrates the relations between them.

2 Object-Oriented Approach to Constraining Type Parameters

We have explored *language constructs for generic programming* in seven modern object-oriented languages: C#, Java 8, Ceylon, Kotlin, Scala, Rust, Swift. As we will see, all of these languages adopt the same approach to constraining type parameters, which we call the "Constraints-are-Types" approach [3]. In this approach, interface-like constructs, which are normally used as types in object-oriented programming, are also used to constrain type parameters. By "interface-like constructs" we mean, in particular, interfaces in C#, Java, Ceylon, and Kotlin, traits in Scala and Rust, protocols in Swift. Fig. 1 shows a corresponding example in C#: `IPrintable` is an interface; it acts as a *type* in the array parameter `xs` in the `PrintArr` function, i. e. `xs` is an array of arbitrary values convertible to string, whereas in the `InParens<T>` function `IPrintable` is used to *constrain* the type parameter `T`. This example is not of particular interest, but it shows a common pattern of how constructs such as interfaces are used for generic

```
interface Equatable<T> {
  fun equal (other: T) : Boolean
  fun notEqual(other: T): Boolean { return !this.equal(other) }
}
class Ident (name : String) : Equatable<Ident> {
  val idname = name.toUpperCase()
  override fun equal (other:Ident): Boolean { return idname == other.idname }
}
fun <T : Equatable<T>> contains(vs : Array<T>, x : T): Boolean
{ for (v in vs)     if (v.equal(x)) return true;
  return false; }
```

Fig. 2. Interfaces and constraints in Kotlin

```
interface Comparable<Other> of Other
          given Other satisfies Comparable<Other> {
  formal Integer compareTo(Other other);
  Integer reverseCompareTo(Other other) { return other.compareTo(this); } }
```

Fig. 3. The use of "self type" in Ceylon interfaces

programming in OO languages. Sec. 2.1 provides a survey of similar constructs for GP in the modern object-oriented languages mentioned above. The main problems and drawbacks of the approach are discussed in Sec. 2.2.

2.1 Language Constructs for Constraining Type Parameters in Object-Oriented Languages

Interfaces in C#, Java and Kotlin. A classical interface describes methods and properties of a type that implements/extends the interface. In C# and Java 7 only signatures of instance methods are allowed inside the interface. Kotlin and Java 8 also support *default method implementations*. This is a useful feature for generic programming. For instance, one can define an interface for equality comparison that provides a default implementation for the inequality operation. Fig. 2 demonstrates corresponding Kotlin definitions: the `Ident` class implements the interface `Equatable<Ident>` that has two methods, `equal` and `notEqual`; as long as `notEqual` has a default implementation in the interface, there is no need to implement it in the `Ident` class.

Note that the `Equatable<T>` interface is generic: it takes the `T` type parameter that "pretends" to be a type implementing the interface, and this is indeed the case for the function `contains<T>` due to the "recursive" constraint `T : Equatable<T>`. The type parameter `T` is needed to solve the so-called binary method problem [5]: the `equal` method of the interface is expected to operate on two values of the same type (thus, `equal` is a "binary method"), with the first value being a receiver of `equal`, and the second value being a parameter of `equal`. `T` is an actual type of the `other` parameter, and it is supposed to be a type of the receiver.

```
struct Point { x: i32, y: i32, }
...
impl Point {
  fn moveOn(&self, dx: i32, dy: i32) -> Point
  { Point {x: self.x + dx, y: self.y + dy } }}
...
impl Point {
  fn reflect(&self) -> Point { Point {x: -self.x, y: -self.y} }}
...
let p1 = Point {x: 4, y: 3};
let p2 = p1.moveOn(1, 1);    let p3 = p1.reflect();
```

Fig. 4. `Point` struct and its methods in Rust

Interfaces in Ceylon. Ceylon interfaces are much similar to the Java 8 and Kotlin ones, but the Ceylon language also allows a declaration of a type parameter as a *self type*. An example is shown in Fig. 3. In the definition of the `Comparable<Other>` interface the declaration "`of Other`" explicitly requires `Other` to be a self type of the interface, i. e. a type that implements this interface. Because of this the `reverseCompareTo` method can be defined: the `other` and `this` values have the type `Other`, with the `Other` implementing `Comparable<Other>`, so the call `other.compareTo(this)` is perfectly legal. Without "`of Other`" the `Other` type can only be *supposed* to be a type of `this`, but this cannot be verified by a compiler, so the `reverseCompareTo` method cannot be written in Java 8 and Kotlin.

Scala Traits. Similarly to advanced interfaces in Java 8, Ceylon, and Kotlin, Scala traits [14, 15] support *default method implementations*. They can also have *abstract type* members, which, in particular, can be used as *associated types* [11, 16]. Associated types are types that are logically related to some entity. For instance, types of edges and vertices are associated types of a graph.

Just as in C#/Java/Ceylon/Kotlin, type parameters (and abstract types) in Scala can be constrained with traits and supertypes (upper bounds): the latter constraints are called *subtype constraints*. But, moreover, they can be constrained with subtypes (lower bounds), which are called *supertype constraints*. None of the languages we discussed so far support supertype constraints nor associated types. Another important Scala feature, implicits [15], will be mentioned later in Par. 2.2 with respect to the Concept design pattern.

Rust Traits. The Rust language is quite different from other object-oriented languages. There is no traditional class construct in Rust, but instead it suggests structs that store the data, and separate method implementations for structs. An example is shown in Fig. 4[1]: two `impl Point` blocks define method implementations for the `Point` struct. If a function takes the `&self`[2] argument (as `moveOn`), it

[1] Some details were omitted for simplicity. To make the code correct, one has to add `#[derive(Debug,Copy,Clone)]` before the `Point` definition.

[2] The "&" symbol means that an argument is passed by reference.

```
trait Equatable { fn equal(&self, that: &Self) -> bool;
  fn not_equal(&self, that: &Self) -> bool { !self.equal(that) } }
trait Printable { fn print(&self); }
...
impl Equatable for i32 {
  fn equal (&self, that: &i32) -> bool { *self == *that } }
...
struct Pair<S, T>{ first: S, second: T }
...
impl <S : Equatable, T : Equatable> Equatable for Pair<S, T> {
  fn equal (&self, that: &Pair<S, T>) -> bool
  { self.first.equal(&that.first) && self.second.equal(&that.second) } }
```

Fig. 5. An example of using Rust traits

is treated as a method. There can be any number of implementation blocks, yet they can be defined at any point after the struct declaration (even in a different module). This gives a huge advantage with respect to generic programming: any struct can be *retroactively* adapted to satisfy constraints. "Retroactively" means "later, after the point of definition". Constraints in Rust are expressed using traits. A trait defines which methods have to be implemented by a type similarly to Scala traits, Java 8 interfaces, and others. Traits can have *default method implementations* and *associated types*; besides that, a *self type* of the trait is directly available and can be used in method definitions. Fig. 5[3] demonstrates an example: the `Equatable` trait defining equality and inequality operations. Note how support for self type solves the binary method problem (here `equal` is a binary method): there is no need in extra type parameter that "pretends" to be a self type, because the self type `Self` is directly available.

Method implementations in Rust can be probably thought of similarly to .NET "extension methods". But in contrast to .NET[4], types in Rust also can *retroactively implement traits* in `impl` blocks as shown in Fig. 5: `Equatable` is implemented by `i32` and `Pair<S, T>`. The latter definition also demonstrates a so-called *type-conditional implementation*: pairs are equality comparable only if their elements are equality comparable. The constraint `<S : Equatable...` is a shorthand, it can be declared in a `where` section as well.

There is no struct inheritance and subtype polymorphism in Rust. Nevertheless, traits can be used as types, and due to this, a dynamic dispatch is provided. This feature is called trait objects in Rust. Suppose `i32` and `f64` implement the `Printable` trait from Fig. 5. Then the following code demonstrates creating and use of a polymorphic collection of values of the `&Printable` type (the type of the `polyVec` elements is a reference type):

```
let pr1 = 3; let pr2 = 4.5; let pr3 = -10;
let polyVec: Vec<&Printable> = vec![&pr1, &pr2, &pr3];
```

[3] Some details were omitted for simplicity. The following declaration is to be provided to make the code correct: `#[derive(Copy, Clone)]` before the definition `struct Pair<S : Copy, T : Copy>`. Yet the type parameters of the `impl` for pair must be constrained with `Copy+Equatable`.

[4] Similarly to .NET, Kotlin supports extending classes with methods and properties, but interface implementation in extensions is not allowed.

```
protocol  Equatable { func equal(that: Self) -> Bool; }
extension Equatable { func notEqual(that: Self) -> Bool
{ return !self.equal(that) }}
func contains<T : Equatable> (values: [T], x:T) -> Bool { ... }

protocol Printable { func print(); }
extension Int : Printable { ... }

protocol Container { associatedtype ItemTy ... }
func allItemsMatch<C1: Container, C2: Container
                   where C1.ItemTy == C2.ItemTy, C1.ItemTy: Equatable> ...
```

Fig. 6. Protocols and their use in Swift

```
interface ITerm<Tm> { IEnumerable<Tm> Subterms(); ... }

interface IEquation<Tm, Eqtn, Subst>        where Tm : ITerm<Tm>
    where Eqtn : IEquation<Tm, Eqtn, Subst>
    where Subst : ISubstitution<Tm, Eqtn, Subst>
{ Subst Solve();
  IEnumerable<Eqtn> Split();  ... }

interface ISubstitution<Tm, Eqtn, Subst>    where Tm : ITerm<Tm>
    where Eqtn : IEquation<Tm, Eqtn, Subst>
    where Subst : ISubstitution<Tm, Eqtn, Subst>
{ Tm SubstituteTm(Tm);
  IEnumerable<Eqtn> SubstituteEq (IEnumerable<Eqtn>); ... }
```

Fig. 7. The C# interfaces for the Unification algorithm

```
for v in polyVec { v.print(); }
```

Swift Protocols. Swift is a more conventional OO language than Rust: it has classes, inheritance, and subtype polymorphism. Classes can be extended with new methods using extensions that are quite similar to Rust method implementations. Instead of interfaces and traits Swift provides protocols. They cannot be generic but support *associated types* and *same-type* constraints, *default method implementations* through protocol extensions, and explicit access to a *self type*; due to the mechanism of extensions, types can *retroactively* adopt protocols. Fig. 6 illustrates some examples: the `Equatable` protocol extended with a default implementation for `notEqual` (pay attention to the use of the `Self` type); the `contains<T>` generic function with a protocol constraint on the type parameter `T`; an extension of the type `Int` that enables its conformance to the `Printable` protocol; the `Container` protocol with the associated type `ItemTy`; the `allItemsMatch` generic function with the same-type constraint on types of elements of two containers, `C1` and `C2`.

2.2 Drawbacks of the "Constraints-are-Types" Approach

The Problem of Multi-Type Constraints. Constructs such as interfaces or traits, which are used both as types in object-oriented code and constraints on type parameters in generic code, describe an interface of a *single* type. And this

has inevitable consequence: *multi-type constraints* (constraints on several types) cannot be expressed naturally. Consider a generic unification algorithm [12]: it takes a set of equations between terms (symbolic expressions), and returns the most general substitution which solves the equations. So the algorithm operates on three kinds of data: terms, equations, substitutions. A signature of the algorithm might be as follows:

```
Subst Unify<Tm, Eqtn, Subst> (IEnumerable<Eqtn>)
```

But a bunch of functions have to be provided to implement the algorithm:
`Subterms : Tm` → `IEnumerable<Tm>`, `Solve : Eqtn` → `Subst`,
`SubstituteTm : Subst` × `Tm` → `Tm`,
`SubstituteEq : Subst` × `IEnumerable<Eqtn>` → `IEnumarable<Eqtn>`,
and some others. All these functions are needed for unification at once, hence it would be convenient to have a single constraint that relates all the type parameters and provides the functions required:

```
Subst Unify<Tm, Eqtn, Subst> (IEnumerable<Eqtn>)
    where <single constraint>
```

But in the languages considered in the previous section the only thing one can do[5] is to define three different interfaces for terms, equations, and substitution, and then separately constrain every type parameter of the `Unify<>` with a respective interface. Fig. 7 shows the C# interface definitions. To set up a relation between mutually dependent interfaces, several type parameters are used: `Tm` for terms, `Eqtn` for equations, and `Subst` for substitution. The parameters are repeatedly constrained with the appropriate interfaces in every interface definition. Those constraints are to be stated in a signature of the unification algorithm as well:

```
Subst Unify<Tm, Eqtn, Subst> (IEnumerable<Eqtn>)
    where Tm : ITerm<Tm>
    where Eqtn : IEquation<Tm, Eqtn, Subst>
    where Subst : ISubstitution<Tm, Eqtn, Subst>
```

There is one more thing to notice here — interfaces are used in both roles in the same piece of code: the `IEnumerable<Eqtn>` interface is used as a type, whereas other interfaces in the `where` sections are used as constraints. So the semantics of the `interface` construct is *ambiguous*.

The Lack of Language Support for Multiple Models. For simplicity, in this part of the paper we call "constraint" any language construct that is used to describe constraints, while the way in which types satisfy the constraints we call "model". All of the object-oriented languages considered earlier allow having only one, unique model of a constraint for the given set of types. And indeed this makes sense for the languages where "Constraints-are-Types" philosophy works, because it is not clear what to do with types that could implement interfaces (or any other similar constructs) in several ways. But how does this affect generic programming? It turns out that sometimes it is desirable to have multiple models

[5] The Concept design pattern can also be used, but it has its own drawbacks. We will discuss concept pattern later, in Par. 2.2.

```
// Type Parameter Constraints
interface IComparable<T> { int CompareTo(T other); }
void Sort<T>(T[] values) where T : IComparable<T> { ... }
class SortedSet<T> where T : IComparable<T> { ... }

// Concept Pattern
interface IComparer<T>  { int Compare(T x, T y); }
void Sort<T>(T[] values, IComparer<T> cmp) { ... }
class SortedSet<T> { private IComparer<T> cmp; ...
  public SortedSet(IComparer<T> cmp) { ... } ... }
```

Fig. 8. The use of the Concept design pattern in C#

of a constraint for the same set of types. For instance, one could imagine sets of strings with case-sensitive and case-insensitive equality comparison; another common example is the use of different orderings on numbers, yet different graph implementations, and so on. Thus, with respect to generic programming, the absence of multiple models is rather a problem than a benefit. Without extending the language the problem of multiple models can be solved in two ways:

1. Using the Adapter pattern. If one wants the type `Foo` to implement the interface `IEquatable<Foo>` in a different way, an adapter of `Foo`, the `Foo1` that implements `IEquatable<Foo1>` can be created. This adapter then can be used instead of `Foo` whenever the `Foo1`-style comparison is required. An obvious shortcoming of this approach is the need to repeatedly wrap and unwrap `Foo` values; in addition, code becomes cumbersome.
2. Using the Concept pattern, which is considered below.

Concept Pattern. The Concept design pattern [15] eliminates two problems:

1. First, it enables *retroactive modeling* of constraints, which is not supported in languages such as C#, Java, Ceylon, Kotlin, or Scala.
2. Second, it allows defining *multiple models* of a constraint for the same set of types.

The idea of the Concept pattern is as follows: instead of constraining type parameters, generic functions and classes take extra arguments that provide a required functionality — "concepts". Fig. 8 shows an example: in the case of the Concept pattern the constraint `T : IComparable<T>` is replaced with an extra argument of the type `IComparer<T>`. The `IComparer<T>` interface represents a concept of comparing: it describes an interface of an object that can compare values of the type `T`. As long as one can define several classes implementing the same interface, different "models" of the `IComparer<T>` "concept" can be passed into `Sort<T>` and `SortedSet<T>`.

This pattern is widely used in generic libraries of mainstream object-oriented languages such as C# and Java; it is also used in Scala. Due to implicits [14, 15], the use of the Concept pattern in Scala is a bit easier: in most cases an appropriate "model" can be found by a compiler implicitly, so there is no need

to explicitly pass it at a call site[6]. Nevertheless, the pattern has two substantial drawbacks. First of all, it brings *run-time overhead*, because every object of a generic class with constraints has at least one extra field for the "concept", while generic functions with constraints take at least one extra argument. The second drawback, which we call *models-inconsistency*, is less obvious but may lead to very subtle errors. Suppose we have `s1` of the type `HashSet<String>` and `s2` of the *same* type, provided that `s1` uses case-sensitive equality comparison, `s2` — the case-insensitive one. Thus, `s1` and `s2` use different, inconsistent models of comparison. Now consider the following function:

```
static HashSet<T> GetUnion<T>(HashSet<T> a, HashSet<T> b)
{ var us = new HashSet<T>(a, a.Comparer); us.UnionWith(b); return us; }
```

Unexpectedly, the result of `GetUnion(s1, s2)` could differ from the result of `GetUnion(s2, s1)`. Despite the fact that `s1` and `s2` have the same type, they use different comparators, so the result depends on which comparator was chosen to build the union. Comparators are run-time objects, so the models-consistency *cannot* be checked at *compile time*.

3 The "Constraints-are-Not-Types" Approach to Constraining Type Parameters

In contrast to object-oriented languages discussed in Sec. 2, type classes [10] in the Haskell language are not used as types, they are used as *constraints only*. Inspired by the design of type classes, several language extensions for C# and Java have been developed. For defining constraints all these extensions suggest *new language constructs* that have *no self types* and *cannot* be used as types. They describe requirements on type parameters in an external way; therefore, retroactive constraints satisfaction (*retroactive modeling*) is automatically provided. Besides retroactive modeling, an integral advantage of such kind of constructs is that *multi-type constraints* can be easily and naturally expressed using them; yet there is no semantic ambiguity which arises when the same construct, such as a C# interface, is used both as a type and constraint, as in the example below:

```
void Sort<T>(ICollection<T>) where T : IComparable<T>;
```

Here `ICollection<T>` and `IComparable<T>` are generic interfaces, but the former is used as a type whereas the latter is used as constraint.

3.1 Language Extensions for Object-Oriented Languages

JavaGI Generalized Interfaces. JavaGI [24] provides *multi-headed* generalized interfaces that adopt several features from Haskell type classes [23] and describe interfaces of several types. There is no self type in such interface, it cannot be

[6] Scala is often blamed for its complex rules of implicits resolution: sometimes it is not clear which implicit object is to be used.

```
interface UNIFY [Tm, Eqtn, Subst] {
  receiver Tm    { IEnumerable<Tm> Subterms(); ... }
  receiver Eqtn  { IEnumerable<Eqtn> Split();  ... }
  receiver Subst { Tm SubstituteTm(Tm); ...        }}
Subst Unify<Tm, Eqtn, Subst>(Enumerable<Eqtn>)
  where [Tm, Eqtn, Subst] implements UNIFY {...}
```

Fig. 9. Generalized interfaces in JavaGI

used as a type. An example of multi-headed interface is shown in Fig. 9: the `UNIFY` interface contains all the functions required by the unification algorithm considered in Sec. 2.2; the requirements on three types (term, equation, substitution) are defined at once in a single interface. Note how succinct is this definition as compared with the one in Fig. 7.

Language G and C++ Concepts. Concept as an explicit language construct for defining constraints on type parameters was initially introduced in 2003 [19]. Several designs have been developed since that time [6, 20, 21]; in the large, the expressive power of concepts is rather close to the Haskell type classes [4]. Concepts were designed to solve the problems of unconstrained C++ templates [1,18]. A new version of concepts, Concepts Lite (C++1z) [22], is under way now. The language G declared as "a language for generic programming" [17] also provides concepts that are very similar to the C++0x concepts. Similarly to a type class, a concept defines a set of requirements on one or more type parameters. It can contain *function signatures* that may be accompanied with *default implementations*, *associated types*, nested *concept-requirements* on associated types, and *same-type constraints*. A concept can *refine* one or more concepts, it means that the refining concept includes all the requirements from the refined concepts. Refinement is very similar to multiple interface inheritance in C# or protocol inheritance in Swift. Due to the concept refinement, a so-called *concept-based overloading* is supported: one can define several versions of an algorithm/class that have different constraints, and then at compile time the most specialized version is chosen for the given instance. The C++ `advance` algorithm for iterators is a classic example of concept-based overloading application.

It is said that a type (or a set of types) *satisfies* a concept if an appropriate model of the concept is defined for this type (types). Model definitions are independent from type definitions, so the modeling relation is established *retroactively*; models can be generic and *type-conditional*.

C# with Concepts. In the C#$^{\text{cpt}}$ project [3] (C# with concepts) concept mechanism integrates with subtyping: type parameters and associated types can be constrained with *supertypes* (as in basic C#) and also with *subtypes* (as in Scala). In contrast to all of the languages we discussed earlier, C#$^{\text{cpt}}$ allows *multiple models* of a concept in the same scope. Some examples are shown in Fig. 10: the `CEquatable[T]` concept with the `Equal` signature and a default implementation of `NotEqual`, the generic interface `ISet<T>` with concept-requirement on the type

```
concept CEquatable[T] { bool Equal(T x, T y);
  bool NotEqual(T x, T y) { return !Equal(x, y); }}

interface ISet<T> where CEquatible[T] { ... }

model default StringEqCaseS for CEquatable[String] { ... }
model StringEqCaseIS for CEquatable[String] { ... }

bool Contains<T>(IEnumerable<T> values, T x)
  where CEquatable[T] using CEq {... if (cEq.Equal(...) ...}
```

Fig. 10. Concepts and models in C#^{cpt}

```
constraint Eq[T] { boolean T.equals(T other); }
constraint GraphLike[V, E] { V E.source(); ... }

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

model CIEq for Eq[String] { ... } // case-insensitive model

model DualGraph[V,E] for GraphLike[V,E] where GraphLike[V,E] g
{ V E.source() { return this.(g.sink)(); } ... }
```

Fig. 11. Constraints and models in Genus

parameter `T`, and two models of `CEquatable[]` for the type `String` — for case-sensitive and case-insensitive equality comparison. The first model is marked as a *default* model[7]: it means that this model is used if a model is not specified at the point of instantiation. For instance, in the following code `StringEqCaseS` is used to test equality of strings in `s1`.

```
ISet<String> s1 = ...;
ISet<String>[using StringEqCaseIS] s2 = ...;
s1 = s2; // Static ERROR, s1 and s2 have different types
```

Note that `s1` and `s2` have *different types* because they use different models of `CEquatible[String]`. Models are compile-time artefacts, so the models-consistency is checked at compile time. One more interesting thing about C#^{cpt}: concept-requirements can be named. In the `Contains<T>` function (Fig. 10) the name `cEq` is given to the requirement on `T`; this name is used later in the body of `Contains<T>` to access the `Equal` function of the concept. It is also worth mention that the interface `IEnumerable<T>` is used as a type along with the concept `CEquatable[T]` being used as a constraint; thus, the role of interfaces is not ambiguous any more, interfaces and concepts are independently used for different purposes.

Constraints in Genus. Like G concepts and Haskell type classes, constraints in Genus [25] (an extension for Java) are used as constraints only. Fig. 11 demonstrates some examples: the `Eq[T]` constraint, which is used to constrain the `T` in the `Set[T]` interface; the model of `Eq[String]` for case-insensitive equality comparison; the multi-parameter constraint `GraphLike[V, E]`, and the type-

[7] The default model can be generated automatically for a type if the type conforms to a concept, i. e. it provides methods required by the concept.

conditional generic model `DualGraph[V,E]`. Methods in Genus classes/interfaces can impose additional constraints:

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

Here the `List[]` interface can be instantiated by any type, but the `remove` method can be used only if the type `E` of elements satisfies the `Eq[E]` constraint. This feature is called *model genericity*.

Just as C#$^{\text{cpt}}$, Genus supports *multiple models* and automatic generation of the *natural* model, which is the same thing as the default model in C#$^{\text{cpt}}$. Models-consistency can also be checked at compile time. In Genus this feature is called *model-dependent types*. As well as in C#$^{\text{cpt}}$, constraint-requirements in Genus can be named; the example is shown in Fig. 11: `g` is a name of the `GraphLike[V,E]` constraint required by the `DualGraph[V,E]` model.

4 Conclusion and Future Work

	Haskell	C#	Java 8	Scala	Ceylon	Kotlin	Rust	Swift	JavaGI	G	C#$^{\text{cpt}}$	Genus
Constraints can be used as types	○	●	●	●	●	●	●	●	◐	○	○	○
Explicit self types	–	○	○	◐	●⁻	○	●	●	◐	–	–	–
Multi-type constraints	●	⚹	⚹	⚹	○	⚹	○	○	●	●	●	●
Retroactive type extension	○	●	○	○	○	●	●	●	○	○	○	○
Retroactive modeling	●	⚹	⚹	⚹	○	⚹	●	●	●	●	●	●
Type conditional models	●	○	○	○	○	○	●	○	●	●	●	●
Static methods	●[a]	○	●	○	●	●	●	●	●	●[a]	●[a]	●[a]
Default method implementation	●	○	●	●	●	●	●	●	◐	●	●	○
Associated types	●	○	○	●	○	○	●	●	○	●	●	○
Constraints on associated types	◐	–	–	●	–	–	●	●	–	●	●	–
Same-type constraints	◐	–	–	●	–	–	●	●	–	●	●	–
Subtype constraints	–	●	●	●	●	●	–	●	○	○	●	○
Supertype constraints	–	○	○	●	○	○	–	○	○	○	●	○
Constraints refinement	●	●	●	●	●	●	●	●	●	●	●	●
Concept-based overloading	○	○	○	○	○	○	●	○	○	◐[b]	○	○
Multiple models	◐[c]	⚹	⚹	⚹	⚹	⚹	○	○	○	◐[d]	●	●
Models-consistency (model-dependent types)	–[e]	○	○	○	○	○	–[e]	–[e]	–[e]	–[e]	●	●
Model genericity	–	⚹	⚹	⚹	⚹	⚹	●	○	○	○	○	●

[a] Constraints constructs have no self types, therefore, any function member of a constraint can be treated as static function.
[b] C++0x concepts, in contrast to G concepts, provide full support for concept-based overloading.
[c] Partially supported with OverlappingInstances extension.
[d] G supports lexically-scoped models but not really multiple models.
[e] If multiple models are not supported, the notion of model-dependent types does not make sense.

Table 1. The levels of support for generic programming in OO languages

Taking into consideration what we have found out in Sec. 2 and Sec. 3, we draw a conclusion that there are merely two language features concerning generic programming that cannot be incorporated in an object-oriented language *together*:

1. the use of a construct both as a type and constraint;
2. natural support for multi-type constraints.

Using the "Constraints-are-Types" approach, the first feature can be supported, but not the second; using the "Constraints-are-Not-Types" approach, vice versa. Can we choose one feature that is more important? The answer is yes. It was shown in the study [9] that in practice interfaces that are used as constraints (such as `IComparable<T>` in C# or `Comparable<X>` in Java) are almost never used as types: authors had checked about 14 millions lines of Java code and found only one such example, which was even rewritten and eliminated. At the same time, multi-type constraints, which can be so naturally expressed under the "Constraints-are-Not-Types" approach, have rather awkward and cumbersome representation in the "Constraints-are-Types" approach. Furthermore, the Concept design pattern used in OO languages to provide the support for multiple models has serious pitfalls, whereas with the "Constraints-are-Not-Types" approach models-consistency can be ensured at compile-time if multiple models are allowed. All other language facilities we discussed could be supported under any approach. Therefore, we claim that *the "Constraints-are-Not-Types" approach is preferable*. Without sacrificing OO features, object-oriented languages can be extended with new language constructs for constraining type parameters to improve the support for generic programming. Nevertheless, further study is needed to identify an effective design and implementation of such extension. The existing designs that support multiple models, C#$^{\text{cpt}}$ and Genus, have at least one essential shortcoming: constraints on type parameters are declared in "predicate-style" rather than "parameter-style". In Haskell, G, C#, Java, Rust, and many other languages, where only one model of a constraint is allowed for the given set of types, constraints on type parameters are indeed predicates: types either satisfy the constraint (if they have a model that is unique) or not. But in Genus and C#$^{\text{cpt}}$ constraints *are not predicates*, they are actually *parameters*, as long as different models of constraints can be used. Unfortunately, the "predicate-style" syntax does not correspond to this semantics. It misleads a programmer and makes it more difficult to write and call generic code. Features such as multiple dynamic dispatch, concept variance, and typing rules in presence of concept parameters are also to be investigated.

Table 1 provides a summary on comparison of the OO languages and language extensions considered: each row corresponds to one property important for generic programming; each column shows levels of support of the properties in one language. Black circle ● indicates full support of a property, ◐ — partial support, ○ means that a property is not supported at language level, ⚹ means that a property is emulated using the Concept pattern, and the "–" sign indicates that a property is not applicable to a language. Related properties are grouped within horizontal lines; some of them, such as "using constraints as

types" and "natural language support for multi-type constraints" are mutually exclusive. The major features analysed in the paper are highlighted in bold. The purpose of this table is to show dependencies between different properties and to graphically demonstrate that the "Constraints-are-Not-Types" approach is more powerful than the "Constraints-are-Types" one. There are some features that can be expressed under any approach, such as static methods, default method implementations, associated types [11], and even type-conditional models.

Acknowledgment

The authors would like to thank Artem Pelenitsyn, Jeremy Siek, and Ross Tate for helpful discussions on generic programming.

References

1. Belyakova, J., Mikhalkovich, S.: A Support for Generic Programming in the Modern Object-Oriented Languages. Part 1. An Analysis of the Problems. Transactions of Scientific School of I.B. Simonenko. Issue 2 (2), 63–77 (in Russian) (2015)
2. Belyakova, J., Mikhalkovich, S.: A Support for Generic Programming in the Modern Object-Oriented Languages. Part 2. A Review of the Modern Solutions. Transactions of Scientific School of I.B. Simonenko. Issue 2 (2), 78–92 (in Russian) (2015)
3. Belyakova, J., Mikhalkovich, S.: Pitfalls of C# Generics and Their Solution Using Concepts. Proceedings of the Institute for System Programming 27(3), 29–45 (Jun 2015)
4. Bernardy, J.P., Jansson, P., Zalewski, M., Schupp, S., Priesnitz, A.: A Comparison of C++ Concepts and Haskell Type Classes. In: Proceedings of the ACM SIGPLAN Workshop on Generic Programming. pp. 37–48. WGP '08, ACM, New York, NY, USA (2008)
5. Bruce, K., Cardelli, L., Castagna, G., Leavens, G.T., Pierce, B.: On Binary Methods. Theor. Pract. Object Syst. 1(3), 221–242 (Dec 1995), http://dl.acm.org/citation.cfm?id=230849.230854
6. Dos Reis, G., Stroustrup, B.: Specifying C++ Concepts. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 295–308. POPL '06, ACM, New York, NY, USA (2006)
7. Garcia, R., Jarvi, J., Lumsdaine, A., Siek, J., Willcock, J.: An Extended Comparative Study of Language Support for Generic Programming. J. Funct. Program. 17(2), 145–205 (Mar 2007)
8. Garcia, R., Jarvi, J., Lumsdaine, A., Siek, J.G., Willcock, J.: A comparative study of language support for generic programming. SIGPLAN Not. 38(11), 115–134 (Oct 2003), http://doi.acm.org/10.1145/949343.949317
9. Greenman, B., Muehlboeck, F., Tate, R.: Getting F-bounded Polymorphism into Shape. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 89–99. PLDI '14, ACM, New York, NY, USA (2014)
10. Hall, C.V., Hammond, K., Peyton Jones, S.L., Wadler, P.L.: Type classes in haskell. ACM Trans. Program. Lang. Syst. 18(2), 109–138 (Mar 1996), http://doi.acm.org/10.1145/227699.227700

11. Järvi, J., Willcock, J., Lumsdaine, A.: Associated Types and Constraint Propagation for Mainstream Object-oriented Generics. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. pp. 1–19. OOPSLA '05, ACM, New York, NY, USA (2005)
12. Martelli, A., Montanari, U.: An Efficient Unification Algorithm. ACM Trans. Program. Lang. Syst. 4(2), 258–282 (Apr 1982), http://doi.acm.org/10.1145/357162.357169
13. Musser, D.R., Stepanov, A.A.: Generic Programming. In: Proceedings of the International Symposium ISSAC'88 on Symbolic and Algebraic Computation. pp. 13–25. ISAAC '88, Springer-Verlag, London, UK, UK (1989), http://dl.acm.org/citation.cfm?id=646361.690581
14. Oliveira, B.C., Gibbons, J.: Scala for Generic Programmers: Comparing Haskell and Scala Support for Generic Programming. J. Funct. Program. 20(3-4), 303–352 (Jul 2010)
15. Oliveira, B.C., Moors, A., Odersky, M.: Type Classes As Objects and Implicits. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 341–360. OOPSLA '10, ACM, New York, NY, USA (2010)
16. Pelenitsyn, A.: Associated Types and Constraint Propagation for Generic Programming in Scala. Programming and Computer Software 41(4), 224–230 (2015)
17. Siek, J.G., Lumsdaine, A.: A language for generic programming in the large. Sci. Comput. Program. 76(5), 423–465 (May 2011), http://dx.doi.org/10.1016/j.scico.2008.09.009
18. Stepanov, A.A., Lee, M.: The Standard Template Library. Technical Report 95-11(R.1), HP Laboratories (November 1995)
19. Stroustrup, B.: Concept Checking — A More Abstract Complement to Type Checking. Technical Report N1510=03-0093, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers (October 2003)
20. Stroustrup, B., Dos Reis, G.: Concepts — Design Choices for Template Argument Checking. Technical Report N1522=03-0105, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers (October 2003)
21. Stroustrup, B., Sutton, A.: A Concept Design for the STL. Technical Report N3351=12-0041, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers (January 2012)
22. Sutton, A.: C++ Extensions for Concepts PDTS. Technical Specification N4377, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers (February 2015)
23. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 60–76. POPL '89, ACM, New York, NY, USA (1989), http://doi.acm.org/10.1145/75277.75283
24. Wehr, S., Thiemann, P.: JavaGI: The Interaction of Type Classes with Interfaces and Inheritance. ACM Trans. Program. Lang. Syst. 33(4), 12:1–12:83 (Jul 2011), http://doi.acm.org/10.1145/1985342.1985343
25. Zhang, Y., Loring, M.C., Salvaneschi, G., Liskov, B., Myers, A.C.: Lightweight, Flexible Object-oriented Generics. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 436–445. PLDI 2015, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2737924.2738008