Language Support for Generic Programming in Object-Oriented Languages: Design Challenges

Julia Belyakova
Institute for Mathematics, Mechanics
and Computer Science
named after I. I. Vorovich
Southern Federal University
Rostov-on-Don, Russia
Email: julbel@sfedu.ru

***Abstract*—It is generally considered that object-oriented (OO) languages provide weaker support for generic programming (GP) as compared with functional languages such as Haskell or SML. There were several comparative studies which showed this. 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 the earlier comparative studies object-oriented languages were usually not treated in any special way. However, the OO features affect language facilities for GP and a style people write generic programs in such languages. In this paper we compare ten modern object-oriented languages and language extensions with respect to their support for generic programming. It has been discovered that every of these languages strictly follows one of the two approaches to constraining type parameters. So the first design challenge we consider is "which approach is better". It turns out that most of the explored OO languages use the less powerful one. The second thing that has a big impact on the expressive power of a programming language is language support for multiple models. We discuss pros and cons of this feature and its relation to other language facilities for generic programming.**

I. Introduction

Almost all modern programming languages provide language support for generic programming (GP) [2]. Some languages do it better than others. For example, Haskell is generally considered to be one of the best languages for generic programming [3, 4], whereas mainstream object-oriented languages such as C# and Java are much less expressive and have many drawbacks. There were several studies that compared language support for generic programming in different languages [3–6]. However, these studies do not make any difference between object-oriented and functional languages. We argue that OO languages are to be treated separately, because they support the distinctive OO features that pure functional languages do not, such as inheritance, interfaces/traits, subtype polymorphism, etc. These features affect the language design and a way people write generic programs in object-oriented languages.

Several new object-oriented languages have appeared in recent years, for instance, Rust, Swift, Kotlin. At the same time, several independent extensions have been developed for the mainstream OO languages [7–10]. These new languages and extensions have many differences, but all of them tend to improve the support for generic programming. There is a lack of a careful comparison of the approaches and mechanisms for generic programming in *modern object-oriented* languages. This study is aimed to fill the gap: it gives a survey, analysis, and comparison of the facilities for generic programming that the chosen OO languages provide. We identify the dependencies between major language features, detect incompatible ones, and point the properties that a language design should satisfy to be effective for generic programming.

This paper is the extended version of the conference paper [1] accepted for the XX Brazilian Symposium on Programming Languages.

II. Main ideas

Ten modern object-oriented languages and language extensions have been explored in this study with respect to generic programming. We have found out that in the case of OO languages there are exactly two approaches to a design of language constructs for generic programming. We call the first one "constraints-are-types", because under this approach OO constructs such as interfaces or traits, which are usually used as types in object-oriented programs, are also used to constrain type parameters in generic programs. The second approach, "constraints-are-Not-types", restricts OO constructs to be used as types only, and provides separate language constructs for constraining type parameters. Hence the first design challenge arises: is one of this approaches better than another? Or the same expressive power can be achieved using any of them? We answer these questions in Sec. III. It turns out that the approaches cannot be integrated together, and the second one is more expressive.

The second point covered in the paper in detail (in Sec. IV) is language support for multiple models (by "model" we mean a way in which types satisfy constraints). There are several questions related to multiple models:

1) Is it desirable to have multiple models of a constraint?
2) How can support for multiple models be provided with the approaches discovered?
3) Why does not Haskell allow multiple models (instances of a type class)?

```
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

```
interface IComparable<T> { int CompareTo(T other); }

class SortedSet<T>  where T : IComparable<T> { ... }
```

Fig. 2. The `IComparable<T>` interface in C#

```
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. 3. The C# interfaces for unification algorithm

4) Is there a language design that reflects the support for multiple models better than the existing ones?

The short answers are:

1) Yes, it is desirable.
2) It can be naturally provided with the second approach but not with the first one.
3) Because of type inference.
4) Yes, there is.

In conclusion, we present a modified version of the well-known table [3, 5] showing the levels of language support for the features important for generic programming. Table I provides information on all of the object-oriented languages considered, introduces some new features, and demonstrates the relations between the features.

III. Two approaches to constraining type parameters

This section provides a survey of *language constructs for generic programming* in several modern *object-oriented* programming languages as well as some language extensions. All of the languages we explored adopt one of the two approaches:

1) 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, C#/Java interfaces, Scala traits, Swift protocols, Rust traits. Fig. 1 shows a corresponding example in C#: `IPrintable` interface acts as the type of `xs` in `PrintArr`, whereas in the function `InParens<T>` it is used to constrain its type parameter `T`.
2) For constraining type parameters a separate language construct is provided, and such construct cannot be used as a type. We will see some examples in Sec. III-B.

In Sec. III-A languages of the first category are analysed; Sec. III-B is devoted to the second one. In Sec. III-C we compare both approaches and answer the question "Which one is better if any?".

A. Languages with "Constraints-are-Types" Philosophy

C# and Java are probably the best-known programming languages in this category, with interfaces being used to constrain type parameters. In comparison with other languages that support generic programming, these ones are much less expressive and have several considerable drawbacks.

- *Lack of retroactive interface implementation.* After a type had been defined, it cannot implement any new interface. A consequence is that generic code with constraints on type parameters can only be instantiated with types *originally* designed to satisfy these constraints. It is impossible to adapt types afterwards, even if they semantically conform the constraints.
- *Drawbacks of F-bounded polymorphism.* F-bounded polymorphism [11] allows "recursive" constraints (F-constraints) on type parameters in the form `T : I<T>`, where `T` is a type parameter, `I<>` is a generic interface. Such kind of constraints solves the binary method problem [12]: Fig. 2 demonstrates a corresponding C# [13] example. The type parameter `T` in the interface `IComparable<T>` pretends to be a type that implements this interface. This is indeed the case for the class `SortedSet<T>` due to the constraint `T : IComparable<T>`, so the method `T.CompareTo(T)` is like a binary function for comparing elements of type `T`. But the semantics of `IComparable<T>` itself has nothing to do with binary methods. One could easily write some class `Foo` implementing `IComparable<Bar>`, and thus the semantics of comparing two `Bar`s would be broken. Another shortcoming of the F-bounded polymorphism is that code with recursive constraints is rather cumbersome and difficult to understand. Yet, as we will see, the F-bounded polymorphism is not the only solution to the binary method problem. More detailed discussion on the pitfalls of the F-bounded polymorphism can be found in [9, 14].
- *Lack of associated types* [14, 15]. Types that are logically related to some entity are often called associated types of the entity. For instance, types of edges and vertices are associated types of a graph. There is no specific language support for associated types in C# and Java: such types are expressed in generic code in the form of extra type parameters.
- *Lack of constraints propagation* [14, 15]. Despite the fact that the definition of the class `SortedSet<T>` in Fig. 2 already contains a constraint on the type parameter `T`, in the `baz<T>` function defined below the constraint on `T` is

to be placed as well.

```
void baz<T>(SortedSet<T> s)
  where T : IComparable<T>{ ... }
```

Although `baz<T>` takes a value of type `SortedSet<T>`, so it is clear from the signature of the function that `T` must be comparable, the code would not compile without an explicit constraint. In other words, a compiler does not propagate the constraints implied by formal parameters, this is a programmer's burden.

Some of the drawbacks mentioned above have been successfully eliminated in the modern object-oriented languages. We briefly examine language facilities for generic programming in several OO languages with the "constraints-are-types" philosophy in the following subsections. But there is a problem common for all languages of this category, the problem of *multi-type constraints* (constraints on several types). Note that an interface (or a similar language construct) describes properties, an interface of a *single* type that implements/extends it. This has inevitable consequence: multi-type constraints cannot be expressed naturally. Consider a generic unification algorithm [16]: 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 has 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 C#/Java the only thing one can do[1] is to define three different interfaces describing a term, equation and substitution, and then separately constrain every type parameter of the `Unify<>` function with a respective interface. Fig. 3 shows the interface definitions. To set up a relation between mutually dependent interfaces, three type parameters are used: `Tm` for terms, `Eqtn` for equations, and `Subst` for substitution. Moreover, the parameters are repeatedly constrained with the appropriate interfaces in every interface definition. These 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>` one is used as a type, whereas in the where sections interfaces are used as constraints.

[1]The Concept design pattern can also be used, but it has its own drawbacks. We will discuss concept pattern later, in Sec. IV-C2.

```
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 }}
```

Fig. 4. Interfaces and constraints in Kotlin

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

Fig. 5. The use of "self type" in Ceylon interfaces

1) Interfaces in Ceylon and Kotlin: In contrast to C#, Ceylon [17] and Kotlin [18] interfaces support *default method implementation*, so Java 8 [19] interfaces do. This is a useful feature for generic programming. For instance, one can define an interface for equality that provides a default implementation for the inequality operation. Fig. 4 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 again in the definition of the `Ident` class.

In addition to default method implementations, the Ceylon language also allows a type parameter to be declared as a *self type*. An example is shown in Fig. 5. 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: both the `other` and `this` values are of type `Other`, with the `Other` implementing `Comparable<Other>`, so the call `other.compareTo(this)` is perfectly legal.

2) Scala Traits: Similarly to advanced interfaces in Java 8, Ceylon, and Kotlin, Scala traits [6, 20] support *default method implementations*. They can also have *abstract type* members, which, in particular, can be used as *associated types* [21]. 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 [20], will be mentioned later in Sec. IV-A with respect to the Concept design pattern.

3) Rust Traits: The Rust language [22] 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

```
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. 6. `Point` struct and its methods in Rust

example is shown in Fig. 6[2]: two `impl Point` blocks define method implementations for the `Point` struct. If a function takes the `&self`[3] argument (as `moveOn`), it 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.

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

Fig. 7. An example of using Rust traits

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, the *self type* of the trait is directly available and can be used in method definitions. Fig. 7[4] demonstrates an example: the `Eqtbl` trait defining the equality and inequality operations. Note how support for the 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 already available.

Method implementations in Rust can be probably thought of similarly to .NET "extension methods". But in contrast to .NET[5], types in Rust also can *retroactively implement traits* in `impl` blocks as shown in Fig. 7: `Eqtbl` 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 : Eqtbl...` is a shorthand, it can be declared in a `where` section as well.

There is no struct inheritance and subtype polymorphism in Rust. Nevertheless, as long as traits can be used not only as constraints but also as types, a dynamic dispatch is provided through a feature called trait objects. Suppose `i32` and `f64` implement the `Printable` trait from Fig. 7. Then the following code demonstrates creating and use of a polymorphic collection (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];
for v in polyVec { v.print(); }
```

```
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. 8. Protocols and their use in Swift

4) 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 the *self type*; due to the mechanism of extensions, types can *retroactively* adopt protocols. Fig. 8 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]Some details were omitted for simplicity. To make the code correct, one has to add `#[derive(Debug,Copy,Clone)]` before the `Point` definition.

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

[4]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`.

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

B. Languages with "Constraints-are-Not-Types" Philosophy

Most of the languages in this category were to some extent inspired by the design of Haskell type classes [23]. For defining constraints these languages suggest *new language constructs*, which are usually second-class citizens[6]. These constructs have *no self types* and *cannot* be used as types, they describe requirements on type parameters in an external way; therefore, retroactive satisfaction of constraints (*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 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 one is used as a type whereas the latter one is used as a constraint.

1) JavaGI Generalized Interfaces: JavaGI [7] generalized interfaces represent a kind of confluence of both "constraints-are-types" and "constraints-are-*not*-types" philosophies. Interfaces such as `PrettyPrintable` defined below are called single-parameter interfaces. They describe interfaces of a single type and can be used both as types and constraints.

```
interface PrettyPrintable { String prettyPrint(); }
```

Such interfaces have explicit access to the *self type* named `This`; an example is shown in Fig. 9, where the self type is used in the interface `EQ`. There is no direct support for default method implementations in JavaGI, but *abstract implementation definitions* can be used for this purpose[7]. For example, the `notEq` method of `EQ` (Fig. 9) is implemented in such a way. Generalized interfaces can be implemented *retroactively* in `implementation` blocks. They do not support associated types but can be generic; moreover, implementations can be generic as well, and the support for *type-conditional interface implementation* is provided:

```
implementation<S, T> EQ [Pair<S, T>] where S implements EQ
    where T implements EQ { ... }
```

Besides single-parameter interfaces, there are *multi-headed* generalized interfaces that adopt several features from Haskell type classes [24] and describe interfaces of several types. There is no self type in a multi-headed interface; therefore, it cannot be used as a type, it is designed to be used as a constraint *only*. An example of multi-headed interface is shown in Fig. 9: the `UNIFY` interface contains all the functions required by the unification algorithm considered earlier; 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. 3.

[6]Second-class citizens cannot be assigned to variables, passed as arguments, returned from functions.

[7]The design of JavaGI we discuss here goes back to 2011 when default method implementations were not supported in Java. With Java 8 this task could probably be solved in a more elegant way.

```
interface EQ { boolean eq(This that);
               boolean notEq(This that); }
abstract implementation EQ [EQ] {
  boolean notEq(This that) { return !this.eq(that); }}

boolean contains<X>(List<X> list, X x)
  where X implements EQ { ... }

abstract class Expr {...}   class IntLit extends Expr {...}
class PlusExpr extends Expr { Expr left; Expr right;  ... }
...
implementation EQ [Expr] {
  boolean eq(Expr that) { return false; }}
implementation EQ [PlusExpr]{boolean eq(PlusExpr that){...}}

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

```
concept InputIterator<Iter> { type value; ... }
concept Monoid<T> { fun identity_elt()  -> T;
                    fun binary_op(T, T) -> T; };

model Monoid<int>
{ fun identity_elt() -> int@ { return 0; } ... };

fun accumulate<Iter> where { InputIterator<Iter>,
                             Monoid<InputIterator<Iter>.value> }
(Iter first, Iter last) -> InputIterator<Iter>.value
{ let init = identity_elt(); ... }
```

Fig. 10. Concepts and their use in G

2) Language G and C++ concepts: Concept as an explicit language construct for defining constraints on type parameters was initially introduced in 2003 [25]. Several designs have been developed since that time [26–28]; in the large, the expressive power of concepts is rather close the Haskell type classes [4]. Concepts were designed to solve the problems of unconstrained C++ templates [14, 29]; they were expected to be included in C++0x standard, but this did not happen. A new version of concepts, Concepts Lite (C++1z) [30], is under way now. The language G declared as "a language for generic programming" [8] also provides concepts that are very similar to the C++0x concepts. G is a subset of C++ extended with several constructs for generic programming. For "C++ concepts" we use the G syntax in this paper.

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.

```
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. 11. 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. 12. Constraints and models in Genus

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*. Fig. 10 illustrates some examples: the `InputIterator<Iter>` concept with the associated type of elements `value`; the `Monoid<T>` concept and its model for the type `int`; the `accumulate<Iter>` generic function with two constraints, on the type of the iterator and on the associated type of this iterator. Note how `identity_elt` is called in `accumulate`: in contrast to the languages from the previous section, `identity_elt` is available in the body of `accumulate` at the top-level; this may lead to some inconvenience even if the autocomplete feature is supported in IDE.

3) C# with concepts: In the C#[cpt] project [9] (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#[cpt] allows *multiple models* of a concept in the same scope. Some examples are shown in Fig. 11: the `CEquatable[T]` concept with the `Equal` signature and default implementation of `NotEqual`, the generic interface `ISet<T>` with the concept-requirement on the type 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[8]: 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]`. This property is called "constraints-compatibility" in [9], but we will refer to it as "models-consistency". One more interesting thing about C#[cpt]: concept-requirements can be named. In the `Contains<T>` function (Fig. 11) 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.

4) Constraints in Genus: Like G concepts and Haskell type classes, constraints in Genus [10] (an extension for Java) are used as constraints only. Fig. 12 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-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 type `E` of the elements satisfies the `Eq[E]` constraint. This feature is called *model genericity*.

Just as C#[cpt], Genus supports *multiple models* and automatic generation of the *natural* model, which is the same thing as the default model in C#[cpt]. Due to this, the following code causes a static type error (we saw the same example in C#[cpt]):

```
Set[String] s1 = ...;
Set[String with CIEq] s2 = ...;
s1 = s2; // Static ERROR, s1 and s2 have different types
```

In Genus this feature is called *model-dependent types*. An important note is to be made here: in contrast to true dependent types that depend on *values*, model-dependent types depend on models, which are compile-time artefacts. So the model-dependent types are just as dependent as generic types are type-dependent types.

As well as concept-requirements in C#[cpt], constraint-requirements in Genus can be named; the example is shown in Fig. 12: `g` is a name of the `GraphLike[V,E]` constraint required by the `DualGraph[V,E]` model. Because function signatures inside constraints are declared with an explicit receiver type (in a style close to JavaGI), such as the type `T` in the `Eq[T]` constraint, syntax of calls to functions in the case of named models is `_.(g.sink)()`, not `g.sink(_)`.

C. Which Philosophy Is Better If Any?

It is time to find out which approach is better. Taking into consideration what we explored in Sec. III-A and Sec. III-B, we draw a conclusion that there are only two language features important for generic programming that cannot be incorporated in a language *together*:

[8]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.

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

Languages with "constraints-are-types" philosophy support the first feature but not the second, languages with "constraints-are-Not-types" philosophy vice versa[9]. Can we determine one feature that is more important?

It was shown in the study [31] 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 could be even rewritten and eliminated. According to [31], the same observation also holds for the code in Ceylon. It is hard to imagine any useful "constraint-and-type" example besides the `IPrintable` interface from Fig. 1. In those rare cases when this could happen, it is possible to provide a lightweight language mechanism for automatic generation of one construct from another. For example, single-parameter Genus constraints with some restrictions could be translated to Java interfaces, with the other direction being easier.

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 as we have seen in Sec. III-A. Language support for multiple models is also a problem in the latter approach: it is considered in detail in the next section. All other language facilities we discussed could be supported under any approach. Therefore, we claim that with respect to generic programming the *"constraints-are-Not-types" approach is preferable*. An additional benefit is that it eliminates the ambiguity in semantics of the interface-like constructs currently used for different purposes in OO languages.

IV. Single Model versus Multiple Models

For simplicity, in this part of the paper we call "constraint" any language construct that is used to describe constraints, while a way in which types satisfy the constraints we call "model". We have seen in the previous section that most of the languages allow having only one, unique model of a constraint for the given set of types; only C#$^{\text{cpt}}$ [9] and Genus [10] support multiple models[10]. And indeed this makes sense for the languages with "constraints-are-types" philosophy, 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 of a constraint for the same set of types. The example of string sets with case-sensitive and case-insensitive equality comparisons we saw earlier is only one of such examples; another one is the use of different orderings on numbers, yet different graph implementations, and so on. Thus, in respect of 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, and both of them have serious drawbacks.

1) Using the Adapter pattern. If one wants the type `Foo` to implement `IComparable<Foo>` in a different way, an adapter of `Foo`, the `Foo1` that implements `IComparable<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, a code becomes cumbersome.
2) Using the Concept design pattern [20], which is considered in Sec. IV-A.

As we have discovered in Sec. III-C, languages with the "constraints-are-types" philosophy are in the large less expressive than the ones with the "constraints-are-Not-types" philosophy. But may languages such as C#$^{\text{cpt}}$ and Genus, which are in the "constraints-are-Not-types" category and support multiple models at the language level, be considered as the best languages for generic programming? Or we can imagine a language with a better design? We discuss this question in Sec. IV-C. And one more question: if language support for multiple models is a good idea, then why does not Haskell [24] allow multiple instances of a type class? This issue is considered in Sec. IV-B.

A. Concept Pattern

The Concept design pattern is suitable for programming languages with the "constraints-are-types" philosophy. It eliminates two problems:

1) Firts, 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. 13 shows an example: in the case of the Concept pattern the F-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 interface of an object that can compare values of

```
// F-bounded polymorphism
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. 13. The use of the Concept design pattern in C#

[9]JavaGI seems to support both of them, but it actually provides different constructs for different purposes: single-parameter interfaces are more like Rust traits or Swift protocols, whereas multi-headed interfaces are similar to concepts and type classes; the latter cannot be used as types.

[10]G [8] allows multiple models only in different lexical scopes.

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 [6, 20], 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[11]. 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 constrained generic functions 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 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 comparers, so the result depends on which comparer was chosen to build the union. Recall that in C#[cpt] and Genus models are part of types; therefore, a similar situation causes the static type error. But in the case of the Concept pattern models-consistency *cannot* be checked at *compile time*.

B. Instance Uniqueness in Haskell

Type classes in Haskell [23] provide the support for ad hoc polymorphism (function overloading). Like concepts and constraints, they define functions available for some types. For instance, a type class for equality comparison is defined in Haskell as follows:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x /= y =  not (x == y)
```

It contains a function signature for the equality operator ==, and provides a default implementation for the inequality operator /=. Instances (models) of this type class can be *retroactively* defined for types. For example, an instance for Int, a *type-conditional* instance for lists, and so on.

```
instance Eq Int where ...          -- (==) implementation
instance Eq a => Eq [a] where ... -- (==) implementation
```

As long as type classes support ad hoc polymorphism, they are "globally transparent". If a function is a part of some type class, every time the name of this function is used, a compiler knows that an instance of the corresponding type class must be provided. Multiple instances of a type class for the same set of types are not allowed in Haskell, and there is a strong reason for that: *type inference*. Consider the following function definition:

```
foo xs ys = if xs == ys then xs else xs ++ ys
```

In Haskell such definition is valid and its type can be inferred. It is Eq a => [a] -> [a] -> [a][12]. Inference succeeds, because a compiler knows the following facts:

- as long as (++) has the type [a] -> [a] -> [a], xs and ys are lists;
- there is an instance of Eq for lists: Eq a => Eq [a].

If there were no Eq a => Eq [a] instance available, type checking would fail.

Suppose that multiple instances of a type class are allowed. What to do with type inference of the foo in this case? To check whether there is at least one instance Eq [a] in the scope? But probably not all Eq [a] instances require Eq a, should not the type of the foo be changed in this case to the type *Eq [a]* => [a] -> [a] -> [a]?

Now look at the following code:

```
class Eq a => Baz a where
  bar :: a -> Int

useBar xs ys = if length xs > length ys then bar xs - bar ys
                                        else bar ys - bar xs
```

If instances are uniquely defined, type checker just checks if there is an instance Eq [a] that implies Baz [a] (xs and ys are inferred to be lists because length has the type [a] -> Int). But if there are multiple Eq [a] instances, then every Baz [a] instance must specify which Eq [a] instance it uses. It can even be the case that there is a Baz [a] instance for one Eq [a], but not for another one. Therefore, at the point of the useBar *definition* a compiler has no idea whether there is an error of missed Baz [a] instance or not, because it knows nothing about the instance that might be used in a call to useBar. This information is available only at the point of the actual *call*, not the function definition.

Note that even with the OverlappingInstances extension for Haskell, multiple models in a sense we discuss in the paper are not supported. This extension indeed allows having in a scope several instances that match the constraints deduced for code. But there must be only *one*, the most specialised instance among them that compiler can select unambiguously (according to some rules) at the point of the code *definition*. Again, not at the call site — at the point of definition. Thus, a user of the code still cannot choose between instances, an instance is already selected by a compiler. Thus, Haskell sacrifices language support for multiple models for the sake of type inference. It is a strong argument for Haskell users, but in the case of the most object-oriented programming languages, which usually do not permit omitting type annotations of function arguments as well as constraints on type parameters, there is *no need to prohibit multiple models* in OO languages.

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

[12] [a] is a type of generic list, it is a notation for Data.List a

C. Parameters versus Predicates

So far we have found out that languages with "constraints-are-Not-types" philosophy may potentially provide better support for generic programming compared to other languages, especially if they also allow multiple models definition. We have seen only two languages with such properties, C#^{cpt} [9] and Genus [10], and there is an essential shortcoming in the design of both of them: constraints on type parameters are declared in "predicate-style" rather than "parameter-style". For example, consider the following Genus definition [10]:

```
Map[V,W] SSSP[V,E,W](V s)
where GraphLike[V,E], Weighted[E,W],
      OrdRing[W], Hashable[V] { ... }
```

`SSSP[V,E,W]` is a function for Dijkstras single-source shortest-path algorithm, with the `GraphLike[V,E]`, `Weighted[E,W]`, `OrdRing[W]` and `Hashable[V]` being constraints on type parameters. The constraints look as if they are predicates on types; and if they were predicates, this function would probably be well-designed. For example, 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#^{cpt} constraints *are not predicates*, they are actually *parameters*, as long as different models of a constraint can be used. In the worst case a call to the `SSSP[V,E,W]` function would be as follows:

```
...pathFromX = SSSP[MyVert, MyEdge, Double
                    with MyGrLike with MyEdgeDW
                    with DescDOR with MyVerHash](x);
```

Whereas in the best case:

```
...pathFromX = SSSP[MyVert, MyEdge, Double](x);
```

Note that edge and weight types cannot be deduced, because they are determined by the models of the constraints, not by the vertex `x` itself. It is easy to imagine that the models of edge weighing (`Weighted[E,W]`) and its ordered ring (`OrdRing[W]`) would often vary, so in many cases a call to `SSSP[V,E,W]` is likely to look like this:

```
...pathFromX = SSSP[MyVert, MyEdge, Double
                    with MyEdgeDW with DescDOR](x);
```

This is not very bad but is also not good enough.

If look again at the `SSSP` algorithm one could notice that it really depends on three things: a source vertex, a model of a weighed graph which this vertex belongs to, and a model of hashing. Furthermore, at the level of the `SSSP` signature the type `E` of edges does not matter, we are interested in the model of weighed graph as a whole. Taking into account this ideas, we can rewrite the `SSSP` in the following way:

```
constraint WeighedGraph[V,E,W]
  extends GraphLike[V,E], Weighted[E,W], OrdRing[W] {}

Map[V,W] SSSP[V,E,W](V s)
  where WeighedGraph[V,E,W], Hashable[V] { ... }
```

Then a call to `SSSP` also becomes better:

```
...pathFromX = SSSP[MyVert, MyEdge, Double with MyWGr](x);
```

```
module type Eq = sig
  type t
  val equal : t -> t -> bool
end

implicit module Eq_int = struct
  type t = int
  let equal x y = ...
end
implicit module Eq_list {E : Eq} = struct
  type t = Eq.t list
  let equal xs ys = ...
end

let foo  {EL : Eq} xs ys = if EL.equal(xs, ys)
                           then xs else xs @ ys
let foo' {E : Eq}  xs ys = if (Eq_list E).equal(xs, ys)
                           then xs else xs @ ys
let x = foo [1;2;3] [4;5]
let y = foo' [1;2;3] [4;5]
```

Fig. 14. OCaml modular implicits

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

concept Ordering[T] refines Equality[T]
{ int Compare(T x, T y); }}

interface ISet<T | Equality[T] eq> { ... }
interface ICollection<T> { ...
  bool Remove<Equality[T] eq>(T x); ... }

bool Contains<T | Equality[T] eq>(IEnumerable<T> vs, T x)
{... if (eq.Equal(...) ...}

int MaxInt<|Ordering[int] ord>(IEnumerable<int> vs) {...}
```

Fig. 15. The use of concept-parameters in Cp#

Nevertheless, we believe that in the case of multiple models the "predicate-style" syntax of constraints is misleading and makes it more difficult to write and call generic code. We suggest that the design of constraints has to be maintained in the "parameter-style". One example of such design is provided by the extension for the OCaml language — *modular implicits* [32]; it is briefly discussed in Sec. IV-C1. A sketch of the "parameter-style" design of constraints for object-oriented languages is presented in Sec. IV-C2.

1) Modular Implicits in OCaml: In the "modular implicits" extension for the OCaml language [32] module types are used to describe constraints, modules represent models, with generic functions explicitly taking *module-parameters*. Fig. 14 demonstrates some examples. By contrast to concepts and genus constraints, module types and modules do not have type parameters, instead they have type members, such as the `t` in the `Eq` module type. `Eq_int` and `Eq_list` are the models of `Eq` for the `int` and generic list. Generic functions that need constraints, such as `foo` and `foo'`, explicitly take the implicit module parameters `EL` and `E`. Notice that just as type parameters, `EL` and `E` are *compile-time* parameters, not run-time. They are called implicit because at a call to generic function actual models can be inferred, as in the `x` and `y` examples in Fig. 14. Note that in the `foo` function any model of comparison of lists is expected, whereas `foo'` expects a model of comparison of elements of lists and fixes the model `Eq_list E` for comparing lists.

	Haskell	C#	Java 8	Scala	Ceylon	Kotlin	Rust	Swift	JavaGI	G	C#[cpt]	Genus	ModImpl
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]	●[a]
Default method implementation	●	○	●	●	●	●	●	●	◐	●	●	○	○
Associated types	●	○	○	●	○	○	●	●	○	●	●	○	●
Constraints on associated types	◐	–	–	●	–	–	●	●	–	●	●	–	●
Same-type constraints	◐	–	–	●	–	–	●	●	–	●	●	–	●
Subtype constraints	–	●	●	●	●	●	–	●	○	○	●	○	–
Supertype constraints	–	○	○	●	○	○	–	○	○	○	●	○	–
Concept-based overloading	○	○	○	○	○	○	●	○	○	◐[d]	○	○	○
Multiple models	○	⚹	⚹	⚹	⚹	⚹	○	○	○	◐[b]	●	●	●
Models-consistency (model-dependent types)	–[c]	○	○	○	○	○	–[c]	–[c]	–[c]	–[c]	●	●	●
Model genericity	–	⚹	⚹	⚹	⚹	⚹	●	○	○	○	○	●	–

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

TABLE I
THE LEVELS OF SUPPORT FOR GENERIC PROGRAMMING IN OO LANGUAGES

2) Concept Parameters for C#: Fig. 15 shows some examples of generic code in the style of concept-parameters, which we call Cp# — C# with concept-Parameters. Concepts are the same as in C#[cpt], whereas constraints on type parameters are not predicates any more, they are explicitly stated as *parameters* in the angle brackets after the "|" sign. In the `ICollection<T>` interface the `Remove` method is obviously generic: it takes the concept-parameter `eq` for comparing values of type `T`. Note that concept-parameters can even be non-generic as in the `MaxInt` function.

If default models are supported, it must be possible to infer concept-arguments just in the same way as in C# or Genus, so that in common cases instances of generic functions and classes can be written in a usual way, without the need to specify the models required:

```
var ints = new ISet<int>(...);
var has5 = Contains(ints, 5);

var maxv = MaxInt(ints);
var minv = MaxInt<|IntOrdDesc>(ints);

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

C#[cpt] and Genus can easily be redesigned to follow the "concept-parameters" style presented here. With this style, the syntax of such languages would perfectly fit the semantics. On the other hand, the "concept-predicates" style used misleads a programmer and masks the fact that constraints can be satisfied non-uniquely.

V. CONCLUSION AND FUTURE WORK

Table I provides a summary on comparison of the languages: 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 the 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. The "ModImpl" column corresponds to the OCaml modular implicits. All the properties that appear in rows of Table I were discussed in Sec. III and Sec. IV. Related properties are grouped within horizontal lines; some of them are mutually exclusive. For example, as we saw earlier, the use of constraints as types and natural language support for multi-type constraints are mutually exclusive features. The major features analysed in the paper are highlighted in bold.

The purpose of this table is not to determine the best language. The purpose 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. It is also easy to see that there are features that can be expressed under any approach, such as static methods, default method implementations, associated types [15], and even type-conditional models.

It should be mentioned that the table is not exhaustive. There is a bunch of facilities that we did not discuss at all, although they can be considered independently of the study we made. Thus, for example, Genus [10] provides a support for such useful feature as *multiple dynamic dispatch*. Consider

the following code:

```
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) {...}
  Shape Triangle.intersect(Circle c) {...} ... }
```

It provides a subtype polymorphism on multiple arguments. So that in the call `s1.intersect(s2)` the most specific version of `intersect` would be used depending on the *dynamic* types of both `s1` and `s2`.

Another interesting feature is *concept variance*. For example, suppose we have the following Cp# definitions:

```
interface ISet<T | Equality[T] eq> { ... }
class B { ... }
class D : B { ... }
model EqB for Equality[B] { ... }
```

Should it be the case that `ISet<D, EqB>` is a legal instance? Under what conditions? It is also desirable to have the class `SortedSet<T | Ordering[T] ord>` implementing the interface `ISet<T|ord>`. Are there any problems here?

Now recall the `ICollection<T>` interface definition:

```
interface ICollection<T> { ...
  bool Remove<Equality[T] eq>(T x); ... }
```

The `SortedSet<T|ord>` class obviously implements the interface `ICollection<T>`. Should it be the case that the `ord` model of `Equality[T]` be used in place of `eq` in the `Remove` method? Or the `Remove` method has to remain model-generic?

And one more question. Consider the following function:

```
void foo<T | Equality[T] eq>(ISet<T|eq> s) { ... }
...
ISet<string | EqStringCaseS> s1 =
    new SortedSet<string | OrdStringCSAsc>(...);
foo(s1);
```

Which model of `Equality[string]` should be used inside the `foo<>`? The static `EqStringCaseS` or the dynamic `OrdStringCSAsc` one?

There are other questions similar to mentioned above that relate constraints on type parameters to usual features of object-oriented programming. Some of these questions require a careful type-theoretical investigation, so this is the subject for future work.

Acknowledgment

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

References

[1] Belyakova J. Language Support for Generic Programming in Object-Oriented Languages: Peculiarities, Drawbacks, Ways of Improvement, *Lecture Notes in Computer Science*, 2016, to appear.

[2] Musser D. R. and Stepanov A. A. Generic Programming, *Proceedings of the International Symposium ISSAC'88 on Symbolic and Algebraic Computation*, ISAAC '88, London, UK, UK: Springer-Verlag, 1989, pp. 13–25.

[3] Garcia R. et al. An Extended Comparative Study of Language Support for Generic Programming, *J. Funct. Program.*, Mar. 2007, vol. 17, no. 2, pp. 145–205.

[4] Bernardy J.-P. et al. A Comparison of C++ Concepts and Haskell Type Classes, *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, WGP '08, Victoria, BC, Canada: ACM, 2008, pp. 37–48.

[5] Garcia R. et al. A Comparative Study of Language Support for Generic Programming, *SIGPLAN Not.*, Oct. 2003, vol. 38, no. 11, pp. 115–134.

[6] Oliveira B. c. d. s. and Gibbons J. Scala for Generic Programmers: Comparing Haskell and Scala Support for Generic Programming, *J. Funct. Program.*, July 2010, vol. 20, no. 3-4, pp. 303–352.

[7] Wehr S. and Thiemann P. JavaGI: The Interaction of Type Classes with Interfaces and Inheritance, *ACM Trans. Program. Lang. Syst.*, July 2011, vol. 33, no. 4, 12:1–12:83.

[8] Siek J. G. and Lumsdaine A. A Language for Generic Programming in the Large, *Sci. Comput. Program.*, May 2011, vol. 76, no. 5, pp. 423–465.

[9] Belyakova J. and Mikhalkovich S. Pitfalls of C# Generics and Their Solution Using Concepts, *Proceedings of the Institute for System Programming*, June 2015, vol. 27, no. 3, pp. 29–45.

[10] Zhang Y. et al. Lightweight, Flexible Object-oriented Generics, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, Portland, OR, USA: ACM, 2015, pp. 436–445.

[11] Canning P. et al. F-bounded Polymorphism for Object-oriented Programming, *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, Imperial College, London, United Kingdom: ACM, 1989, pp. 273–280.

[12] Bruce K. et al. On Binary Methods, *Theor. Pract. Object Syst.*, Dec. 1995, vol. 1, no. 3, pp. 221–242.

[13] Kennedy A. and Syme D. Design and Implementation of Generics for the .NET Common Language Runtime, *SIGPLAN Not.*, May 2001, vol. 36, no. 5, pp. 1–12.

[14] Belyakova J. and 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*, 2015, no. 2, 63–77 (in Russian).

[15] Järvi J., Willcock J., and Lumsdaine A. Associated Types and Constraint Propagation for Mainstream Object-oriented Generics, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, San Diego, CA, USA: ACM, 2005, pp. 1–19.

[16] Martelli A. and Montanari U. An Efficient Unification Algorithm, *ACM Trans. Program. Lang. Syst.*, Apr. 1982, vol. 4, no. 2, pp. 258–282.

[17] *The Ceylon Language Specification, version 1.2.2 (March 11, 2016).*

[18] *The Kotlin Reference, version 1.0 (February 11, 2016).*

[19] *Java Platform, Standard Edition (Java SE) 8.*

[20] Oliveira B. C., Moors A., and Odersky M. Type Classes As Objects and Implicits, *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 341–360.

[21] Pelenitsyn A. Associated Types and Constraint Propagation for Generic Programming in Scala, English, *Programming and Computer Software*, 2015, vol. 41, no. 4, pp. 224–230.

[22] *The Rust Reference, version 1.7.0 (March 3, 2016).*

[23] Hall C. V. et al. Type Classes in Haskell, *ACM Trans. Program. Lang. Syst.*, Mar. 1996, vol. 18, no. 2, pp. 109–138.

[24] Wadler P. and Blott S. How to Make Ad-hoc Polymorphism Less Ad Hoc, *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, Austin, Texas, USA: ACM, 1989, pp. 60–76.

[25] 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, 2003.

[26] Stroustrup B. and Dos Reis G. *Concepts — Design Choices for Template Argument Checking*, Technical Report N1522=03-0105, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, 2003.

[27] Dos Reis G. and Stroustrup B. Specifying C++ Concepts, *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, Charleston, South Carolina, USA: ACM, 2006, pp. 295–308.

[28] Stroustrup B. and Sutton A. *A Concept Design for the STL*, Technical Report N3351=12-0041, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, 2012.

[29] Stepanov A. A. and Lee M. *The Standard Template Library*, Technical Report 95-11(R.1), HP Laboratories, 1995.

[30] Sutton A. *C++ Extensions for Concepts PDTS*, Technical Specification N4377, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, 2015.

[31] Greenman B., Muehlboeck F., and Tate R. Getting F-bounded Polymorphism into Shape, *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, Edinburgh, United Kingdom: ACM, 2014, pp. 89–99.

[32] White L., Bour F., and Yallop J. Modular Implicits, *ArXiv e-prints*, Dec. 2015, arXiv: 1512.01895 [cs.PL].