

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

Julia Belyakova
julbel@sfnedu.ru

I. I. Vorovich Institute for Mathematics, Mechanics and Computer Science
Southern Federal University
Rostov-on-Don, Russia

September 23rd 2016

XX Brazilian Symposium on Programming Languages
SBLP 2016
#CBSOft

Contents

- 1 Generic Programming
- 2 Language Support for GP in Object-Oriented Languages
- 3 Language Extensions for Generic Programming
- 4 Conclusion

Generic Programming

A term “Generic Programming” (GP) was coined in 1989 by Alexander Stepanov and David Musser [1].

Idea

Code is written in terms of **abstract** types and operations (parametric polymorphism).

Purpose

Writing highly reusable code.

An Example of Unconstrained Generic Code (C#)

```
static int Count<T>(T[] vs, Predicate<T> p)
{
    int cnt = 0;
    foreach (var v in vs)
        if (p(v)) ++cnt;
    return cnt;
}
```

Figure: Calculating amount of elements in vs that satisfy the predicate p

An Example of Unconstrained Generic Code (C#)

```
static int Count<T>(T[] vs, Predicate<T> p)
{
    // p : T -> Bool
    int cnt = 0;
    foreach (var v in vs)
        if (p(v)) ++cnt;
    return cnt;
}
```

Figure: Calculating amount of elements in vs that satisfy the predicate p

Count<T> can be instantiated with **any** type!

```
int[] ints = new int[]{ 3, 2, -8, 61, 12 };
var evCnt = Count(ints, x => x % 2 == 0); // 3

string[] strs = new string[]{ "hi", "bye", "hello", "stop" };
var evLenCnt = Count(strs, x => x.Length % 2 == 0); // 2
```

We Need More Genericity!

Look again at the vs parameter:

```
static int Count<T>(T[] vs, Predicate<T> p)
{ ... }
```

```
int[] ints = ...
var evCnt = Count(ints, ...
```

```
string[] strs = ...
var evLenCnt = Count(strs, ...
```

We Need More Genericity!

Look again at the vs parameter:

```
static int Count<T>(T[] vs, Predicate<T> p)
{ ... }
```

```
int[] ints = ...
var evCnt = Count(ints, ...
```

```
string[] strs = ...
var evLenCnt = Count(strs, ...
```

The Problem

Generic Count<T> function is not generic enough.
It works with **arrays only!**

True C# Code for the Count Function

```
interface IEnumerable<T> : IEnumerable
{
    IEnumerator<T> GetEnumerator(); ...
}
```

Figure: IEnumerable<T> interface

```
static int Count<T>(IEnumerable<T> vs, Predicate<T> p)
{
    // p : T -> Bool
    int cnt = 0;
    foreach (var v in vs) ...
}
```

Figure: Calculating amount of elements in vs that satisfy the predicate p

```
var ints = new int[]{ 3, 2, -8, 61, 12 };
var evCnt = Count(ints, x => x % 2 == 0); // array

var intSet = new SortedSet<int>{ 3, 2, -8, 61, 12 };
var evSCnt = Count(intSet, x => x % 2 == 0); // set
```


When Constraints are Needed

How to write a **generic** function that finds maximum element in a collection?

When Constraints are Needed

How to write a **generic** function that finds maximum element in a collection?

```
static T FindMax<T>(IEnumerable<T> vs)
{
    T mx = vs.First();
    foreach (var v in vs)
        if (mx < v)           // ERROR: operator '<'
            mx = v;         // is not provided for the type T
    ...
}
```

When Constraints are Needed

How to write a **generic** function that finds maximum element in a collection?

```
static T FindMax<T>(IEnumerable<T> vs)
{
    T mx = vs.First();
    foreach (var v in vs)
        if (mx < v)           // ERROR: operator '<'
            mx = v;          // is not provided for the type T
    ...
}
```

To find maximum in `vs`, values of type `T` must **be comparable!**

“Being comparable” is a **constraint**.

An Example of Generic Code with Constraints (C#)

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

static T FindMax<T>(IEnumerable<T> vs)
    where T : IComparable<T>           // F-bounded polymorphism
{
    T mx = vs.First();
    foreach (var v in vs)
        if (mx.CompareTo(v) < 0) mx = v;
    return mx;
}
```

Figure: Searching for maximum element in vs

An Example of Generic Code with Constraints (C#)

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

static T FindMax<T>(IEnumerable<T> vs)
    where T : IComparable<T>           // F-bounded polymorphism
{
    T mx = vs.First();
    foreach (var v in vs)
        if (mx.CompareTo(v) < 0) mx = v;
    return mx;
}
```

Figure: Searching for maximum element in vs

FindMax<T> can **only** be instantiated with types implementing the IComparable<T> interface.

```
var ints = new int[]{ 3, 2, -8, 61, 12 };
var iMax = FindMax(ints); // 61
var strs = new LinkedList<string>{ "hi", "bye", "stop", "hello" };
var sMax = FindMax(strs); // "stop"
```

Explicit Constraints on Type Parameters

Programming languages provide various language mechanisms for generic programming based on **explicit constraints**:

- Haskell: type classes;
- SML, OCaml: modules;
- Rust, Scala: traits;
- Swift: protocols;
- Ceylon, Kotlin, C#, Java: interfaces;
- etc.

C++

C++ Templates are unconstrained!

Explicit Constraints on Type Parameters

Programming languages provide various language mechanisms for generic programming based on **explicit constraints**:

- Haskell: type classes;
- SML, OCaml: modules;
- Rust, Scala: traits;
- Swift: protocols;
- Ceylon, Kotlin, C#, Java: interfaces;
- etc.

C++

C++ Templates are unconstrained!

It was shown in earlier studies that C# and Java yield to many languages with respect to language support for GP [2–4].

Motivation for the Study

Poor Language Support for Generic Programming

Is it a problem of C# and Java only?

Or is it a **typical** problem of **object-oriented** languages?

Motivation for the Study

Poor Language Support for Generic Programming

Is it a problem of C# and Java only?

Or is it a **typical** problem of **object-oriented** languages?

To answer the question, let's look at the modern object-oriented languages [name (first appeared, recent stable release)]:

- Scala (2004, 2016);
- Rust (2010, 2016);
- Ceylon (2011, 2016);
- Kotlin (2011, 2016);
- Swift (2014, 2016).

Constraints as Types

All the OO languages explored follow the *same* approach to constraining type parameters.

The “Constraints-are-Types” Approach

Interface-like language constructs are used in code in two different roles:

- 1 as **types** in object-oriented code;
- 2 as **constraints** in generic code.

Recall the example of C# generic code with constraints:

```
interface IEnumerable<T> { ... }  
interface IComparable<T> { ... }
```

```
static T FindMax<T>(IEnumerable<T> vs) where T : IComparable<T>
```

Inevitable Limitations of the OO approach

An interface/trait/protocol describes properties of a **single** type that implements/extends/adopts it. Therefore:

Inevitable Limitations of the OO approach

An interface/trait/protocol describes properties of a **single** type that implements/extends/adopts it. Therefore:

- **Multi-type constraints** cannot be expressed naturally.
Instead of

```
double Foo<A, B>(A[] xs) where <single constraint on A, B>  
// the constraint includes functions like B[] Bar(A a)
```

Inevitable Limitations of the OO approach

An interface/trait/protocol describes properties of a **single** type that implements/extends/adopts it. Therefore:

- **Multi-type constraints** cannot be expressed naturally.
Instead of

```
double Foo<A, B>(A[] xs) where <single constraint on A, B>
// the constraint includes functions like B[] Bar(A a)
```

we have:

```
interface IConstraintA<A, B> where A : IConstraintA<A, B>
where B : IConstraintB<A, B> {...}
interface IConstraintB<A, B> where A : IConstraintA<A, B>
where B : IConstraintB<A, B> {...}
double Foo<A, B>(A[] xs)
  where A : IConstraintA<A, B>
  where B : IConstraintB<A, B> {...}
```

Inevitable Limitations of the OO approach

An interface/trait/protocol describes properties of a **single** type that implements/extends/adopts it. Therefore:

- **Multi-type constraints** cannot be expressed naturally.
Instead of

```
double Foo<A, B>(A[] xs) where <single constraint on A, B>
// the constraint includes functions like B[] Bar(A a)
```

we have:

```
interface IConstraintA<A, B> where A : IConstraintA<A, B>
                                where B : IConstraintB<A, B> {...}
interface IConstraintB<A, B> where A : IConstraintA<A, B>
                                where B : IConstraintB<A, B> {...}

double Foo<A, B>(A[] xs)
    where A : IConstraintA<A, B>
    where B : IConstraintB<A, B> {...}
```

- **Multiple models** cannot be supported at language level.

Concept Pattern

With the Concept design pattern¹ [5] ("Type Classes As Objects and Implicits" by Oliveira et. al., 2010), constraints on type parameters are replaced with extra arguments – "**concepts**".

F-Bounded Polymorphism

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

static T FindMax<T>(
  IEnumerable<T> vs)
  where T : IComparable<T> // *
{
  T mx = vs.First();
  foreach (var v in vs)
    if (mx.CompareTo(v) < 0) // *
      ...
}
```

Concept Pattern

```
interface IComparer<T>
{ int Compare(T x, T y); } // *

static T FindMax<T>(
  IEnumerable<T> vs,
  IComparer<T> cmp) // *
{
  T mx = vs.First();
  foreach (var v in vs)
    if (cmp.Compare(mx,v) < 0) // *
      ...
}
```

¹Concept pattern \approx Strategy design pattern?

Advantages of the Concept Pattern

Both limitations of the “Constraints-are-Types” approach are eliminated with this design pattern!

- 1 multi-type constraints are multi-type “concept” arguments;

```
interface IConstraintAB<A, B>  
{ B[] Bar(A a); ... }
```

```
double Foo<A, B>(A[] xs, IConstraintAB<A, B> c)  
{ ... c.Bar(...) ... }
```

- 2 multiple “models” are allowed as long as several classes can implement the same interface.

```
class IntCmpDesc : IComparer<int> { ... }  
class IntCmpMod42 : IComparer<int> { ... }
```

```
var ints = new int[]{ 3, 2, -8, 61, 12 };
```

```
var minInt = FindMax(ints, new IntCmpDesc());  
var maxMod42 = FindMax(ints, new IntCmpMod42());
```


Drawbacks of the Concept Pattern I

The Concept design pattern is **widely used** in standard generic libraries of C#, Java, and Scala, but it has several **problems**.

Possible runtime overhead

Extra class fields or function arguments.

```
interface IComparer<T>
{ ... }

class SortedSet<T> : ...
{
    IComparer<T> Comparer;
    ...
}
```



Drawbacks of the Concept Pattern II

The Concept design pattern is **widely used** in standard generic libraries of C#, Java, and Scala, but it has several **problems**.

Models inconsistency

Objects of the same type can have different models (at runtime).

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

Attention!

GetUnion(s1, s2) could differ from GetUnion(s2, s1)!

Alternative Approach

There are several language extensions for generic programming inspired by Haskell type classes [6]:

- C++ concepts [7, 8] (2003–2014) and concepts in language G [9] (2005–2011);
- Generalized interfaces in JavaGI [10] (2007–2011);
- Concepts for C# [3] (2015);
- Constraints in Java Genus [11] (2015).

All these extensions follow the *alternative* approach to constraining type parameters.

The “Constraints-are-Not-Types” Approach

To **constrain** type parameters, a **separate** language construct is used. It cannot be used as type.

Constraints with Java Genus

```
interface Iterable[T] { ... }

constraint Eq[T] { boolean T.equals(T other); }
constraint Comparable[T] extends Eq[T] { int T.compareTo(T other); }

static T FindMax[T](Iterable[T] vs) where Comparable[T]
{ ...
    if (mx.compareTo(v) < 0) ... }
```

Figure: Searching for maximum element in vs

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

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

Set[String] s1 = ...; // case-sensitive natural model is used by default
Set[String with StringCIEq] s2 = ...;
s1 = s2; // Static ERROR, s1 and s2 have different types
```

Figure: Models Consistency

Which Approach is Better?

“Constraints-are-Types”

Lack of language support for **multi-type constraints** and **multiple models**, with the Concept design pattern having its own drawbacks.

Constraints can be used as **types**.

“Constraints-are-Not-Types”

Language support for **multi-type constraints** and **multiple models**.

Constraints cannot be used as **types**.

“Constraints-are-Not-Types” Is Preferable

There are at least 3 reasons for this assertion:

“Constraints-are-Not-Types” Is Preferable

There are at least 3 reasons for this assertion:

- According to [12] (the “material-shape separation”), in practice interfaces that are used as **constraints** (such as `IComparable<T>`) are **almost never used as types**.

“Constraints-are-Not-Types” Is Preferable

There are at least 3 reasons for this assertion:

- According to [12] (the “material-shape separation”), in practice interfaces that are used as **constraints** (such as `IComparable<T>`) are **almost never used as types**.
- By contrast, **multi-type constraints** and **multiple models** are often **desirable** for generic programming.

“Constraints-are-Not-Types” Is Preferable

There are at least 3 reasons for this assertion:

- According to [12] (the “material-shape separation”), in practice interfaces that are used as **constraints** (such as `IComparable<T>`) are **almost never used as types**.
- By contrast, **multi-type constraints** and **multiple models** are often **desirable** for generic programming.
- As for the other features important for generic programming, they can be supported using any approach.

“Constraints-are-Not-Types” Is Preferable

There are at least 3 reasons for this assertion:

- According to [12] (the “material-shape separation”), in practice interfaces that are used as **constraints** (such as `Comparable<T>`) are **almost never used as types**.
- By contrast, **multi-type constraints** and **multiple models** are often **desirable** for generic programming.
- As for the other features important for generic programming, they can be supported using any approach.

Research Problem

How to combine the approach with **subtype constraints** on types.

Concept Parameters versus Concept Predicates

When multiple models are supported, constraints on type parameters are *not predicates* any more, they are **compile-time parameters** [13] (just as types are parameters of generic code).

Concept Predicates

```
interface List[T] { ...  
  boolean remove(T x) where Eq[T];  
}  
List[int] xs = ...  
xs.remove[with StringCIEq](5);  
  
interface Set[T where Eq[T]] {...}  
Set[String] s1 = ...;  
Set[String with StringCIEq] s2=...;
```

Concept Parameters

```
interface List<T> { ...  
  boolean remove<! Eq[T] eq>(T x);  
}  
List<int> xs = ...  
xs.remove<StringCIEq>(5);  
  
interface Set<T ! Eq[T] eq> {...}  
Set<String> s1 = ...;  
Set<String ! StringCIEq> s2 = ...;
```

References I



D. Musser and A. Stepanov. “Generic programming”. English. In: *Symbolic and Algebraic Computation*. Ed. by P. Gianni. Vol. 358. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1989, pp. 13–25.



R. Garcia et al. “An Extended Comparative Study of Language Support for Generic Programming”. In: *J. Funct. Program.* 17.2 (Mar. 2007), pp. 145–205.



J. Belyakova and S. Mikhalkovich. “Pitfalls of C# Generics and Their Solution Using Concepts”. In: *Proceedings of the Institute for System Programming* 27.3 (June 2015), pp. 29–45.



J. Belyakova and S. Mikhalkovich. “A Support for Generic Programming in the Modern Object-Oriented Languages. Part 1. An Analysis of the Problems”. In: *Transactions of Scientific School of I.B. Simonenko. Issue 2 2* (2015), 63–77 (in Russian).

References II



B. C. Oliveira, A. Moors, and M. Odersky. “Type Classes As Objects and Implicits”. In: *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.



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



G. Dos Reis and B. Stroustrup. “Specifying C++ Concepts”. In: *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.

References III



D. Gregor et al. “Concepts: Linguistic Support for Generic Programming in C++”. In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: ACM, 2006, pp. 291–310.



J. G. Siek and A. Lumsdaine. “A Language for Generic Programming in the Large”. In: *Sci. Comput. Program.* 76.5 (May 2011), pp. 423–465.



S. Wehr and P. Thiemann. “JavaGI: The Interaction of Type Classes with Interfaces and Inheritance”. In: *ACM Trans. Program. Lang. Syst.* 33.4 (July 2011), 12:1–12:83.



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

References IV



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



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

Comparison of Languages and Extensions

Language Support for GP in OO Languages	Haskell	C#	Java 8	Scala	Ceylon	Kotlin	Rust	Swift	JavaGI	G	C#-cpt	Genus	Modimpl
Constraints can be used as types	○	●	●	●	●	●	●	●	◐	○	○	○	○
<i>Explicit self types</i>	—	○	○	◐	●	○	●	●	◐	—	—	—	—
Multi-type constraints	●	*	*	*	○	*	○	○	●	●	●	●	●
<i>Retroactive type extension</i>	—	●	○	○	○	●	●	●	○	○	○	○	—
<i>Retroactive modeling</i>	●	*	*	*	○	*	●	●	●	●	●	●	●
<i>Type conditional models</i>	●	○	○	○	○	○	●	○	●	●	●	●	●
<i>Static methods</i>	●	○	●	○	●	●	●	●	●	●	●	●	●
<i>Default method implementation</i>	●	○	●	●	●	●	●	●	◐	●	●	○	○
<i>Associated types</i>	●	○	○	●	○	○	●	●	○	●	●	○	●
<i>Constraints on associated types</i>	◐	—	—	●	—	—	●	●	—	●	●	—	●
<i>Same-type constraints</i>	◐	—	—	●	—	—	●	●	—	●	●	—	●
<i>Concept-based overloading</i>	○	○	○	○	○	○	●	○	○	◐	○	○	○
Multiple models	○	*	*	*	*	*	○	○	○	◐ ^a	●	●	●
Models consistency (model-dependent types)	— ^b	○	○	○	○	○	— ^b	— ^b	— ^b	— ^b	●	●	●
<i>Model genericity</i>	—	*	*	*	*	*	●	○	○	○	○	●	—

* means support via the Concept pattern. ^aG supports lexically-scoped models but not really multiple models.

^bIf multiple models are not supported, the notion of model-dependent types does not make sense.

Dependent Types

```

-- natural number
data Nat      -- Nat : Type
  = Zero      -- Zero : Nat
  | Succ Nat  -- Succ : Nat -> Nat

-- generic list
data List a   -- List : Type -> Type
  = []         -- [] : List a
  | (::) a (List a) -- (::) : a -> List a -> List a

-- vector of the length k (dependent type)
data Vect : Nat -> Type -> Type where
  Nil  : Vect Zero a
  Cons : a -> Vect k a -> Vect (Succ k) a

```

Figure: Data types and dependent types in Idris

Dependent Types

```

-- natural number
data Nat      -- Nat : Type
  = Zero      -- Zero : Nat
  | Succ Nat  -- Succ : Nat -> Nat

-- generic list
data List a   -- List : Type -> Type
  = []         -- [] : List a
  | (::) a (List a) -- (::) : a -> List a -> List a

-- vector of the length k (dependent type)
data Vect : Nat -> Type -> Type where
  Nil  : Vect Zero a
  Cons : a -> Vect k a -> Vect (Succ k) a

```

Figure: Data types and [dependent types](#) in Idris

If we had dependent types in OO languages, we would also have [models consistency](#) (a compaper could be a part of the type).

Constrained Generic Code in Scala

Traits are used in Scala instead of interfaces.

```
trait Iterable[A] {  
  def iterator: Iterator[A]  
  def foreach ...  
}  
  
trait Ordered[A] {  
  abstract def compare (that: A): Int  
  def < (that: A): Boolean ...  
}
```

Figure: `Iterable[A]` and `Ordered[A]` traits (Scala)

Constrained Generic Code in Scala

Traits are used in Scala instead of interfaces.

```
trait Iterable[A] {  
  def iterator: Iterator[A]  
  def foreach ...  
}  
  
trait Ordered[A] {  
  abstract def compare (that: A): Int  
  def < (that: A): Boolean ...  
}
```

Figure: `Iterable[A]` and `Ordered[A]` traits (Scala)

```
def findMax[A <: Ordered[A]] (vs: Iterable[A]): A {  
  ...  
  if (mx < v) ...  
}
```

Figure: Extract from the `findMax[A]` function

Concept Pattern in Scala

In Scala it has a special support: **context bounds** and **implicit**s.

F-Bounded Polymorphism

```

trait Ordered[A] {
  abstract def compare
    (that: A): Int
  def < (that: A): Boolean = ...
}

// upper bound
def findMax[A <: Ordered[A]]
  (vs: Iterable[A]): A
{ ... }

```

Concept Pattern

```

trait Ordering[A] {
  abstract def compare
    (x: A, y: A): Int
  def lt(x: A, y: A): Boolean = ...
}

// context bound (syntactic sugar)
def findMax[A : Ordering]
  (vs: Iterable[A]): A
{ ... }

// implicit argument (real code)
def findMax(vs: Iterable[A])
  (implicit ord: Ordering[A])
{ ... }

```