Concept Parameters as a New Mechanism of Generic Programming for C# Language

Julia Belyakova
julbel@sfedu.ru

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

July 17th 2016

Doctoral Symposium
The 30th European Conference on Object-Oriented Programming (2016)
ECOOP DS 2016
1. Generic Programming
   - Constraints on Type Parameters

2. Research Problem

3. Concept Parameters
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.
How to write a **generic** function that finds maximum element in a generic collection?

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

static T FindMax<T>(IEnumerable<T> vs)  // could be ..(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.
Constrained Generic Code

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

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

static T FindMax<T>(IEnumerable<T> vs) // could be ..(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#)

```csharp
interface IEnumerable<T> : IEnumerable { ... }  
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#)

```csharp
interface IEnumerable<T> : IEnumerable { ... }
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 };  // 61
var iMax = FindMax(ints);
var strs = new LinkedList<string>{ "hi", "bye", "stop", "hello" };  // "stop"
var sMax = FindMax(strs);
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.

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

1. Generic Programming
2. Research Problem
3. Concept Parameters
The Goal of the Research

To develop a mechanism of **generic programming** that improves language support for GP in mainstream **object-oriented** languages.
Motivation

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?

Constraints—are—Types
All of them follow the same approach to constraining type parameters [5]: OO constructs used as types (such as interfaces) are also used as constraints.
Motivation

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?

What about modern object-oriented languages?
[name (first appeared, recent stable release)]

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

Constraints-are-Types

All of them follow the same approach to constraining type parameters [5]: OO constructs used as types (such as interfaces) are also used as constraints.
Inevitable Limitations of the OO approach
(are usually “solved” with the Concept design pattern)

An interface/trait/protocol describes properties of a *single* type that implements/extends/adopts it. Therefore:
Inevitable Limitations of the OO approach
(are usually “solved” with the Concept design pattern)

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

```csharp
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 (are usually “solved” with the Concept design pattern)

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

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

we have:

```csharp
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
(are usually “solved” with the Concept design pattern)

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>
  interface IConstraintB<A, B> where A : IConstraintA<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.
There are several language extensions for generic programming influenced by Haskell type classes [6]:

- Generalized interfaces in JavaGI [10] (2007–2011);

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.
Drawbacks of the Existing Solutions

Neither of the extensions supports all the features:

1. multiple models;
2. associated types;
3. subtype constraints;
4. supertype constraints;
5. concept-based overloading;
6. multiple dynamic dispatch.

The extensions are implemented via translation to the basic language, but:

1. resulting generic classes contain extra fields, whereas generic functions take extra arguments (this brings run-time overhead);
2. translation is not reversible (this breaks separate compilation).
Research Problem

Research Track

1. To identify key **problems** of object-oriented languages with respect to their support for generic programming.

2. To **design** a **language extension** for C# that improves means of generic programming in the language.

3. To develop a **type-safe model** of the extension for FGJ [12].

4. To provide a “**proof-of-concept” implementation** of the extension for C#.
Research Problem

Research Track

1. To identify key problems of object-oriented languages with respect to their support for generic programming.

2. To design a language extension for C# that improves means of generic programming in the language.

3. To develop a type-safe model of the extension for FGJ [12].

4. To provide a “proof-of-concept” implementation of the extension for C#.
Contents

1. Generic Programming
2. Research Problem
3. Concept Parameters
Concepts and Generic Code in Cp#
(Cp# stands for C# extended with concept parameters)

Concepts:

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

class Ordering[T] refines Equality[T]
{
    int Compare(T x, T y);
    bool Less(T x, T y) {...} ... 
}

class Unifying[Tm, Eqtn, Subst]
{
    Subst Solve(IEnumerable<Eqtn> eqs); ... } 
```

Generic Code:

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

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

class HashSet<T | Equality[T] eq> ... 
```
// default case-sensitive equality comparison
model default EqStringCaseS for Equality[string] { ... }

// case-insensitive equality comparison
model EqStringCaseIS for Equality[string]
{
    bool Equal ( string x , string y )
    {
        return x.ToLower() == y.ToLower();
    }
}

// default lexicographical ordering
model default OrdStringCSAsc for Ordering[string]
    refines EqStringCaseS { ... }

Models consistency is provided!

var s1 = new HashSet<string>(...); // s1 : HashSet<string|EqStringCaseS>
var s2 = new HashSet<string | EqStringCaseIS>(...);
s1.UnionWith(s2); // static ERROR: s1 and s2 have different types
Translation of Cp# to C#

- concept ⇒ generic interface
- model ⇒ class
- generic code ⇒ generic code with extra type arguments

```csharp
static class ConceptSingleton<C> where C : new()
{
    public static C Instance ... }

interface Equality<T> { bool Equal(T x, T y); }
interface Ordering<T> : Equality<T> { ... }

bool Contains<T, eq>(IEnumerable<T> vs, T x)
    where eq : Equality<T>, new()
{ ... if (ConceptSingleton<eq>.Instance.Equal(...) ... }

interface ISet<T, eq> where eq : Equality<T>, new() { ... }
class SortedSet<T, ord> : ISet<T, ord>
    where ord : Ordering<T>, new() { ... }

class EqStringCaseIS : Equality<string> { ... }
```
Benefits of the Translation Method

1. Extra **compile-time** type arguments are used instead of run-time class fields/function arguments.
2. Supplemented with attributes, the translation becomes **reversible**!

Why Is It Possible?

Because .NET CIL (Common Intermediate Language) preserves information on type parameters of generics.


References II


References III


### Comparison of Languages and Extensions

**Language Support for GP in OO Languages**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Haskell</th>
<th>C#</th>
<th>Java 8</th>
<th>Scala</th>
<th>Ceylon</th>
<th>Kotlin</th>
<th>Rust</th>
<th>Swift</th>
<th>JavaGI</th>
<th>Go</th>
<th>C#pt</th>
<th>Genus</th>
<th>ModImpl</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Constraints can be used as types</strong></td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
</tr>
<tr>
<td><strong>Explicit self types</strong></td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
</tr>
<tr>
<td><strong>Multi-type constraints</strong></td>
<td>⬜️</td>
<td>*</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
</tr>
<tr>
<td><strong>Retroactive type extension</strong></td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
</tr>
<tr>
<td><strong>Retroactive modeling</strong></td>
<td>⬜️</td>
<td>*</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
</tr>
<tr>
<td><strong>Type conditional models</strong></td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
</tr>
<tr>
<td><strong>Static methods</strong></td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
</tr>
<tr>
<td><strong>Default method implementation</strong></td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
</tr>
<tr>
<td><strong>Associated types</strong></td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
</tr>
<tr>
<td><strong>Constraints on associated types</strong></td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
</tr>
<tr>
<td><strong>Same-type constraints</strong></td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
</tr>
<tr>
<td><strong>Concept-based overloading</strong></td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
</tr>
<tr>
<td><strong>Multiple models</strong></td>
<td>⬜️</td>
<td>🇨 órgão</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
</tr>
<tr>
<td><strong>Models consistency (model-dependent types)</strong></td>
<td>⬜️</td>
<td>🇨 órgão</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
</tr>
<tr>
<td><strong>Model genericity</strong></td>
<td>⬜️</td>
<td>🇨 órgão</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
</tr>
<tr>
<td><strong>Multiple dynamic dispatch</strong></td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
<td>⬜️</td>
</tr>
</tbody>
</table>

☆ means support via the Concept pattern.  

*a* G supports lexically-scoped models but not really multiple models.  

*b* If multiple models are not supported, the notion of model-dependent types does not make sense.
With the Concept design pattern [13] (“Type Classes As Objects and Implicits” by Oliveira et. al., 2010), constraints on type parameters are replaced with extra arguments — “concepts”.

F-Bounded Polymorphism

```csharp
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

```csharp
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)// *
      ...
}
```
In Scala it has a special support: **context bounds and implicits.**

**F-Bounded Polymorphism**

```scala
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**

```scala
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]) {
  ... }
```
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;

```csharp
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.

```csharp
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

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

### Drawbacks

1. **runtime overhead (extra class fields or function arguments);**
2. **models inconsistency.**

```csharp
interface IEqualityComparer<T>
{
    ... 
}

class HashSet<T> : ...
{
    IEqualityComparer<T> Comparer;

    ... 
}

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

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