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

Contents

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.

Constrained Generic Code

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

```
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
    ...
```

Constrained Generic Code

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

```
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#)

```
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#)

```
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 };
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!

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

Contents

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?

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);
- Swift (2014, 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

```
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

```
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

(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>
                              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.

Related Work

There are several language extensions for generic programming influenced 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);
- 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.

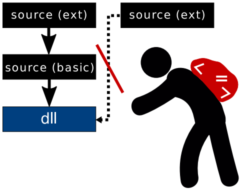
Drawbacks of the Existing Solutions

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 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 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

Concepts and Generic Code in Cp#

(Cp# stands for C# extended with concept parameters)

Concepts:

```
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);
    bool Less(T x, T y) {...} ... }

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

Generic Code:

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

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

class HashSet<T | Equality[T] eq> ...
```

Models in Cp#

```
// 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

```
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 I

[1] 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.

[2] 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.

[3] 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.

[4] 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

[5] J. Belyakova. “Language Support for Generic Programming in Object-Oriented Languages: Peculiarities, Drawbacks, Ways of Improvement”. In: *Proceedings of the Institute for System Programming* (2016), to appear.

[6] 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.

[7] 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

[8] 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.

[9] 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.

[10] 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.

[11] 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

[12] Igarashi, Atsushi and Pierce, Benjamin C. and Wadler, Philip. "Featherweight Java: A Minimal Core Calculus for Java and GJ". In: *ACM Trans. Program. Lang. Syst.* 23.3 (May 2001), pp. 396–450.

[13] 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.

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	○	●	●	●	●	●	●	●	◐	○	○	○	○
Explicit self types	—	○	○	◐	●-	○	●	●	◐	—	—	—	—
Multi-type constraints	●	⚹	⚹	⚹	○	⚹	○	○	●	●	●	●	●
Retroactive type extension	—	●	○	○	○	●	●	●	○	○	○	○	—
Retroactive modeling	●	⚹	⚹	⚹	○	⚹	●	●	●	●	●	●	●
Type conditional models	●	○	○	○	○	○	●	○	●	●	●	●	●
Static methods	●	○	●	○	●	●	●	●	●	●	●	●	●
Default method implementation	●	○	●	●	●	●	●	●	◐	●	●	○	○
Associated types	●	○	○	●	○	○	●	●	○	●	●	○	●
Constraints on associated types	◐	—	—	●	—	—	●	●	—	●	●	—	●
Same-type constraints	◐	—	—	●	—	—	●	●	—	●	●	—	●
Concept-based overloading	○	○	○	○	○	○	●	○	○	◐	○	○	○
Multiple models	○	⚹	⚹	⚹	⚹	⚹	○	○	○	◐ [a]	●	●	●
Models consistency (model-dependent types)	— [b]	○	○	○	○	○	— [b]	— [b]	— [b]	— [b]	●	●	●
Model genericity	—	⚹	⚹	⚹	⚹	⚹	●	○	○	○	○	●	—
Multiple dynamic dispatch	—	○	○	○	○	○	○	○	◐	○	○	●	—

⚹ 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.

Concept Pattern I

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

```
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)// *
      ...
```

Concept Pattern II

In Scala it has a special support: context bounds and implicits.

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])
{ ... }
```

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

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.

```
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)`!