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

Julia Belyakova
julbel@sfedu.ru

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

May 30th 2016
Spring/Summer Young Researchers’ Colloquium on Software Engineering SYRCoSE ’2016
1. Generic Programming
2. Language Support for GP in Object-Oriented Languages
3. Language Extensions for Generic Programming
4. Conclusion
A term “Generic Programming” (GP) was coined in 1989 by Alexander Stepanov and David Musser [1].

**Idea**

A code is written in terms of abstract types and operations.

**Purpose**

Writing highly reusable code.
An Example of Unconstrained Generic Code (C#)

```csharp
static int Count<T>(IEnumerable<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#)

```csharp
static int Count<T>(IEnumerable<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`

Count<T> can be instantiated with any type!
**An Example of Unconstrained Generic Code (C#)**

```csharp
static int Count<T>(IEnumerable<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`.

Count<T> can be instantiated with any type!

```csharp
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
```
When Constraints are Needed

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

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

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

"Being comparable" is a constraint.
When Constraints are Needed

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

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

"Being comparable" is a constraint.
How to write a *generic* function that finds maximum element in a collection?

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

To find maximum in `vs`, values of type `T` must be *comparable*!
How to write a generic function that finds maximum element in a collection?

```csharp
static T FindMax<T>(IEnumerable<T> vs)
{
    // vs check
    T mx = vs.First();
    foreach (var v in vs)
    {
        if (mx < v) // ERROR: operator '<'
            ... // 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 IComparable<T> { int CompareTo(T other); }

static T FindMax<T>(IEnumerable<T> vs) where T : IComparable<T>
{
    // vs check
    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 IComparable<T> { int CompareTo(T other); }

static T FindMax<T>(IEnumerable<T> vs) where T : IComparable<T>
{
    // vs check
    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.
An Example of Generic Code with Constraints (C#)

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

static T FindMax<T>(IEnumerable<T> vs) where T : IComparable<T>
{
    // vs check
    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.

```csharp
int[] ints = new int[]{ 3, 2, -8, 61, 12 };
int iMx = FindMax(ints);    // 61

string[] strs = new string[]{ "hi", "bye", "hello", "stop" };
string sMx = 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!
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].
Motivation for the Study

Poor Language Support for Generic Programming

Is it a problem of C# and Java only? Or is it a \textbf{typical} problem of \textbf{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);
Constraints as Types

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

The “Constraints-are-Types” Approach

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

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

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

The “Constraints-are-Types” Approach

Interface-like language constructs are used in a 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:

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

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

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

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 IConstraint1<A, B>     where A : IConstraint1<A, B>
           where B : IConstraint2<A, B> {...}
  interface IConstraint2<A, B>     where A : IConstraint1<A, B>
           where B : IConstraint2<A, B> {...}
  double Foo<A, B>(A[] xs) where A : IConstraint1<A, B>
                      where B : IConstraint2<A, B> {...}
  ```
Inevitable Limitations

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

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

we have:

```java
interface IConstraint1<A, B> where A : IConstraint1<A, B>
where B : IConstraint2<A, B> {...}
```

```java
interface IConstraint2<A, B> where A : IConstraint1<A, B>
where B : IConstraint2<A, B> {...}
```

```java
double Foo<A, B>(A[] xs)
where A : IConstraint1<A, B>
where B : IConstraint2<A, B> {...}
```

Multiple models cannot be supported at language level.
Concept Pattern I

With the Concept design pattern [5], constraints on type parameters are replaced with extra arguments — “concepts”.

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

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

Advantages: both limitations are eliminated

1. multi-type constraints are multi-type “concept” arguments;
2. multiple “models” are allowed as long as several classes can implement same interface.
The Concept design pattern is widely used in standard generic libraries of C#, Java, and Scala, but it has serious problems!

### Drawbacks

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

#### Models Inconsistency

```csharp
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)!
Alternative Approach

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

- Generalized interfaces in JavaGI [10] (2007–2011);
- Concepts for C# [3] (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.
interface Iterable[T] { ... }

custom constraint Eq[T] { boolean T.equals(T other); }
custom 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 = ...;
Set[String with StringCIEq] s2 = ...;
s1 = s2; // Static ERROR, s1 and s2 have different types

Figure: Constraints Consistency
Which Approach is Better?

“Constraints-are-Types”

Lack of language support for multi-type constraints and multiple models, with Concept 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.
There are at least three reasons for this assertion:
“Constraints-are-Not-Types” Is Preferable

There are at least three reasons for this assertion:

- According to [12], 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 three reasons for this assertion:

- According to [12], 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 facilities for generic programming.
“Constraints-are-Not-Types” Is Preferable

There are at least three reasons for this assertion:

- According to [12], 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 facilities for generic programming.
- As for other features important for generic programming, they can be supported using any approach.
## Comparison of Languages and Extensions

<table>
<thead>
<tr>
<th>Language Support for GP in OO Languages</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>G</th>
<th>C#&lt;sup&gt;pt&lt;/sup&gt;</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></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></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></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. <sup>a</sup>G supports lexically-scoped models but not really multiple models. 
<sup>b</sup>If multiple models are not supported, the notion of model-dependent types does not make sense.
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**

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

```java
interface Set[T where Eq[T]] {...}
Set[String] s1 = ...;
Set[String with StringCIEq] s2=...;
```

**Concept Parameters**

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

```java
interface Set<T ! Eq[T] eq> {...}
Set<String> s1 = ...;
Set<String ! StringCIEq> s2 = ...;
```
References I


References II


