Pitfalls of C# Generics and Their Solution Using Concepts

Julia Belyakova, Stanislav Mikhalkovich

Institute for Mathematics, Mechanics and Computer Science named after I. I. Vorovich
Southern Federal University

28th May, 2015

Table of Contents

Mechanisms of Generic Programming

1. **Unconstrained C++ Templates**.

2. **Mechanisms Based on Explicit Constraints**: C# and Java generics, Haskell type classes, ML signatures, Scala traits, etc.

Mechanisms of Generic Programming

1. **Unconstrained C++ Templates**.
   - flexibility;
   - expressiveness;
   - late stage of error detection;
   - unclear error messages.
2. **Mechanisms Based on Explicit Constraints**: C# and Java generics, Haskell type classes, ML signatures, Scala traits, etc.

Mechanisms of Generic Programming

1. **Unconstrained C++ Templates**.
   - flexibility;
   - expressiveness;
   - late stage of error detection;
   - unclear error messages.
2. **Mechanisms Based on Explicit Constraints**: C# and Java generics, Haskell type classes, ML signatures, Scala traits, etc.
   - early stage of error detection;
   - error messages in terms of constraints;
   - (in most cases) weaker expressiveness.

Explicit-Constraints-Based Mechanisms

How do they differ?

1. Support different kinds of constraints/requirements (function signatures, associated types, same-type constraints, etc.)
2. Provide different features (retroactive modeling, multi-type constraints, constraints-propagation, etc.)

- Haskell Type Classes (the most powerful)
- Scala Traits
- ...
- C#/Java Generics (one of the poorest)

Explicit-Constraints-Based Mechanisms

How do they differ?

1. Support different kinds of constraints/requirements (function signatures, associated types, same-type constraints, etc.)
2. Provide different features (retroactive modeling, multi-type constraints, constraints-propagation, etc.)

- Haskell Type Classes (the most powerful)
- Scala Traits
- ...
- C#/Java Generics (one of the poorest)
- C++ Concepts

Concepts

A term "concept" comes from the Standard Template Library (STL).

C++ Concepts as a new **language construct**:

- are underway in C++ community since 2000 (Bjarne Bjarne, Gabriel Dos Reis, Douglas Gregor, Jaakko Järvi and others);
- in respect to expressive power are comparable with Haskell type classes;
- are as effective as templates;
- do not suffer from templates diseases;
- are treated as a possible substitution of unconstrained templates;

Concepts

A term "concept" comes from the Standard Template Library (STL).

C++ Concepts as a new **language construct**:

- are underway in C++ community since 2000 (Bjarne Bjarne, Gabriel Dos Reis, Douglas Gregor, Jaakko Järvi and others);
- in respect to expressive power are comparable with Haskell type classes;
- are as effective as templates;
- do not suffer from templates diseases;
- are treated as a possible substitution of unconstrained templates;
- are still not included in C++.

The Goal of This Study

To introduce concepts into C# language to improve current mechanism of generic programming.

C# Generics (based on F-bounded polymorphism): constraints on type parameters of generic classes and methods are expressed in terms of interfaces and subtyping.

Concepts can be used with interfaces simultaneously.

Why C#?

There are two aspects:

1. A Design. The design of concepts proposed is applicable to C#, Java and any .NET language with GP mechanism based on *F-bounded polymorphism*.
2. An Implementation. The method of concepts *translation* is strongly oriented to .NET Framework.

C# is suitable both for
syntax demonstration and implementation.

Concept Sample I

Concept

- represents some abstraction;
- defines a named set of requirements on type parameters.

Monoid example

```
concept CMonoid[T]
{
    T binOp(T x, T y);
    T ident;
}
```

```
static T Accumulate<T>(T[] values)
    where CMonoid[T] using cM
{
    T result = cM.ident;
    foreach (T val in values)
        result = cM.binOp(result, val);
    return result;
}
```

Concept Sample II

Monoid example in C#

```
interface IMonoid<T>
    where T : IMonoid<T>
{
    T binOp(T other);
}
```

```
static T Accumulate<T>(
    T[] values, T ident
)
    where T : IMonoid<T>
{
    T result = ident;
    foreach (T val in values)
        result = result.binOp(val);
    return result;
}
```

C# Pitfalls

- lack of retroactive interface implementation;
- recursive constraints;
- constraints-compatibility problem;
- multi-type constraints problem;
- constraints duplication;
- verbose type parameters.

Constraints Compatibility Problem

Generics

```
class HashSet<T>
  (IEqualityComparer<T>)

static HashSet<T> GetUnion<T>(
    HashSet<T> s1,
    HashSet<T> s2
){
  var us = new HashSet<T>(
     s1, s1.Comparer
  );
  us.UnionWith(s2);
  return us;
}

// GetUnion(s1, s2)
// != GetUnion(s2, s1)
```

Generics with Concepts

```
class HashSet<T>
where CEqualityComparable[T]

static HashSet<T> GetUnion<T>(
    HashSet<T> s1,
    HashSet<T> s2
){
  var us = new HashSet<T>(
      s1
  );
  us.UnionWith(s2);
  return us;
}

// GetUnion(s1, s2)
// == GetUnion(s2, s1)
```

The Sketch of Translation

Owing to the properties of the .NET Framework:

1. A resultant code of translation is generic.
2. Meta-information is preserved via attributes.

- Concept — abstract generic class. Type parameters and nested concept requirements — type parameters of this generic class.
- Generic class — generic class with extra type parameters for concept requirements.
- Model — class, a subtype of the corresponding abstract generic class of concept.
- Instantiation of generic class — instantiation of the corresponding generic class with extra type parameters.

The Advantages of Translation

1. Lowering the run-time expenses due to passing concepts as *types* (in contrast to G concepts [4] and Scala "concept pattern" [3]).

2. Modularity can be provided due to preserving full type information and meta-information.

Comparison of “Concepts” Designs Under Garcia et. al. [1]

Feature	G	C++	$\text{C\#}^{\text{ext}}$	JGI	Scl	$\text{C\#}^{\text{cpt}}$
multi-type constraints	+	+	±[1]	+	+[2]	+
associated types	+	+	+	−	+	+
same-type constraints	+	+	+	−	+	+
subtype constraints	−	−	+	+	+	+
retroactive modeling	+	+	±[1]	+	+[3]	+
multiple models	+	−	±[1]	−	+	+
anonymous models	−	−	−	−	+[3]	+
concept-based overloading	+	+	−	−	±[4]	−
constraints-compatibility	+	+	−	+	−	+

“$\text{C\#}^{\text{ext}}$” means C# with associated types [2].
“Scl” means Scala [3].
“$\text{C\#}^{\text{ext}}$” means C# with concepts.

[1] partially supported via “concept pattern”
[2] supported via “concept pattern”
[3] supported via “concept pattern” and implicits
[4] partially supported by prioritized overlapping implicits

Future Work

- Formalization of translation.
- Implementation of C# compiler for restricted language.
- Concept syntax “approbation”.

References

[1] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock.
An Extended Comparative Study of Language Support for Generic Programming.
J. Funct. Program., 17(2):145–205, March 2007.

[2] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine.
Associated Types and Constraint Propagation for Mainstream Object-oriented Generics.
In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 1–19, New York, NY, USA, 2005. ACM.

[3] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky.
Type Classes As Objects and Implicits.
In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, New York, NY, USA, 2010. ACM.

[4] Jeremy G. Siek.
A Language for Generic Programming.
PhD thesis, Indianapolis, IN, USA, 2005.
AAI3183499.

Monoid Concept Translation

Monoid Concept

```
concept CMonoid[T]
{
    T binOp(T x, T y);
    T ident;
}
```

```
abstract class CMonoid<T>
{
    public abstract T binOp(T x, T y);
    public abstract T ident { get; };
}
```

Generic Method

```
static T Accumulate<T>(
    T[] values
)
    where CMonoid[T]
{ ... }
```

```
static T Accumulate<T, CMonoid_T>(
    T[] values
)
    where CMonoid_T : CMonoid<T>
{ ... }
```