Pitfalls of C# Generics and Their Solution Using Concepts

Julia Belyakova
Institute for Mathematics, Mechanics
and Computer Science
Southern Federal University
Rostov-on-Don, Russia
Email: julbel@sfedu.ru

Stanislav Mikhalkovich
Institute for Mathematics, Mechanics
and Computer Science
Southern Federal University
Rostov-on-Don, Russia
Email: miks@sfedu.ru

***Abstract*—In comparison with Haskell type classes and C++ concepts, such object-oriented languages as C# and Java provide much limited mechanisms of generic programming based on F-bounded polymorphism. Main pitfalls of C# generics are considered in this paper. Extending C# language with *concepts* which can be simultaneously used with interfaces is proposed to solve the problems of generics; a design and translation of concepts are outlined.**

I. Introduction

Generic programming is supported in different programming languages by various techniques such as C++ templates, C# and Java generics, Haskell type classes, etc. Some of these techniques were found more expressive and suitable for generic programming, other ones more verbose and worse maintainable [1]. Thus, for example, the mechanism of expressive and flexible C++ unconstrained templates suffers from unclear error messages and a late stage of error detection [2], [3]. New language construct called **concepts**[1] was proposed for C++ language as a possible substitution of unconstrained templates. A design of C++ concepts[2] conforms to main principles of effective generic tools design [1].

In comparison with concepts and Haskell type classes [1], [7], such mainstream object-oriented languages as C# and Java provide much limited mechanisms of generic programming based on F-bounded polymorphism. Pitfalls of C# generics are analysed in this paper in detail (Sec. II): we discuss some known drawbacks and state the problems of subtle semantics of recursive constraints (Sec. II-B) and constraints-compatibility (Sec. II-C). To manage the pitfalls considered extending of C# with concepts is proposed: a design of concepts is briefly presented in Sec. IV. We also discuss a translation of such extension to standard C#.

C# language is used in this paper primarily for the sake of syntax demonstration. As for the pitfalls of C# generics, they hold for Java as well with slight differences. However, while the concepts *design* proposed in the paper could be easily adapted for Java (and also for any .NET-language with interface-based generics), the technique of language extension *translation* (which we consider in Sec. IV) cannot be applied for Java directly. Unlike Java Virtual Machine, .NET Framework preserves type information in its byte code, this property being crucial for the translation method.

[1] Term "concept" was initially introduced in a documentation of the Standard Template Library (STL) [4] to describe requirements on template parameters in informal way.

[2] There were several designs of C++ concepts [3], [5], [6]; all of them share some general ideas.

```
(ICmp-1)   interface IComparable<T> {int CompareTo(T other);}
(ICmp-2)   interface IComparer<T>   {int Compare(T x, T y);}

(s-1)      Sort<T>(T[]) where T : IComparable<T>;
(s-2)      Sort<T>(T[], IComparer<T>);
```

Fig. 1. IComparable<T>/IComparer<T> interfaces and its applications

II. Pitfalls of C# Generics

C# and Java interfaces originally developed to be an entity of object-oriented programming were later applied to generic programming as constraints on generic type parameters. There are several shortcomings of this approach.

A. Lack of Retroactive Interface Implementation

Interfaces cannot be implemented *retroactively*, i. e. it is impossible to add the relationship "type T implements interface I" if type T is already defined. Consider a generic algorithm for sorting arrays Sort<T> with the following signature:

```
Sort<T>(T[]) where T : IComparable<T>;
```

If some type Foo provides an operation of comparison but does not implement the interface IComparable<Foo>, Sort<Foo> is not a valid instance of Sort<>. What one can do in this case? If type cannot be changed (it may be defined in external .dll, for instance), the only way to cope with sorting is to define an adapter class FooAdapter which implements Sort<FooAdapter> interface, pack all Foo objects into FooAdapter ones, sort them and unpack back to an array of Foo objects. Apparently, there must be a better approach.

Fortunately, in the .NET Framework standard library the Array.Sort<T> method [8] is provided with two "branches" of overloads:

1) For any type T which implements IComparable<T> interface ((s-1) example, Fig. 1).

```
(1)  interface IComparableTo<S> { int CompareTo(S other); }

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

Fig. 2. `IComparable<T>` vs `IComparableTo<S>` example

2) For any type `T` with an external comparer of type `IComparer<T>` provided (`(s-2)` example, Fig. 1).

Hence, if some type is already defined, values of this type can be compared, but this type does not implement `IComparable<>` interface (as in the `Foo` example above), `Sort<>` with `IComparer<>` (branch 2) is to be used. Thus one can simulate retroactive modeling property (in Scala the similar approach is referred to as a programming with the *"concept pattern"* [9]). Consequently, if retroactive modeling is required, a programmer has to write a generic code twice — in "interface-oriented" and in "concept pattern" styles. The amount of necessary overloads grows exponentially: if one needs two retroactively modeled constraints on generic type, corresponding generic code would consist of four "twins", if three — eight "twins" and so on.

B. Drawbacks of Recursive Constraints

Example 1. The following reason about the `Sort<T>` method for `IComparable<T>` may be not obvious. The notation of `Sort<T>` in `(s-1)` example (Fig. 1) looks a little bit redundant; such a recursive constraint on type `T` might look even frightening, but it is well formed. Furthermore, the word "comparable" in this context is very likely associated with the ability to compare values of type `T` with each other. But the interface `IComparable<T>` (`(ICmp-1)`, Fig. 1) does not correspond this semantics: it designates the ability of some type (which implements this interface) to be comparable with type `T`. The same problem with `Comparable<X>` interface in Java is explored in [10]. The particular role of recursive constraints in generic programming is explored in [11].

It would be better to split the single `IComparable<>` interface into two different interfaces (Fig. 2):

1) `IComparableTo<S>` which requires some type (which implements this interface) to be comparable with `S`.
2) `IComparable<T>` which requires values of type `T` to be comparable with each other.

Note that the definition of the latter interface needs the constraint `where T : IComparable<T>` (q.v. Fig. 2).

Example 2. As an another example consider a generic definition of graph with peculiar structure: graph stores some data in vertices; every vertex contains information about its predecessors and successors thereby defining arcs. A graph itself consists of set of vertices instead of set of edges. Such kind of graph is suitable for a task of data flow analysis in the area of optimizing compilers [12] because "movement along arcs up and down" is intensively used action in an analysis of a control flow graph.

Fig. 3 illustrates parts of the corresponding definitions: `IDataGraph<Vertex, DataType>` describes interface of a data graph; `IDataVertex<Vertex, DataType>` describes interface of a vertex in such graph. While the graph interface really depends on type parameters `Vertex` and `DataType`, we have to include `Vertex` as a type parameter into the vertex interface `IDataVertex<,>` as well. Similarly to `IComparable<>` example the constraints (*) and (#) in Fig. 3 are not superfluous. Suppose we have the following types:

```
class V1 : IDataVertex<V1, int> { ... }
class V2 : IDataVertex<V1, int> { ... }
```

Thanks to the constraints (*) and (#) the instantiation of graph `IDataGraph<V2, int>` is not allowed, since type `V2` does not implement interface `IDataVertex<V2, int>`. Without these constraints we might accept some inconsistent graph with vertices of type `V2` which refer to vertices of type `V1`.

Vertex and graph interface definitions are unclear and non-obvious. If programmers might be used to use interface `IComparable<>`, it is more difficult to manage such things as `IDataGraph<,>` example. In some cases one may prefer to abandon writing generic code because of this awkwardness.

```
interface IDataVertex<Vertex, DataType>
    where Vertex : IDataVertex<Vertex, DataType>    // (*)
{   ...
    IEnumerator<Vertex> OutVertices { get; }
    ... }
interface IDataGraph<Vertex, DataType>
    where Vertex : IDataVertex<Vertex, DataType>    // (#)
{   ... }
```

Fig. 3. `IDataGraph<,>` and `IDataVertex<,>` interfaces

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

Fig. 4. Union of `HashSet<T>` objects

C. Ambiguous Semantics of Generic Types

When using flexible `Sort<T>` method with an external `IComparer<T>` parameter (Fig. 1), a programmer has clear understanding of how elements are sorted, since such a comparer is a *parameter of an algorithm*. But when one uses generic *types*, this information is implicit. For instance, `SortedSet<T>` class takes `IComparer<T>` object as a constructor parameter, `HashSet<T>` class taking `IEqualityComparer<T>`. Therefore, given two sets of the same generic type one cannot check at *compile time* whether these sets are *constraints-compatible* (in case of `HashSet<T>` "constraints-compatibility" means that the given sets use the same equality comparer). And it seems that a programmer usually does not suppose that objects of the same type can have different comparers (or addition operators, coercions, etc). But they can, and it leads to subtle errors.

Suppose we have a simple function `GetUnion<T>` (q.v. Fig. 4) which returns a union of the two given sets. If some arguments `a` and `b` provide different equality comparers (e.g., case-sensitive and case-insensitive comparers for type `string`), the result of `GetUnion(a, b)` would differ from the result of `GetUnion(b, a)`. Note that Haskell type classes do not suffer

```
interface IObserver<O, S> where O : IObserver<O, S>
                          where S : ISubject<O, S>
{   void update(S subj); }

interface ISubject<O, S>  where O : IObserver<O, S>
                          where S : ISubject<O, S>
{   List<O> getObservers();
    void register(O obs);
    void notify();        }
```

Fig. 5. Observer pattern in C#

from such an ambiguity because every type provides only one instance of a type class.

D. The Problem of Multi-Type Constraints

The well-known problem of multi-type constraints holds for C# interfaces. Requirements concerning on several types cannot be naturally expressed within interfaces. The paper [10] deals with the example of Observer pattern in Java. The Observer pattern connects two types: Observer and Subject. Both types has methods which take the another type of this pair as an argument: the Observer provides `update(Subject)`, the Subject — `register(Observer)`.

Fig. 5 shows the interface definitions `IObserver<O, S>` for Observer and `ISubject<O, S>` for Subject in standard C#. We need two different interfaces and have to duplicate the constraints on `O` and `S` in both definitions to establish consistent connection between type parameters `O` and `S`. And again we face with recursive constraints on types `O` (which represents the Observer) and `S` (which represents the Subject). This example looks even worse than the case of vertex and graph interfaces presented in Fig. 3. But it is the only way to define a type family [13] of Observer pattern correctly.

E. Constraints Duplication and Verbose Type Parameters

All constraints required by a definition of generic type are to be repeatedly specified in every generic component which uses this type. Consider the generic algorithm `GetSubgraph<,,>` depending on type parameter `G` which implements `IDataGraph<,>` interface (q.v. Fig. 3).

```
G GetSubgraph<G, Vertex, DataType>(
    G g, Predicate<DataType> p)
where G : IDataGraph<Vertex, DataType>, new()
where Vertex : IDataVertex<Vertex, DataType> { ... }
```

`GetSubgraph<G, Vertex, DataType>` method is not correct without explicit specification of constraint on type parameter `Vertex`. This constraint is induced by the definition of `IDataGraph<Vertex, DataType>` interface and should be repeated every time one uses `IDataGraph<,>`.

Another property of `GetSubgraph<...>` definition is a plenty of generic parameters. Clearly, vertex and data types are fully determined by the type of specific graph. At the level of `GetSubgraph<...>` signature vertex type even does not matter at all. Such types are often referred to as *associated types*. Some programming languages allow to declare associated types explicitly (SML, C++ via traits, Scala via abstract types and some other), but in C# and Java they can only be represented by extra type parameters. It makes generic definitions verbose and breaks encapsulation of constraints on associated types. Issues of repeated constraints specification and lack of associated types are considered in [14], [1] in more detail.

III. Related Work

We consider two studies concerning modification of generic interfaces in this section:

1) [14] proposes the extension of C# generics with associated types and constraint propagation.
2) [10] generalizes Java 1.5 interfaces enabling retroactive interface implementation, multi-headed interfaces (expressing multi-type constraints) and some other features.

Both studies *revise interfaces* to improve interface-based mechanism of generic programming and to approach to C++ concepts and Haskell type classes, which are considered being rather similar [7]. Some features of Scala language in respect to problems considered in Sec. II will also be mentioned.

A. C# with Associated Types and Constraint Propagation

Member types in interfaces and classes are introduced in [14] to provide direct support of *associated types*. A mechanism of *constraint propagation* is also proposed to lower verbosity of generic components and get rid of constraints duplication as was mentioned in Sec. II-E. The example of Incidence Graph concept from the Boost Graph Library (BGL) [15] is considered. It is shown that features proposed can significantly improve a support of generic programming not only in C# language but in any object-oriented language with F-bounded polymorphism.

But the problems of multi-type constraints and recursive constraints cannot be solved with this extension. Thus, the code of Observer pattern (Fig. 5) cannot be improved at all because of recursive constraints; the same holds for `IComparable<T>` interface. The issue of retroactive implementation is also not touched upon in [14]: extended interfaces are still interfaces which cannot be implemented retroactively.

B. JavaGI: Java with Generalized Interfaces

In contrast to [14], the study [10] is mainly concentrated on the problems of retroactive implementation, multi-type constraints (solved with *multi-headed interfaces*) and recursive interface definitions[3]. For instance, Observer pattern is expressed in JavaGI with generalized interfaces as shown in Fig. 6 [10]. Methods of a whole interface are grouped by a receiver type with keyword `receiver`. A syntax of an interface looks a little bit verbose but it is essentially better than two interfaces with duplicated constraints shown in Fig. 5. Moreover, JavaGI interfaces allow *default implementation* of methods (as `register` and `notify`). Retroactive implementation of interfaces is also allowed, but it is possible to define only one implementation of an interface for the given set of types in a namespace.

[3] This problem is usually connected with so-called *binary methods problem*.

```
interface ObserverPattern[O, S] {
    receiver O { void update(S subj); }
    receiver S {
        List<O> getObservers();
        void register(O obs) { getObservers().add(obs); }
        void notify() { ... }
    }
}
class MultiheadedTest {
    <S,O> void genericUpdate(S subject, O observer)
            where [S,O] implements ObserverPattern {
        observer.update(subject);
    }
}
```

Fig. 6. Observer pattern in JavaGI

It turns out that interfaces become some restricted version of C++ concepts [5], [16] (in particular, they do not support associated types) and, moreover, they lose a semantics of object-oriented interfaces[4]. JavaGI interfaces only act as *constraints* on generic type parameters, but they cannot act as types, so one cannot use JavaGI interfaces as in Java.

C. "Concept Pattern" and Context Bounds in Scala

The idea of programming with "concept pattern" has been reflected in Scala language [9]. Due to the combination of generic *traits* (something like interfaces with abstract types and implementation), *implicits* (objects used by default as function arguments or class fields) and *context bounds* (like `T : Ordering` in Fig. 7) Scala provides much more powerful mechanism of generic programming than C# or Java. Fig. 7 illustrates the examples of sorting and observer pattern.

Context bounds provide simple syntax for single-parameter constraints: the sugared `(s-s)` version of `Sort[T]` algorithm is translated into `(s-u)` one by desugaring. Retroactive modeling is supported since one can define new `Ordering[]` object and use it for sorting. And one does not need to provide two versions of the sort algorithm as for C# language (q.v. Fig. 1): `Sort[]` with one argument would use default ordering due to `implicit` keyword. `ObserverPattern[S, O]` looks rather similar to corresponding JavaGI interface (Fig. 6). There is no syntactic sugar for multi-parameters traits, so the notation of `genericUpdate[S, O]` cannot be shortened.

In respect to the *constraints-compatibility* problem discussed in Sec. II-C Scala's "concept pattern" reveals the same drawback as C#. Generic types take "concept objects" as constructor parameters. In such a way `TreeSet[A]` [17] implicitly takes `Ordering[A]` object, therefore, for instance, the result of intersection operation would depend on an order of arguments if they use different ordering.

IV. Design of Concepts for C# Language

A. Interfaces and Concepts

It seems that a *new language construct* for generic programming should be introduced into such object-oriented languages as C# or Java. If we extend interfaces preserving their object-oriented essence [14], a generic programming mechanism becomes better but still not good enough, since such problems as

[4]The way to preserve compatibility with Java code is considered in [10], but "real interfaces" no longer exist in JavaGI.

```
(s-s)   def Sort[T : Ordering](elems: Array[T]) { ... }
(s-u)   def Sort[T](elems: Array[T])
            (implicit ord: Ordering[T]) { ... }

trait ObserverPattern[S, O] {
    def update(obs: O, subj: S);
    def getObservers(subj: S): Seq[O];
    def setObservers(subj: S, observers: Seq[O]);
    def register(subj: S, obs: O)
    { setObservers(subj, getObservers(subj) :+ obs); }
    def notify(subj: S) { ... }
}
object MultiheadedTest {
    def genericUpdate[S, O](subject: S, observer: O)
            (implicit obsPat: ObserverPattern[S, O]) {
        obsPat.update(observer, subject);
    }
}
```

Fig. 7. `Sort[T]` and `ObserverPattern[S,O]` examples in Scala

retroactive modeling or constraints-compatibility remain. If we make interfaces considerably better for generic programming purposes [10], they lose their object-oriented essence and can no longer be used as types.

We advocate the assertion that *both* features have to be provided in an object-oriented language:

1) Object-oriented **interfaces** which are used as *types*.
2) Some new construct which is used to *constrain* generic type parameters. C++ like **concepts** are proposed to serve this goal.

B. C# with Concepts: Design and Translation

In this section we present a sketch of C# concepts design. Concept mechanism introduces the following constructs into the programming language:

1) **Concept**. Concepts describe a named set of *requirements* (or *constraints*) on one or more types called *concept parameters*.
2) **Model**. Models determine the manner in which specific types *satisfy* concept. Models are external for types; they can be defined later than types. It means that a type can *retroactively* model a concept if it semantically conforms to this concept. Types may have several models for the same concept. In some cases a default model can be implicitly generated by a compiler.
3) **Constraints** are used in generic code to describe requirements on generic type parameters.

Concepts support the following kinds of constraints:

- associated types and associated values;
- function signatures (may have default implementation);
- nested concept requirements (for concept parameters and associated types);
- same-type constraints;
- subtype and supertype constraints;
- aliases for types and nested concept requirements.

The main distinction of C# concepts proposed in comparison with other concepts designs (C++, G [16]) is the support of *subtype* constraints and *anonymous models* (like anonymous classes). Concept-based mechanism of constraining generic type parameters surpasses the abilities of interface-based one.

Construct of extended language	Construct of base language
Concept	Abstract class
Concept parameter	Type parameter
Associated type	Type parameter
Concept refinement	Subtyping
Associated value	Property (only read)
Nested concept requirement	Type parameter
Concept requirement in generic code	Type parameter
Model	Class

Fig. 8. Translation of C# extension with concepts

At the same time interfaces can be used as usual without any restrictions.

Concepts can be implemented in existing compilers via the translation to standard C#. Fig. 8 presents correspondence between main constructs of extended and standard C# languages. To preserve maximum information about the source code semantics, some additional metainformation has to be included into translated code. In particular, one needs to distinguish generic type parameters in the resultant code as far as they may represent concept parameters, associated types or nested concept requirements. To resolve such ambiguities we propose using *attributes*.

The method of translation suggested is strongly determined by the properties of .NET Framework. Due to preserving type information and attributes in a .NET byte code, translated code can be unambiguously recognized as a result of code-with-concepts translation. Moreover, it can be restored into its source form, what means that *modularity* could be provided: having the binary module with definitions in extended language one can add it to the project (in extended language either) and use in an ordinary way.

Fig. 9 illustrates several concept definitions (in the left column) and their translation to standard C# (in the right column). Basic syntax of concepts is shown: concept declarations (start with keyword `concept`), signature constraints, signature constraints with default implementation (`NotEqual` in `CEquatible[T]`), refinement (concept `CComparable[T]` refines `CEquatible[T]`, i.e. it includes all requirements of refined concept and adds some new ones), associated types (`Data` in `CTransferFunction[TF]`), multi-type concept `CObserverPattern[O, S]`, nested concept requirements (`CSemilattice[Data]` in `CTransferFunction[TF]`).

Concepts are translated to generic classes. Function signatures are translated to abstract or virtual (if implementation is provided) class methods. Concept parameters and associated types are represented by type parameters (marked with attributes) of a generic abstract class as well as *nested concept requirements*. For instance, `CSemilattice_Data` type parameter of `CTransferFunction<>` denotes `CSemilattice[Data]` concept requirement because this parameter is attributed with `[IsNestedConceptReq]`, corresponding subtype constraint being in a where-clause.

Some examples of generic code with concept constraints are presented in the left column of Fig. 10. Concept requirements can be used with alias (as `CComparable[T]` in the class of binary search tree). Note that a singular definition of generic component is sufficient. Translated generic code (in the right column) demonstrates significant property of translation: concept requirements are translated into extra type parameters instead of extra method and constructor parameters (as it is in Scala and G [16]). Therefore, constraints-compatibility can be checked at compile time, methods and objects being saved from unnecessary arguments and fields.

Fig. 11 presents the model of concept `CComparable[]` for class `Rational` of rational number. It is translated to derived class `CComparable_Rational_Def` of `CComparable<Rational>` and then used as the second type argument of generic instance `BST<,>`. Fig. 12 demonstrates using of anonymous model to find a number with a numerator equal to 5.

```
static bool Contains<T>(T x, IEnumerable<T> values)
    where CEquatible[T] { ... }
static void TestContains
{
    Rational[] nums = ...;
    var hasNumer5 = Contains[model CEquatible[Rational] {
        bool Equal(Rational x, Rational y)
        { return x.Num == y.Num; }
    }](new Rational(5), nums);
}
```

Fig. 12. Anonymous model example

Feature	G	C++	C#^{ext}	JGI	Scl	C#^{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	+	+	–	+	–	+

"C#^{ext}" means C# with associated types [1].
"Scl" means Scala [9].
"C#^{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

Fig. 13. Comparison of "concepts" designs

V. Conclusion and Future Work

Many problems of C# and Java generics seem to be well understood now. Investigating generics and several approaches to revising OO interfaces, we faced with some pitfalls of these solutions which were not considered yet.

1) Recursive constraints used to solve the binary method problem appear to be rather complex and often do not correspond a semantics assumed by a programmer.
2) The "concept pattern" breaks constraints-compatibility.
3) Using interfaces both as types and constraints on generic type parameters leads to awkward programs with low understandability.

To solve problems considered we proposed to extend C# language with the *new* language construct — **concepts**. Keeping interfaces untouched, concept mechanism provides much better support of the features crucial for generic programming [1]. The support of these features in C# with concepts

```
concept CEquatible[T]
{   bool Equal(T x, T y);    // function signature
    // function signature with default implementation
    bool NotEqual(T x, T y) { return !Equal(x, y); }
}
// refining concept
concept CComparable[T] refines CEquatible[T]
{   int Compare(T x, T y);
    // overrides Equal from refined concept CEquatible[T]
    override bool Equal(T x, T y) { ... }
}
concept CTransferFunction[TF]
{   type Data;      // associated type
    // nested concept requirement
    require CSemilattice[Data];
    Data Apply(TF trFun, Data d);
    TF Compose(TF trFun1, TF trFun2);
}

concept CObserverPattern[O, S]
{   void UpdateSubject(O obs, S subj);
    ICollection<O> GetObservers(S subj);
    void RegisterObserver(S subj, O obs)
    { GetObservers(subj).Add(obs); }
    void NotifyObservers(S subj) { ... }
}
```

```
[Concept] abstract class CEquatible<[IsConceptParam]T>
{   public abstract bool Equal(T x, T y);
    public virtual bool NotEqual(T x, T y)
    { return !this.Equal(x, y); }
}
[Concept] abstract class CComparable<[IsConceptParam]T>
        : CEquatible<T>
{   public abstract int Compare(T x, T y);
    public override bool Equal(T x, T y) { ... }
}
[Concept] abstract class CTransferFunction<
        [IsConceptParam]TF, [IsAssocType]Data,
        [IsNestedConceptReq]CSemilattice_Data>
    where CSemilattice_Data : CSemilattice<Data>, new()
{   public abstract Data Apply(TF trFun, Data d);
    public abstract TF Compose(TF trFun1, TF trFun2);
}

[Concept] abstract class CObserverPattern<
        [IsConceptParam]O, [IsConceptParam]S>
{   public abstract void UpdateSubject(O obs, S subj);
    public abstract ICollection<O> GetObservers(S subj);
    public virtual void RegisterObserver(S subj, O obs)
    { GetObservers(subj).Add(obs); }
    public virtual void NotifyObservers(S subj) { ... }
}
```

Fig. 9. Concept examples and their translation to basic C#

```
static void Sort<T>(T[] values)
    where CComparable[T] { ... }

class BinarySearchTree<T>
    // concept requirement with alias
    where CComparable[T] using cCmp
{
    private BinTreeNode<T> root;
    ...
    private bool AddAux(T x, ref BinTreeNode<T> root)
    {
        ...
        // reference to concept by alias
        if (cCmp.Equal(x, root.data))       return false;
    ...
}
```

```
[GenericFun] static void Sort<[IsGenericParam]T,
        [IsRequireConceptParam]CComparable_T>(T[] values)
    where CComparable_T : CComparable<T>, new() { ... }

[GenericClass] [ConceptAlias("CComparable_T", "cCmp")]
class BinarySearchTree<[IsGenericParam]T,
[IsRequireConceptParam]CComparable_T>
        where CComparable_T : CComparable<T>, new()
{   private BinTreeNode<T> root;
    ...
    private bool AddAux(T x, ref BinTreeNode<T> root)
    {   ...
        CComparable_T cCmp =
            ConceptSingleton<CComparable_T>.Instance;
        if (cCmp.Equal(x, root.data))       return false;
    ...
}
```

Fig. 10. Generic code and its translation to basic C#

```
// class for rational number with properties
// Num for numenator and Denom for denominator
class Rational { ... }
model CComparable[Rational]
{
    bool Equal(Rational x, Rational y)
    { return (x.Num == y.Num) && (x.Denom == y.Denom); }
    int Compare(Rational x, Rational y) { ... }
}
...
    BST<Rational> rations = new BST<Rational>();   // *
```

```
class Rational { ... }
[ExplicitModel] class CComparable_Rational_Def
   : CComparable<Rational>
{
   public override bool Equal(Rational x, Rational y)
   { return (x.Num == y.Num) && (x.Denom == y.Denom); }
   public override int Compare(Rational x, Rational y){...}
}
...
    BST<Rational, CComparable_Rational_Def> rations   // *
       = new BST<Rational, CComparable_Rational_Def>();
```

* "BST" is used instead of "BinarySearchTree" for short.

Fig. 11. Model `CComparable[Rational]` and its translation to basic C#

extension and its comparison with some other generic mechanisms are presented in Fig. 13. The design of C# concepts is rather similar to C++ concepts designs, but it supports subtype and supertype constraints.

We also suggested a novel way of concepts translation: in contrast to G concepts [16] and Scala "concept pattern" [9], C# concept requirements are translated to *type* parameters instead of object parameters; this lowers the run-time expenses on passing extra objects to methods and classes.

Much further investigation is to be fulfilled. First of all, type safety of C# concepts has to be formally proved. The design of concepts proposed seems to be rather expressive, but it needs an approbation. So the next step is developing of the tool for compiling a code in C# with concepts. Currently we are working on formalization of translation from extended language into standard C#.

Acknowledgment

The authors would like to thank the participants of the study group on the foundations of programming languages Vitaly Bragilevsky and Artem Pelenitsyn for discussions on topics of type theory and concepts.

REFERENCES

[1] R. Garcia, J. Jarvi, A. Lumsdaine, J. Siek, and J. Willcock, "An Extended Comparative Study of Language Support for Generic Programming," *J. Funct. Program.*, vol. 17, no. 2, pp. 145–205, Mar. 2007.

[2] B. Stroustrup and G. Dos Reis, "Concepts — Design Choices for Template Argument Checking," C++ Standards Committee Papers, Technical Report N1522=03-0105, ISO/IEC JTC1/SC22/WG21, October 2003.

[3] G. Dos Reis and B. Stroustrup, "Specifying C++ Concepts," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '06. New York, NY, USA: ACM, 2006, pp. 295–308.

[4] M. H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

[5] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine, "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*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 291–310.

[6] B. Stroustrup and A. Sutton, "A Concept Design for the STL," C++ Standards Committee Papers, Technical Report N3351=12-0041, ISO/IEC JTC1/SC22/WG21, January 2012.

[7] J.-P. Bernardy, P. Jansson, M. Zalewski, S. Schupp, and A. Priesnitz, "A Comparison of C++ Concepts and Haskell Type Classes," in *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, ser. WGP '08. New York, NY, USA: ACM, 2008, pp. 37–48.

[8] "`System.Array.Sort(T)` Method," URL: http://msdn.microsoft.com/library/system.array.sort.aspx.

[9] 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*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 341–360.

[10] S. Wehr, R. Lmmel, and P. Thiemann, "JavaGI: Generalized Interfaces for Java," in *ECOOP 2007 Object-Oriented Programming*, ser. Lecture Notes in Computer Science, E. Ernst, Ed., vol. 4609. Springer Berlin Heidelberg, 2007, pp. 347–372.

[11] 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*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 89–99.

[12] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006, ch. Code Optimization.

[13] E. Ernst, "Family Polymorphism," in *Proceedings of the 15th European Conference on Object-Oriented Programming*, ser. ECOOP '01. London, UK, UK: Springer-Verlag, 2001, pp. 303–326.

[14] J. Järvi, J. Willcock, and A. 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*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 1–19.

[15] *The Boost Graph Library: User Guide and Reference Manual*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[16] J. G. Siek, "A Language for Generic Programming," Ph.D. dissertation, Indianapolis, IN, USA, 2005, aAI3183499.

[17] "`TreeSet[A]` Class," URL: http://www.scala-lang.org/api/current/#scala.collection.mutable.TreeSet.