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

Julia Belyakova[1]

1 **I. I. Vorovich Institute of Mathematics, Mechanics and Computer Science Southern Federal University, Rostov-on-Don, Russia**
`julbel@sfedu.ru`

Abstract

As was shown in earlier studies, mainstream object-oriented (OO) languages C# and Java provide weaker support for generic programming (GP) as compared with functional languages such as Haskell or SML. Having explored the instruments for GP in modern OO languages Scala, Ceylon, Kotlin, Rust, and Swift, we have found out that all of them, as well as C# and Java, stick to the same approach to constraining type parameters, which we called the "Constraints-are-Types" approach. It turns out that the approach has several inevitable limitations, and because of that Haskell type classes still provide better support for GP than modern object-oriented languages do. This research is aimed to discover a mechanism that would allow to radically improve the support for generic programming in OO languages. We identify the requirements that such mechanism should satisfy, present a design of the appropriate mechanism (concept parameters), and propose a sketch of its implementation for the C# language.

1998 ACM Subject Classification D.3.3 Language Constructs and Features, D.3.4 Processors

Keywords and phrases Generic programming, object-oriented languages, language extensions, concepts, constraints

1 Introduction

Generic programming (GP) [17] is a programming paradigm for writing highly reusable code. As opposed to concrete code, generic code is written in terms of *abstract* types and operations. In the C# example below, `Count<T>` is a generic function that depends on the *type parameter* `T`. It takes an array of elements of type `T` and a predicate `p` over `T`. The function returns the number of elements in the array `vs` that satisfy `p`.

```
static int Count<T>(T[] vs, Predicate<T> p)  // p : T -> Bool
{   int cnt = 0;
    foreach (var v in vs)
        if (p(v)) ++cnt;
    return cnt;             }
```

`Count<T>` can be *instantiated* with any type, i. e. `T` can be substituted with any concrete type. For instance, `Count<string>` is a valid instantiation of `Count<T>`.

Most of the modern programming languages provide language support for generic programming with various forms of parametric polymorphism [8, 7]. As was shown in earlier comparative studies [11, 10, 5, 18], some languages do it better than others. For example, Haskell is generally considered to be one of the best languages for generic programming [10, 5], whereas mainstream object-oriented (OO) languages, such as C# and Java, are much less expressive and have many drawbacks [14, 3]; some of them are outlined in Sec. 2. The main source of the differences between programming languages with regard to GP is a way in which *constraints* on type parameters are defined. Constraints limit possible values of type parameters that generic functions can be instantiated with. For instance, generic sort makes sense only for types that support comparison, such as `int` (but not functional types).

The ultimate goal of this research is to develop a mechanism for generic programming that would radically improve language support for GP in mainstream object-oriented languages. To achieve the goal, a number of tasks are to be solved. First of all, a design of language constructs for GP has to be developed. There must be no conflicts with existing object-oriented features, and the design should be expressive enough and type-safe. The second major point of the research is to suggest a possible way of implementation of the language extension proposed. Both the design and implementation of the C# extension for generic programming are sketched out in Sec. 4. A "toy" compiler for a subset of C# is to be implemented to provide a proof of concept. In future work we also plan to elaborate a mechanized proof of type safety of the similar extension for FJG [13] in Coq, but it is fully a matter of future work.

A reasonable question is whether such kind of research is relevant at all, because many new object-oriented languages have appeared in recent years, such as Kotlin and Swift. They probably have already improved the support for GP, haven't they? It turns out that the answer is no. Having made a comparative study of several modern OO languages [4], we have discovered that they still yield to Haskell with respect to language support for generic programming. We discuss the reasons in Sec. 3.

2 The Research Problem: Weak Support for Generic Programming in Mainstream Object-Oriented Languages

Language support for generic programming in C#/Java is provided by means of already existed object-oriented constructs — *interfaces* [15, 6]. Interfaces, which are normally used as *types* in object-oriented code, are reused to describe *constraints* on type parameters in generic code. A C# example is shown below. In order to provide ordering of `T`s in `Sort<T>`, the type parameter `T` is constrained to implement the interface `IComparable<T>`.

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

void Sort<T>(T[] vs) where T : IComparable<T>
{ ... if (vs[i].CompareTo(...) < 0) ... }
```

An incomplete list of drawbacks of the mechanisms for GP in C#/Java is enumerated below.

- *Lack of retroactive interface implementation.* After a type had been defined, it cannot implement any new interface. Consequently, generic code can be instantiated only with types that were *originally* designed to satisfy the constraints required.
- *The problem of multi-type constraints.* Because an interface can be used as a type, it describes properties of a *single* type, therefore, multi-type constraints (constraints on several types) cannot be expressed naturally. For example, consider a generic unification algorithm [16]: it takes a set of equations between terms (symbolic expressions), and returns the most general substitution that solves the equations. So the algorithm operates on three kinds of data: terms, equations, substitutions. A bunch of functions is also needed for its implementation. To express such constraints in C#, one has to define three separate interfaces for terms, equations, and substitution, and then separately constrain every type parameter of the algorithm with an appropriate interface:

  ```
  interface IEquation<Tm, Eqtn, Subst>
      where Tm : ITerm<Tm, Eqtn, Subst> ...
  ...
  Subst Unify<Tm, Eqtn, Subst> (IEnumerable<Eqtn> eqs)
  ```

```
    where Tm : ITerm<Tm, Eqtn, Subst>
    where Eqtn : IEquation<Tm, Eqtn, Subst>
    where Subst : ISubstitution<Tm, Eqtn, Subst>
```

- *The problem of multiple models.* We call "model" a way in which types satisfy a constraint. As long as a type cannot implement an interface in different ways, only unique models of constraints are allowed for types in C#/Java. But sometimes it is desirable to have several models of a constraint for the same set of types. For instance, one could imagine sets of strings with case-sensitive and case-insensitive equality comparisons, yet different orderings on numbers, different graph implementations, etc. One common solution to this problem is the use of the *Concept design pattern* [19]. The idea is as follows: instead of constraining type parameters, generic functions and classes take extra arguments that provide a required functionality called "concepts". For example, in the case of the `Sort<T>` algorithm, the constraint `T : IComparable<T>` is replaced with the function argument `cmp` of the type `IComparer<T>`:

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

void Sort<T>(T[] vs, IComparer<T> cmp)
{ ... if (cmp.Compare(vs[i], ...) < 0) ... }
```

  However, this pattern has two substantial pitfalls [29, 3]: first of all, it brings *run-time overhead*, because every object of a generic class with constraints has at least one extra field for the "concept", with constrained generic functions taking at least one extra argument. The second drawback, which we call *models inconsistency*, is less obvious but may lead to very subtle errors. Suppose we have `s1` of the type `HashSet<String>` and `s2` of the *same* type, provided that `s1` uses case-sensitive equality comparer, `s2` — the case-insensitive one. Thus, `s1` and `s2` use different, inconsistent models of comparison. Now consider the `GetUnion<T>` function:

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

  Unexpectedly, a result of `GetUnion(s1, s2)` could differ from `GetUnion(s2, s1)`. Despite the fact that `s1` and `s2` have the same type, they use *different comparers*, so the result of a function call depends on which comparer was chosen to build the union. Comparers are run-time objects, therefore, models consistency *cannot* be checked at *compile time*.
- *Lack of associated types* [14, 3, 20]. Types that are logically related to some entity are often called *associated types*. For instance, types of edges and vertices are associated types of a graph. There is no specific language support for associated types in C# and Java: such types are expressed in generic code in the form of extra type parameters.

3 "Constraints-are-Types" Disease of the Modern Object-Oriented Languages

Weak points of language facilities for generic programming in mainstream object-oriented languages are rather well known [11, 10, 5, 18, 3]. However, there is a question that seems to be not discussed yet: are these problems typical for OO languages in general or are they the problems of C# and Java only? To answer this question, we have explored several modern object-oriented languages: Scala, Ceylon, Kotlin, Rust, and Swift. It turned out that all of these languages, as well as C# and Java, followed the same approach to constraining type parameters, which we called the "Constraints-are-Types" approach. With this approach, interface-like constructs, which were normally used *as types* in object-oriented programming, are also reused *to constrain* type parameters. By "interface-like constructs" we mean here interfaces in C#, Java, Ceylon, and Kotlin; Scala traits, Swift protocols, and Rust traits.

The "Constraints-are-Types" approach is specific to object-oriented languages and has *inevitable limitations*: multi-type constraints and multiple models cannot be supported at language level, because interfaces and similar object-oriented constructs describe properties of a single type, with classes/structures implementing them in a unique way. By contrast, Haskell type classes follow the alternative approach to constraining type parameters, the "Constraints-are-Not-Types" one. With this approach, constructs defining constraints on types cannot act as types: they can be used *as constraints only*. Therefore, there is no fundamental difference between single-parameter and multi-parameter type classes, they describe constraints on types in an external way. Thus, one can easily write constraints on several types in one type class. For instance, recall the unification example. As compared with C#, in Haskell it can be written in much more elegant and succinct way:

```
class Unifiable tm eqtn subst where
  solve :: eqtn -> subst ...

unify :: Unifiable tm eqtn subst => [eqtn] -> subst
unify ...
```

As we have found out, other features important for generic programming, such as static interface members, associated types, and even retroactive modeling, which are available in Haskell but not in C# and Java, can be successfully incorporated into a programming language using any approach to constraining type parameters. Thus, for example, Scala, Rust, and Swift allow defining associated types; Rust and Swift support retroactive trait/protocol implementation. So, to summarise, there is only one set of mutually exclusive language features for GP that cannot be provided together:

1. the use of a construct both as a type and constraint;
2. language support for multi-type constraints and multiple models.

Using the "Constraints-are-Types" approach, the first one can be supported, but not the second; using the "Constraints-are-Not-Types" approach, vice versa.

Our point is that the second possibility is much more important for generic programming than the first one. It was shown in the study [12] that in practice interfaces used as constraints (such as `IComparable<T>` in C# or `Comparable<X>` in Java) are almost never used as types. Such interfaces are called "shapes". So, one can see that the first feature is not that useful for real programming. At the same time, lack of the language support for multi-type constraints and multiple models in OO languages leads to the use of the Concept pattern [19], which, in turn, has its own downsides. Hence, the "Constraints-are-Types" approach used in OO languages is rather a problem than a benefit. It sacrifices multi-type constraints and multiple models for the sake of literally nothing. That is why even young modern OO languages such as Rust and Swift still yield to older functional ones with respect to language support for GP.

4 How to Provide Better Support for Generic Programming in Object-Oriented Languages

Taking into account what was discussed in the previous sections, we suggest that to improve facilities for generic programming, a design of language constructs for GP in object-oriented languages should satisfy the following requirements:

1. The "Constraints-are-Not-Types" approach is to be used, i. e. in addition to interfaces/traits/protocols that can be used as types, a *new* language construct for *defining constraints* on type parameters has to be introduced into a language.

2. Multiple models have to be supported at language level, so that *models consistency* could be checked at *compile time*.
3. "Classic" features important for generic programming, such as retroactive modeling, associated types, constraint on associated types, and some others are to be preserved.
4. Safe interaction of new language constructs with object-oriented features such as inheritance and subtype polymorphism should be provided.

A common approach to evaluation of a design of language extension for generic programming is redeveloping some mature generic library in this extension. It might be the Boost Graph Library [1], for example, or a part of the FindBugs Java library. Another case study we found interesting is a development of generic library for data flow analysis [2].

After the design is developed, a problem of implementation arises. One obvious way to implement a language extension is a *translation to basic language*. To provide separate compilation and modularity, the translation has to be *reversible*. It means that one can compile code in the extended language into a module, and then use this module in a project also written in the extended language. Thus a compiler must be able to reconstruct source code in the extended language that the module was compiled from.

4.1 The Design of Concept Parameters for the C# Language

In this section we present a sketch of the mechanism for better support for GP in the C# language that satisfies the requirements outlined above. We call this mechanism *concept parameters*. C# was chosen as a target language, not Java, because .NET framework provides better instruments for implementing a reversible translation from the extended language with minimal run-time overhead. In particular, in contrast to Java byte code, .NET CIL (Common Intermediate Language) preserves information on type parameters of generics [15].

4.1.1 Concepts

Term "concept" was initially introduced in the documentation of the Standard Template Library (STL) [22] to describe requirements on template parameters in informal way. Concept as an explicit language construct for defining constraints on type parameters was proposed for C++ in 2003 [23]. Several designs have been developed since that time [24, 9, 25]; in the large, the expressive power of concepts[1] is rather close the Haskell type classes [5].

A *concept* describes a named set of requirements (or *constraints*) on one or more type parameters. Concepts support the following kinds of constraints:

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

A concept can *refine* one or more concepts. Refinement is a kind of a concept inheritance.

The design of concepts for C# we present is rather similar to the C++0x one. The major difference is support for subtype and supertype constraints. In the example below one can see the `Equality[T]` concept with function signatures `Equal` and `NotEqual` (the latter one has a default implementation); the `Ordering[T]` concept refines the `Equality[T]` one, i. e. it includes

[1] Concepts were designed to solve the problems of unconstrained C++ templates [22, 3]; they were expected to be included in C++0x standard, but this did not happen. A new version of concepts, Concepts Lite (C++1z) [26], is under way now.

all constraints from the refined concept; the `Unifying[]` concept is an example of multi-type concept. Note that the `IEnumerable<Eqtn>` interface is used as a type in the latter concept.

```
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); ... }
```

4.1.2 Models

A *model* defines a way in which types satisfy a concept. Models are defined retroactively, yet there can be several named models of a concept for the same set of types. One of these named models can be marked as a *default* one: it is then used to instantiate generic code if model is not specified at a call site. In such a way the support for multiple models is provided, with extra efforts not being required from a programmer in common cases. Default models are especially convenient when types have only one model of a concept. Because of support for multiple models, refining models should specify which models they refine. For example, `OrdStringCSAsc` refines the `EqStringCaseS` model in the example below.

```
model default EqStringCaseS for Equality[string] { ... }
model EqStringCaseIS for Equality[string]
{ bool Equal(string x, string y) { return x.ToLower() == y.ToLower(); }}

model default OrdStringCSAsc for Ordering[string]
    refines EqStringCaseS { ... }
```

4.1.3 Generic Code

The main novelty of the design proposed is that instead of the predicates `where T : ...` constraints on type parameters are expressed in the form of extra *concept parameters*. Here is an example of generic code with constraints:

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

Whereas `T` is the type parameter, `eq` is the concept parameter. Note that if only unique model of a constraint is allowed for the given set of types, constraints on type parameters are indeed predicates: types either satisfy the constraints or not. But in the case of multiple models it is important to know *how* concept constraints are satisfied. In contrast to the predicate-style, the parameter-style syntax clearly reflects this idea. Then even so-called *model generics* are defined in obvious way. For instance, in the following code the `Remove` method of the generic interface has no extra type parameters but takes the `eq` concept parameter.

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

As type parameters, concept parameters are parts of generic types. Hence, generic types instantiated with the same type arguments but different models are treated as different types, so the models consistency can be checked at compile time:

```
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
```

4.1.4 Notes

There are some type-theoretical issues regarding concept parameters to be studied. First of all, a unification algorithm of constraints has to be developed. Basic Hindley-Milner unification does not work here, because concepts can have subtype and supertype constraints. Another interesting thing is a *concept variance*. Consider the following definitions:

```
interface ISet<T | Equality[T] eq> { ... }
class B { ... }
class D : B { ... }
model EqB for Equality[B] { ... }
```

Should it be the case that `ISet<D, EqB>` is a legal instance? Under what conditions? It is also desirable to have the class `SortedSet<T|Ordering[T] ord>` implementing the interface `ISet<T|ord>`. Are there any problems here? This is a matter of future work. Now consider the following function:

```
void foo<T | Equality[T] eq>(ISet<T|eq> s) { ... }
...
ISet<string|EqStringCaseS> s1 = new SortedSet<string|OrdStringCSAsc>(..);
foo(s1);
```

Which model of `Equality[string]` should be used inside the `foo<>`? The static `EqStringCaseS` or the dynamic `OrdStringCSAsc` one?

4.2 A Sketch of Translation

An example of concepts and models translation to basic C# is shown below.

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

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> { ... }
```

In contrast to the Concept pattern where concept parameters are expressed in the form of extra class fields or function arguments, we suggest translating concept parameters to extra *type arguments* accompanied with appropriate subtype constraints. For example, the `eq` concept parameter of the `Contains<>` function is translated to the type parameter `eq` with the constraint `eq : Equality<T>` in a `where` section. Thus we can minimize run-time overhead on passing extra "concept" objects, and, furthermore, ensure the models consistency in basic language. Concepts are translated to interfaces, models are translated to classes that implement corresponding concept interfaces. As long as only one instance of every model is needed, a kind of the Singleton pattern can be used to access that model instance. Thus, for example, the `eq.Equal` call in the extended language (`Contains<>` function) is translated to the call `ConceptSingleton<eq>.Instance.Equal` in basic language. Associated types can be translated to extra type parameters. Somewhat tricky thing is a translation of default concept methods provided that multiple concept refinement is allowed; this part of the translation is under way. To make the translation actually reversible, translated code has to be annotated with attributes (they were omitted for simplicity).

5 Related Work

A number of language extensions for generic programming influenced by Haskell type classes have been already proposed for mainstream object-oriented languages [9, 21, 27, 29, 4] C++ and Java. As well as concept parameters for C# we present, all of them follow the "Constraints-are-Not-Types" approach, and provide the support for multi-type constraints and retroactive modeling. In addition, C++0x concepts [9, 25] provide a strong mechanism of concept-based overloading but do not support separate compilation. Concepts in language G [21], by contrast, limit the concept-based overloading to provide some modularity. Neither C++0x no G concepts allow *subtype/supertype constraints*, yet they do not interact with inheritance in any way. Generalized interfaces in JavaGI [27] and constraints in Genus [29] are both extensions for Java language. Unlike other extensions, they support concept variance and multiple dynamic dispatch (JavaGI only partially): these features have not been covered yet in this research. Genus also provides a mechanism for use-site genericity that generalizes the mechanism of Java wildcards. However, associated types and subtype/supertype constraints are allowed neither in JavaGI nor in Genus. The issue of static vs dynamic model resolution mentioned in Sec. 4.1.4 has not been studied yet in the earlier works.

If talk about OO languages and their extensions, multiple models are supported at language level in Genus only. However, constraints on type parameters are expressed here in the predicate-style, which misleads a programmer and makes it quite difficult to use the power of multiple models. But besides OO Genus, there is an extension for OCaml language, modular implicits [28], which also supports multiple models. For constraints on type parameters it uses the parameter-style syntax similar to the one suggested for concept parameters. Unlike concept parameters though, modular implicits are not aimed at object-oriented programming.

As well as for C# concept parameters, implementation via translation is proposed for G, JavaGI, and Genus. The main distinguishing feature of our translation is that it employs the facilities of .NET CIL to minimize run-time overhead and takes care of the *source code recovery*. G, JavaGI and Genus translation methods bring run-time overhead in a similar way as the Concept design pattern does. Yet the issues of modularity and reversible translation were not carefully discussed in the earlier studies.

Acknowledgements The authors would like to thank her advisor, Stanislav Mikhalkovich. The author is also grateful to Artem Pelenitsyn, Jeremy Siek, and Ross Tate for helpful discussions on generic programming.

References

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

2 Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*, chapter Code Optimization. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

3 Julia Belyakova and Stanislav Mikhalkovich. A Support for Generic Programming in the Modern Object-Oriented Languages. Part 1. An Analysis of the Problems. *Transactions of Scientific School of I.B. Simonenko. Issue 2*, (2):63–77 (in Russian), 2015.

4 Julia Belyakova and Stanislav Mikhalkovich. A Support for Generic Programming in the Modern Object-Oriented Languages. Part 2. A Review of the Modern Solutions. *Transactions of Scientific School of I.B. Simonenko. Issue 2*, (2):78–92 (in Russian), 2015.

5 Jean-Philippe Bernardy, Patrik Jansson, Marcin Zalewski, Sibylle Schupp, and Andreas Priesnitz. A Comparison of C++ Concepts and Haskell Type Classes. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, WGP '08, pages 37–48, New York, NY, USA, 2008. ACM.

6 Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 183–200, New York, NY, USA, 1998. ACM.

7 Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded Polymorphism for Object-oriented Programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 273–280, New York, NY, USA, 1989. ACM.

8 Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985.

9 Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ Concepts. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 295–308, New York, NY, USA, 2006. ACM.

10 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.

11 Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy G. Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. *SIGPLAN Not.*, 38(11):115–134, October 2003.

12 Ben Greenman, Fabian Muehlboeck, and Ross Tate. Getting F-bounded Polymorphism into Shape. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 89–99, New York, NY, USA, 2014. ACM.

13 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

14 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.

15 Andrew Kennedy and Don Syme. Design and Implementation of Generics for the .NET Common Language Runtime. *SIGPLAN Not.*, 36(5):1–12, May 2001.

16 Alberto Martelli and Ugo Montanari. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, April 1982.

17 David R. Musser and Alexander A. Stepanov. Generic Programming. In *Proceedings of the International Symposium ISSAC'88 on Symbolic and Algebraic Computation*, ISAAC '88, pages 13–25, London, UK, UK, 1989. Springer-Verlag.

18 Bruno C.d.S. Oliveira and Jeremy Gibbons. Scala for Generic Programmers: Comparing Haskell and Scala Support for Generic Programming. *J. Funct. Program.*, 20(3-4):303–352, July 2010.

19 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.

20 A. Pelenitsyn. Associated Types and Constraint Propagation for Generic Programming in Scala. *Programming and Computer Software*, 41(4):224–230, 2015.

21 Jeremy G. Siek and Andrew Lumsdaine. A language for generic programming in the large. *Sci. Comput. Program.*, 76(5):423–465, May 2011.

22 Alexander A. Stepanov and Meng Lee. The Standard Template Library. Technical Report 95-11(R.1), HP Laboratories, November 1995.

23 Bjarne Stroustrup. Concept Checking — A More Abstract Complement to Type Checking. Technical Report N1510=03-0093, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, October 2003.

24 Bjarne Stroustrup and Gabriel Dos Reis. Concepts — Design Choices for Template Argument Checking. Technical Report N1522=03-0105, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, October 2003.

25 Bjarne Stroustrup and Andrew Sutton. A Concept Design for the STL. Technical Report N3351=12-0041, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, January 2012.

26 Andrew Sutton. C++ Extensions for Concepts PDTS. Technical Specification N4377, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, February 2015.

27 Stefan Wehr and Peter Thiemann. JavaGI: The Interaction of Type Classes with Interfaces and Inheritance. *ACM Trans. Program. Lang. Syst.*, 33(4):12:1–12:83, July 2011.

28 L. White, F. Bour, and J. Yallop. Modular Implicits. *ArXiv e-prints*, December 2015.

29 Yizhou Zhang, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, and Andrew C. Myers. Lightweight, Flexible Object-oriented Generics. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 436–445, New York, NY, USA, 2015. ACM.