Comparative Study of Generic Programming Features in Object-Oriented Languages[1]

Julia Belyakova
julbinb@gmail.com

February 3rd 2017

NEU CCIS
Programming Language Seminar

---

[1] Based on the papers [Belyakova 2016b; Belyakova 2016a]

Generic Programming

A term “Generic Programming” (GP) was coined in 1989 by Alexander Stepanov and David Musser Musser and Stepanov 1989.

Idea

Code is written in terms of abstract types and operations (parametric polymorphism).

Purpose

Writing highly reusable code.

Contents

1. Language Support for Generic Programming
2. Peculiarities of Language Support for GP in OO Languages
3. Language Extensions for GP in Object-Oriented Languages
4. Conclusion

1. Language Support for Generic Programming
   - Unconstrained Generic Code
   - Constraints on Type Parameters
2. Peculiarities of Language Support for GP in OO Languages
3. Language Extensions for GP in Object-Oriented Languages
4. Conclusion

An Example of Unconstrained Generic Code (C#)

```
// parametric polymorphism: T is a type parameter
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;
}
```

An Example of Unconstrained Generic Code (C#)

```
// parametric polymorphism: T is a type parameter
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

```
int[] ints = new int[]{ 3, 2, -8, 61, 12 };
var  evCnt = Count(ints, x => x % 2 == 0);           // T == int

string[] strs = new string[]{ "hi", "bye", "hello", "stop" };
var  evLenCnt = Count(strs, x => x.Length % 2 == 0); // T == string
```

We Need More Genericity!

Look again at the `vs` parameter:

```
static int Count<T>(T[] vs, Predicate<T> p)
{ ... }                         // p : T -> Bool

int[] ints = ...
var  evCnt = Count(ints, ...

string[] strs = ...
var  evLenCnt = Count(strs, ...
```

We Need More Genericity!

Look again at the `vs` parameter:

```
static int Count<T>(T[] vs, Predicate<T> p)
{ ... }                         // p : T -> Bool

int[] ints = ...
var   evCnt = Count(ints, ...

string[] strs = ...
var   evLenCnt = Count(strs, ...
```

The Problem

Generic `Count<T>` function is not generic enough.
It works with arrays only.

True C# Code for the Count Function

Solution: using `IEnumerable<T>` interface instead of array.

```
// provides iteration over the elements of type T
interface IEnumerable<T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();  ...
}

static int Count<T>(IEnumerable<T> vs, Predicate<T> p)
{
    int cnt = 0;
    foreach (var v in vs) ...
```

```
var   ints = new int[]{ 3, 2, -8, 61, 12 };           // array
var  evCnt = Count(ints, x => x % 2 == 0);

var intSet = new SortedSet<int>{ 3, 2, -8, 61, 12 };  // set
var evSCnt = Count(intSet, x => x % 2 == 0);
```

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

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

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

Figure: The first attempt: fail

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

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

Figure: The first attempt: fail

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#)

```
// provides comparison with T
interface IComparable<T> { int CompareTo(T other); }

static T FindMax<T>(IEnumerable<T> vs)
    where T : IComparable<T>                 // F-bounded polymorphism
{
    T mx = vs.First();
    foreach (var v in vs)
        if (mx.CompareTo(v) < 0) mx = v;
    return mx;
}
```

An Example of Generic Code with Constraints (C#)

```
// provides comparison with T
interface IComparable<T> { int CompareTo(T other); }

static T FindMax<T>(IEnumerable<T> vs)
    where T : IComparable<T>             // F-bounded polymorphism
{
    T mx = vs.First();
    foreach (var v in vs)
        if (mx.CompareTo(v) < 0) mx = v;
    return mx;
}
```

`FindMax<T>` can only be instantiated with types implementing the `IComparable<T>` interface

```
var ints = new int[]{ 3, 2, -8, 61, 12 };
var iMax = FindMax(ints);   // 61
var strs = new LinkedList<string>{ "hi", "bye", "stop", "hello" };
var sMax = FindMax(strs);   // "stop"
```

Explicit Constraints on Type Parameters

Programming languages provide various language mechanisms for generic programming based on **explicit constraints**[2], e.g.:

- Haskell: type classes;
- SML, OCaml: modules;
- Rust: traits;
- Scala: traits & subtyping[3];
- Swift: protocols & subtyping;
- Ceylon, Kotlin, C#, Java: interfaces & subtyping;
- Eiffel: subtyping.

---

[2] By contrast, C++ templates are *un*constrained.

[3] Constraints of the form `T : C`, where C is a class.

1 Language Support for Generic Programming

2 Peculiarities of Language Support for GP in OO Languages

- Pitfalls of C#/Java Generics
- The “Constraints-are-Types” Approach

3 Language Extensions for GP in Object-Oriented Languages

4 Conclusion

Some Deficiencies of GP in C#/Java I

It was shown in [Garcia et al. 2003; Garcia et al. 2007] that C# and Java provide weaker language support for generic programming as compared with languages such as Haskell or SML

- Lack of support for retroactive modeling: class cannot implement interface once the class has been defined. (Not to be confused with *extension methods* in C#, Kotlin.)

```
interface IWeighed { double GetWeight(); }
static double MaxWeight<T>(T[] vs)
    where T : IWeighed { ... }

class Foo { ... double GetWeight(); }
MaxWeight<Foo>(...) // ERROR: Foo does not implement IWeighed
```

Some Deficiencies of GP in C#/Java II

- Lack of support for associated types and constraints propagation.

```
interface IEdge<Vertex> { ... }
interface IGraph<Edge, Vertex> where Edge : IEdge<Vertex>{ ... }

... BFS<Graph, Edge, Vertex>(Graph g, Predicate<Vertex> p)
    where Edge : IEdge<Vertex>
    where Graph : IGraph<Edge, Vertex> { ... }
```

- Lack of default method's implementation.

```
interface IEquatable<T>
{
    bool Equal(T other);
    bool NotEqual(T other); // { return !this.Equal(other); }
}
```

Some Deficiencies of GP in C#/Java III

- Binary method problem: how to express requirement for binary method `binop(T, T)`?

```
// T only pretends to be an "actual type" of the interface
interface IComparable<T> { int CompareTo(T other); }

class A { ... }
// provides non-symmetric comparison B.CompareTo(A)
class B : IComparable<A> { ... }

// requires symmetric comparison T.CompareTo(T)
static T FindMax<T>(...)
    where T : IComparable<T> { ... }
```

- Lack of static methods.
- Lack of support for multiple models (q.v. slide 17).
- Lack of support for multi-type constraints (q.v. slide 17).

Real World C# Example: Iterative Algorithm (DFA)

Iterative algorithm can be implemented in C# in a generic manner.

```
public abstract class IterativeAlgorithm<
                          BasicBlock, // CFG Basic Block
                          V,          // Vertex Type
                          G,          // Control Flow Graph
                          Data,       // Analysis Data
                          TF,         // Transfer Function
                          TFInitData> // Data to Initialize TF
    where V : IVertex<V, BasicBlock>
    where G : IGraph< V, BasicBlock>
    where Data : ISemilattice<Data>, class
    where TF : ITransferFunction<Data, BasicBlock, TFInitData>, new()
{ ...
    public IterativeAlgorithm(G graph) { ... }
    protected abstract void Initialize();
    public void Execute() { ... }
}
```

Figure: Signature of the iterative algorithm executor's class

OO Languages Chosen for the Study

Apart from C# and Java, the following object-oriented languages were explored in our study:

- Scala (2004, Dec 2016)[4];
- Rust (2010, Dec 2016);
- Ceylon (2011, Nov 2016);
- Kotlin (2011, Dec 2016);
- Swift (2014, Dec 2016).

---

[4] <name> (<first appeared>, <recent stable release>)

The State of The Art

Some of the C#/Java problems are eliminated in the modern OO languages.

- default method's implementation: Java 8, Scala, Ceylon, Swift, Rust.
- static methods: Java 8, Ceylon, Swift, Rust.
- self types[5]: Ceylon, Swift, Rust.
- associated types: Scala, Swift, Rust.
- retroactive modeling: Swift, Rust.

---

[5] Neatly solve binary method problem.

Constraints as Types

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

The “Constraints-are-Types” Approach

Interface-like language constructs are used in 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:

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

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

Inevitable Limitations of the OO approach

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

Inevitable Limitations of the OO approach

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

- Multi-type constraints cannot be expressed naturally. Instead of

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

Inevitable Limitations of the OO approach

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

- Multi-type constraints cannot be expressed naturally. Instead of

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

  we have:

  ```
  interface IConstraintA<A, B>  where A : IConstraintA<A, B>
                                where B : IConstraintB<A, B> {...}
  interface IConstraintB<A, B>  where A : IConstraintA<A, B>
                                where B : IConstraintB<A, B> {...}
  double Foo<A, B>(A[] xs)
      where A : IConstraintA<A, B>
      where B : IConstraintB<A, B> {...}
  ```

Inevitable Limitations of the OO approach

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

- Multi-type constraints cannot be expressed naturally. Instead of

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

  we have:

  ```
  interface IConstraintA<A, B>  where A : IConstraintA<A, B>
                                where B : IConstraintB<A, B> {...}
  interface IConstraintB<A, B>  where A : IConstraintA<A, B>
                                where B : IConstraintB<A, B> {...}
  double Foo<A, B>(A[] xs)
      where A : IConstraintA<A, B>
      where B : IConstraintB<A, B> {...}
  ```

- Multiple models cannot be supported at language level.

Concept Pattern

With the Concept pattern[6] [Oliveira, Moors, and Odersky 2010], constraints on type parameters are replaced with extra function arguments/class fields — "concepts".

F-Bounded Polymorphism

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

static T FindMax<T>(
  IEnumerable<T> vs)
  where T : IComparable<T>     // *
{
  T mx = vs.First();
  foreach (var v in vs)
    if (mx.CompareTo(v) < 0)   // *
      ...
```

Concept Pattern

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

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

---

[6] Concept pattern ≈ Strategy design pattern

Advantages of the Concept Pattern

Both limitations of the "Constraints-are-Types" approach are eliminated with this design pattern.

1. multi-type constraints are multi-type "concept" arguments;

```
interface IConstraintAB<A, B>
{ B[] Bar(A a); ... }

double Foo<A, B>(A[] xs, IConstraintAB<A, B> c)
{ ... c.Bar(...) ... }
```

2. multiple "models" are allowed as long as several classes can implement the same interface.

```
class IntCmpDesc  : IComparer<int> { ... }
class IntCmpMod42 : IComparer<int> { ... }

var ints = new int[]{ 3, 2, -8, 61, 12 };

var   minInt = FindMax(ints, new IntCmpDesc());
var maxMod42 = FindMax(ints, new IntCmpMod42());
```

Drawbacks of the Concept Pattern I

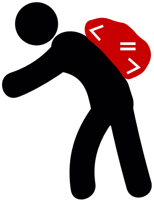
The Concept design pattern is **widely used** in standard generic libraries of C#, Java, and Scala, but it has several **problems**.

Possible runtime overhead

Extra class fields or function arguments.

```
interface IComparer<T>
{ ... }

class SortedSet<T> : ...
{
    IComparer<T> Comparer;
    ...
}
```



Drawbacks of the Concept Pattern II

The Concept design pattern is **widely used** in standard generic libraries of C#, Java, and Scala, but it has several **problems**.

Models inconsistency

Objects of the same type can use different models (at runtime).

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

Attention!

`GetUnion(s1, s2)` could differ from `GetUnion(s2, s1)`!

Type-Safe Concept Pattern

It is possible to guarantee models consistency in basic C#
if express “concept” as type parameter:

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

class SafeSortedSet<T, CmpT>
    where CmpT : IComparer<T>, struct   // CmpT is "concept parameter"
{   ...
        CmpT cmp = default(CmpT); ...
        if (cmp.Compare(a, b) < 0) ...  }

struct IntCmpDesc : IComparer<int> {...} ...

var ints = new int[]{ 3, 2, -8, 61, 12 };
var s1 = new SafeSortedSet<int,  IntCmpDesc>(ints);
var s2 = new SafeSortedSet<int, IntCmpMod42>(ints);
s1.UnionWith(s2);  // ERROR: s1 and s2 have different types
```

See «Classes for the Masses» at ML Workshop, ICFP 2016:
prototype implementation for Concept C#.

1 Language Support for Generic Programming

2 Peculiarities of Language Support for GP in OO Languages

3 Language Extensions for GP in Object-Oriented Languages
- The “Constraints-are-Not-Types” Approach

4 Conclusion

Alternative Approach

Several language extensions for GP inspired by Haskell type classes [Wadler and Blott 1989] were designed:

- C++ concepts (2003–2014) [Dos Reis and Stroustrup 2006; Gregor et al. 2006] and concepts in language G (2005–2011) [Siek and Lumsdaine 2011];
- Generalized interfaces in JavaGI (2007–2011) [Wehr and Thiemann 2011];
- Concepts for C# [Belyakova and Mikhalkovich 2015];
- Constraints in Java Genus [Zhang et al. 2015].

The "Constraints-are-Not-Types" Approach

To constrain type parameters, a separate language construct is provided. It cannot be used as type.

Constraints in Java Genus I

```
interface Iterable[T] { ... }

constraint Eq[T]                         { boolean T.equals(T other); }
// constraint's "inheritance"
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) ...   }

constraint Baz[A, B] { ... }
static double Foo[A, B]( ... ) where Baz[A, B] { ... }
```

As constraints are external to types, Java Genus supports:

- static methods;
- retroactive modeling;
- multi-type constraints.

Constraints in Java Genus II

Several models are allowed, and models consistency is guaranteed at the types level.

```
interface Set[T where Eq[T]]    {...}

model StringCIEq for Eq[String] {...} // case-insensitive equality model
model StringFLEq for Eq[String] {...} // equality on first letter

// case-sensitive natural model is used by default
Set[String] s1 = ...;

Set[String with StringCIEq] s2 = ...;
Set[String with StringFLEq] s3 = ...;

s1 = s2;           // Static ERROR, s1 and s2 have different types
s2.UnionWith(s3); // Static ERROR, s2 and s3 have different types
```

1. Language Support for Generic Programming

2. Peculiarities of Language Support for GP in OO Languages

3. Language Extensions for GP in Object-Oriented Languages

4. Conclusion
    - Can we prefer one approach to another?
    - Subtype constraints

Which Approach is Better?

“Constraints-are-Types”

“Constraints-are-Not-Types”

Lack of language support for multi-type constraints and multiple models.

Language support for multi-type constraints and multiple models.

Constraints can be used as types.

Constraints cannot be used as types.

Concept Pattern

Runtime flexibility.

Models inconsistency and possible overhead.

"Constraints-are-Not-Types" Is Preferable

In practice, interfaces that are used as constraints, are almost never used as types

According to [Greenman, Muehlboeck, and Tate 2014] ("Material-Shape Separation"[7]):

- 1 counterexample in 13.5 million lines of open-source generic-Java code;
- 1 counterexample in committed Ceylon code;
- counterexamples are similar and can be written.

---

[7] shapes are constraints, materials are types

None of the “Constraints-are-Not-Types” extensions support subtype constraints, although they still can be useful (not only for backward compatibility).

```
concept CFG[G] { type B; /* basic block */ ... }

concept TransferFunc[TF] { ... }

abstract class TFBuilder<TF, G | CFG[G] cfg>
{ abstract void Init(G g);   abstract TF Build (cfg.B bb); }

concept BuildableTF[TF] extends TransferFunc[TF]
{ type G;  CFG[G] cfg;  type Builder : TFBuilder<TF,G,cfg>, new(); }

class IterAlgo<G,..., TF,... | CFG[G] cfg, BuildableTF[TF] btf,...>
{ ...
    btf.Builder bld = new btf.Builder();
    bld.Init(g);
    foreach (cfg.B bb in cfg.Blocks(g))
        tfs[bb] = bld.Build(bb);           ...                     }
```

None of the “Constraints-are-Not-Types” extensions support subtype constraints, although they still can be useful (not only for backward compatibility).

```
concept CFG[G] { type B; /* basic block */ ... }

concept TransferFunc[TF] { ... }

abstract class TFBuilder<TF, G | CFG[G] cfg>
{ abstract void Init(G g);   abstract TF Build (cfg.B bb); }

concept BuildableTF[TF] extends TransferFunc[TF]
{ type G;  CFG[G] cfg;  type Builder : TFBuilder<TF,G,cfg>, new(); }

class IterAlgo<G,..., TF,... | CFG[G] cfg, BuildableTF[TF] btf,...>
{ ...
    btf.Builder bld = new btf.Builder();
    bld.Init(g);
    foreach (cfg.B bb in cfg.Blocks(g))
        tfs[bb] = bld.Build(bb);           ...                     }
```

Research Problem

How to combine the approach with **subtype constraints** on types?

Open Design Questions

- Model's reuse for subclasses.

```
class Foo<T | Bar[T] b> { ... }

class B { ... }       class D : B { ... }

model BarB for Bar[B] { ... }
```

Under what conditions `Foo<D | BarB>` is allowed (sound)?

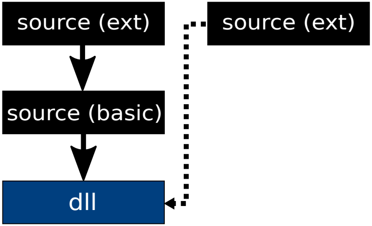
- Static/dynamic binding of concept parameters.

```
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 `foo<>`?
Static `EqStringCaseS` or dynamic `OrdStringCSAsc`?

Implementation Challenges

- Efficient type checking (conventional unification of equalities is not enough).
- Support for separate compilation and modularity (if the extension is implemented via translation to basic language).


source (ext)

source (basic)

dll

source (ext)

References I

J. Belyakova. “Language Support for Generic Programming in Object-Oriented Languages: Peculiarities, Drawbacks, Ways of Improvement”. In: *Programming Languages: 20th Brazilian Symposium, SBLP 2016, Maringá, Brazil, September 22-23, 2016, Proceedings*. Ed. by Castor, Fernando and Liu, Yu David. Cham: Springer International Publishing, 2016, pp. 1–15.

J. Belyakova. “Language Support for Generic Programming in Object-Oriented Languages: Design Challenges”. In: *Proceedings of the Institute for System Programming* 28.2 (2016), pp. 5–32.

D. Musser and A. Stepanov. “Generic programming”. English. In: *Symbolic and Algebraic Computation*. Ed. by P. Gianni. Vol. 358. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1989, pp. 13–25.

R. Garcia et al. “A Comparative Study of Language Support for Generic Programming”. In: *SIGPLAN Not.* 38.11 (Oct. 2003), pp. 115–134.

References II

R. Garcia et al. "An Extended Comparative Study of Language Support for Generic Programming". In: *J. Funct. Program.* 17.2 (Mar. 2007), pp. 145–205.

B. C. Oliveira, A. Moors, and M. Odersky. "Type Classes As Objects and Implicits". In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '10. Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 341–360.

P. Wadler and S. Blott. "How to Make Ad-hoc Polymorphism Less Ad Hoc". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: ACM, 1989, pp. 60–76.

G. Dos Reis and B. Stroustrup. "Specifying C++ Concepts". In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '06. Charleston, South Carolina, USA: ACM, 2006, pp. 295–308.

References III

D. Gregor et al. “Concepts: Linguistic Support for Generic Programming in C++”. In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications.* OOPSLA ’06. Portland, Oregon, USA: ACM, 2006, pp. 291–310.

J. G. Siek and A. Lumsdaine. “A Language for Generic Programming in the Large”. In: *Sci. Comput. Program.* 76.5 (May 2011), pp. 423–465.

S. Wehr and P. Thiemann. “JavaGI: The Interaction of Type Classes with Interfaces and Inheritance”. In: *ACM Trans. Program. Lang. Syst.* 33.4 (July 2011), 12:1–12:83.

J. Belyakova and S. Mikhalkovich. “Pitfalls of C# Generics and Their Solution Using Concepts”. In: *Proceedings of the Institute for System Programming* 27.3 (June 2015), pp. 29–45.

References IV

Y. Zhang et al. "Lightweight, Flexible Object-oriented Generics". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2015. Portland, OR, USA: ACM, 2015, pp. 436–445.

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*. PLDI '14. Edinburgh, United Kingdom: ACM, 2014, pp. 89–99.

L. White, F. Bour, and J. Yallop. "Modular Implicits". In: *ArXiv e-prints* (Dec. 2015). arXiv: `1512.01895 [cs.PL]`.

Scala's Modular Roots. Available here.

Comparison of Languages and Extensions

Language Support for GP in OO Languages	Haskell	C#	Java 8	Scala	Ceylon	Kotlin	Rust	Swift	JavaGI	G	C#^{cpt}	Genus	ModImpl
Constraints can be used as types	○	●	●	●	●	●	●	●	◐	○	○	○	○
Explicit self types	—	○	○	◐	●$^-$	○	●	●	◐	—	—	—	—
Multi-type constraints	●	⚹	⚹	⚹	○	⚹	○	○	●	●	●	●	●
Retroactive type extension	—	●	○	○	○	●	●	●	○	○	○	○	—
Retroactive modeling	●	⚹	⚹	⚹	○	⚹	●	●	●	●	●	●	●
Type conditional models	●	○	○	○	○	○	●	○	●	●	●	●	●
Static methods	●	○	●	○	●	●	●	●	●	●	●	●	●
Default method implementation	●	○	●	●	●	●	●	●	◐	●	●	○	○
Associated types	●	○	○	●	○	○	●	●	○	●	●	○	●
Constraints on associated types	◐	—	—	●	—	—	●	●	—	●	●	—	●
Same-type constraints	◐	—	—	●	—	—	●	●	—	●	●	—	●
Concept-based overloading	○	○	○	○	○	○	●	○	○	◐	○	○	○
Multiple models	○	⚹	⚹	⚹	⚹	⚹	○	○	○	◐[a]	●	●	●
Models consistency (model-dependent types)	—[b]	○	○	○	○	○	—[b]	—[b]	—[b]	—[b]	●	●	●
Model genericity	—	⚹	⚹	⚹	⚹	⚹	●	○	○	○	○	●	—

⚹ means support via the Concept pattern. [a] G supports lexically-scoped models but not really multiple models.
[b] If multiple models are not supported, the notion of model-dependent types does not make sense.

Dependent Types

```
-- natural number
data Nat       --  Nat : Type
  = Zero       -- Zero : Nat
  | Succ Nat   -- Succ : Nat -> Nat

-- generic list
data List a          -- List : Type -> Type
  = []               --    [] : List a
  | (::) a (List a) -- (::) : a -> List a -> List a

-- vector of the length k (dependent type)
data Vect : Nat -> Type -> Type where
  Nil  : Vect Zero a
  Cons : a -> Vect k a -> Vect (Succ k) a
```

Figure: Data types and dependent types in Idris

Dependent Types

```
-- natural number
data Nat        --  Nat : Type
  = Zero        -- Zero : Nat
  | Succ Nat    -- Succ : Nat -> Nat

-- generic list
data List a           -- List : Type -> Type
  = []                --    [] : List a
  | (::) a (List a) -- (::) : a -> List a -> List a

-- vector of the length k (dependent type)
data Vect : Nat -> Type -> Type where
  Nil  : Vect Zero a
  Cons : a -> Vect k a -> Vect (Succ k) a
```

Figure: Data types and dependent types in Idris

If we had dependent types in OO languages, we would also have models consistency (a comparer could be a part of the type).

Concept Parameters versus Concept Predicates

When multiple models are supported, constraints on type parameters are *not predicates* any more, they are compile-time parameters [White, Bour, and Yallop 2015] (just as types are parameters of generic code).

Concept Predicates

```
// model-generic methods
interface List[T] { ...
  boolean remove(T x) where Eq[T];
}
List[int] xs = ...
xs.remove[with StringCIEq](5);

// model-generic types
interface Set[T where Eq[T]] {...}
Set[String] s1 = ...;
Set[String with StringCIEq] s2=...;
```

Concept Parameters

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

// model-generic types
interface Set<T ! Eq[T] eq>  {...}
Set<String> s1 = ...;
Set<String ! StringCIEq> s2 = ...;
```

More examples of Concept Parameters I

```
(* equality *)
module type Eq = sig
  type t val
  equal : t -> t -> bool
end

(* foo:  {Eq with t = ’a list} ->
   ’a list -> ’a list -> ’a list *)
let foo  {EL : Eq} xs ys =
  if EL.equal(xs, ys)
  then xs else xs @ ys

(* foo’: {Eq with t = ’a} ->
   ’a list -> ’a list -> ’a list *)
let foo’ {E : Eq}  xs ys =
  if (Eq_list E).equal(xs, ys)
  then xs else xs @ ys
```

```
(* equality of ints *)
implicit module Eq_int =
struct
  type t = int
  let equal x y  = ...
end

(* natural equality of lists *)
implicit module Eq_ls {E : Eq} =
struct
  type t = Eq.t list
  let equal xs ys = ...
end

let x=foo [1] [4;5](*Eq_ls Eq_int*)

let y=foo’ [1] [4;5] (* Eq_int *)
```

Figure: OCaml modular implicits White, Bour, and Yallop 2015

More examples of Concept Parameters II

Concept Predicates

```
// constraints depend
// on type parameters
constraint WeighedGraph[V, E, W]
{ ... }

Map[V, W] SSSP[V, E, W](V s)
where WeighedGraph[V, E, W]
{ ... }

...pX = SSSP[MyV, MyE, Double
        with MyWeighedGraph](x);
```

Concept Parameters

```
// types can be taken
// from constraints
constraint WeighedGraph[V, E, W]
{ ... }

Map<V, W> SSSP<V, E, W
! WeighedGraph[V, E, W] wg>(V s)
{ ... }

...pX = SSSP<? ! MyWeighedGraph>
        (x);
```

Some Deficiencies of GP in C#/Java

Lack of static methods.

```
interface IMonoid<T>
{
    T BinOp(T other);
    T Ident(); // ???
}
static T Accumulate<T>(IEnumerable<T> vs) where T : IMonoid<T>
{
    T result = ???;     // Ident
    foreach (T val in values)
        result = result.BinOp(val);
    return result;   }
```

Haskell Type Classes

```
class Eq a where
  (==) :: a -> a -> Bool ...      -- type class (concept) for equality

class Eq a => Ord a where         -- type class for ordering
  compare :: a -> a -> Ordering
  (<=) :: a -> a -> Bool ...

instance Ord Int where            -- instance (model)
  ...    -- Ord functions’ implementation
```

Figure: Examples of Haskell type classes

```
findMax :: Ord a => [a] -> a      -- ‘‘a’’ is constrained with Ord
...
findMax (x:xs) = ... if mx < x ...
```

Figure: The use of the `Ord` type class

Haskell Type Classes

```
class Eq a where
  (==) :: a -> a -> Bool ...        -- type class (concept) for equality

class Eq a => Ord a where           -- type class for ordering
  compare :: a -> a -> Ordering
  (<=) :: a -> a -> Bool ...

instance Ord Int where              -- instance (model)
  ...    -- Ord functions' implementation
```

Figure: Examples of Haskell type classes

```
findMax :: Ord a => [a] -> a        -- ``a'' is constrained with Ord
...
findMax (x:xs) = ... if mx < x ...
```

Figure: The use of the `Ord` type class

Multi-parameter type classes are supported

Only unique instance of the type class is allowed

Generic Code Examples in Rust and Swift

```
trait Eqtbl {
  fn equal(&self, that: &Self) -> bool;
  fn not_equal(&self, that: &Self) -> bool { !self.equal(that) }
}
impl Eqtbl for i32
{ fn equal (&self, that: &i32) -> bool { *self == *that } }
```

Figure: GP in Rust: self types, default method's implementation, retroactive modeling

```
protocol  Equatable { func equal(that: Self) -> Bool; }
extension Equatable
{ func notEqual(that: Self) -> Bool { return !self.equal(that) } }
extension Foo : Equatable { ... }

protocol Container { associatedtype ItemTy ... }
func allItemsMatch<C1: Container, C2: Container where
    C1.ItemTy == C2.ItemTy, C1.ItemTy: Equatable> ...
```

Figure: GP in Swift: self types, default method's implementation, retroactive modeling, associated types

Constrained Generic Code in Scala

Traits are used in Scala instead of interfaces.

```
trait Iterable[A] {
  def iterator: Iterator[A]
  def foreach ...
}

trait Ordered[A] {
  abstract def compare (that: A): Int
  def < (that: A): Boolean ...
}
```

Figure: `Iterable[A]` and `Ordered[A]` traits (Scala)

Constrained Generic Code in Scala

Traits are used in Scala instead of interfaces.

```
trait Iterable[A] {
  def iterator: Iterator[A]
  def foreach ...
}

trait Ordered[A] {
  abstract def compare (that: A): Int
  def < (that: A): Boolean ...
}
```

Figure: `Iterable[A]` and `Ordered[A]` traits (Scala)

```
def findMax[A <: Ordered[A]] (vs: Iterable[A]): A {
  ...
  if (mx < v) ...
}
```

Figure: Extract from the `findMax[A]` function

Concept Pattern in Scala

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

F-Bounded Polymorphism

```
trait Ordered[A] {
  abstract def compare
             (that: A): Int
  def < (that: A): Boolean = ...
}

// upper bound
def findMax[A <: Ordered[A]]
           (vs: Iterable[A]): A
{ ... }
```

Concept Pattern

```
trait Ordering[A] {
  abstract def compare
               (x: A, y: A): Int
  def lt(x: A, y: A): Boolean = ...
}

// context bound (syntactic sugar)
def findMax[A : Ordering]
           (vs: Iterable[A]): A
{ ... }

// implicit argument (real code)
def findMax(vs: Iterable[A])
    (implicit ord: Ordering[A])
{ ... }
```

Scala Path-Dependent Types [*Scala's Modular Roots*]

```
trait Ordering
{ type T;          def compare(x: T, y: T): Int }

object IntOrdering extends Ordering
{ type T = Int;    def compare(x: T, y: T): Int = x - y }

trait SetSig
{ type Elem; type Set
  def empty: Set
  def member(e: Elem, s: Set): Boolean ... }

abstract class UnbalancedSet extends SetSig
{ val Element: Ordering;   type Elem = Element.T
  sealed trait Set;        case object Leaf extends Set
  case class Branch(left: Set, elem: Elem, right: Set) extends Set
  val empty = Leaf
  def member(x: Elem, s: Set): Boolean = ... } ... }

object S1 extends UnbalancedSet { val Element: Ordering = IntOrdering }
object S2 extends UnbalancedSet { val Element: Ordering = IntOrdering }

var set1 = S1.insert(1, S1.empty); var set2 = S2.insert(2, S2.empty);
S1.member(2, set2) // ERROR
```