Comparative Study of Generic Programming Features in Object-Oriented Languages

Julia Belyakova
julbinb@gmail.com

February 3rd 2017
NEU CCIS
Programming Language Seminar

1Based on the papers [Belyakova 2016b; Belyakova 2016a]
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
// parametric polymorphism: T is a type parameter
static int Count<T>(T[] vs, Predicate<T> p)
{
  int cnt = 0;
  foreach (var v in vs)
  {
    if (p(v)) ++cnt;
  }
  return cnt;
}
// parametric polymorphism: T is a type parameter

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

Count<T> can be instantiated with any type

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

```csharp
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:

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

```c
int[] ints = ...
var evCnt = Count(ints, ...)
```

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

The Problem

Generic `Count<T>` function is not generic enough. It works with **arrays only**.
Solution: using `IEnumerable<T>` interface instead of array.

```csharp
// provides iteration over the elements of type T
text interface IEnumerable<T> : IEnumerable
{
    IEnumerable<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?

```csharp
// 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?

```csharp
// max element in vs
static T FindMax<T>(IEnumerable<T> vs)
{
    T mx = vs.First();
    foreach (var v in vs)
    {
        // ERROR: operator ‘<’
        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#)

```csharp
// 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"
```
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
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.)

```java
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.

```csharp
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.

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

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

- Scala (2004, Dec 2016);
- Rust (2010, Dec 2016);
- Ceylon (2011, Nov 2016);
- Kotlin (2011, Dec 2016);
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**: 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:

```csharp
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

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

we have:

```java
interface 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

```csharp
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

```csharp
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 \(\approx\) 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;

```java
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.

```java
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());
```
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.

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

class SortedSet<T> : ...
{
    IComparer<T> Comparer;
    ...
}
```
The Concept design pattern is \textbf{widely used} in standard generic libraries of C#, Java, and Scala, but it has several \textbf{problems}.

\textbf{Models inconsistency}

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

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

\textbf{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:

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

class SafeSortedSet<T, CmpT>
    where CmpT : IComparer<T>, struct
{ ...
    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
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

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

class Eq[T] { boolean T.equals(T other); }

// constraint’s "inheritance"
class Comparable[T] extends Eq[T] { int T.compareTo(T other); }

class Baz[A, B] { ... }

class Foo[A, B] { ... }

double FindMax[T](Iterable[T] vs) where Comparable[T]
{ ... if (mx.compareTo(v) < 0) ... }  // static methods.

As constraints are external to types, Java Genus supports:

• static methods;
• retroactive modeling;
• multi-type constraints.
```
Several models are allowed, and models consistency is guaranteed at the types level.

```java
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?

<table>
<thead>
<tr>
<th>“Constraints-are-Types”</th>
<th>“Constraints-are-Not-Types”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lack of language support for multi-type constraints and multiple models.</td>
<td>Language support for multi-type constraints and multiple models.</td>
</tr>
<tr>
<td>Constraints can be used as types.</td>
<td>Constraints cannot be used as types.</td>
</tr>
</tbody>
</table>

### 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”):

- 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).

```java
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).

```java
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).
References I


References II


References III


Scala’s Modular Roots. Available here.
## Comparison of Languages and Extensions

<table>
<thead>
<tr>
<th>Language Support for GP in OO Languages</th>
<th>Haskell</th>
<th>C#</th>
<th>Java 8</th>
<th>Scala</th>
<th>Ceylon</th>
<th>Kotlin</th>
<th>Rust</th>
<th>Swift</th>
<th>JavaGI</th>
<th>G</th>
<th>C#pt</th>
<th>Genus</th>
<th>Mod/Impl</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Constraints can be used as types</strong></td>
<td>o</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td><strong>Explicit self types</strong></td>
<td>-</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Multi-type constraints</strong></td>
<td>●</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><strong>Retroactive type extension</strong></td>
<td>-</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Retroactive modeling</strong></td>
<td>●</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><strong>Type conditional models</strong></td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><strong>Static methods</strong></td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><strong>Default method implementation</strong></td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Associated types</strong></td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><strong>Constraints on associated types</strong></td>
<td>○</td>
<td>-</td>
<td>-</td>
<td>●</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>●</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Same-type constraints</strong></td>
<td>○</td>
<td>-</td>
<td>-</td>
<td>●</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>●</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Concept-based overloading</strong></td>
<td>o</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Multiple models</strong></td>
<td>o</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Models consistency (model-dependent types)</strong></td>
<td>-</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Model genericity</strong></td>
<td>o</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>●</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

* 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
\begin{verbatim}
data Nat  -- Nat : Type  
  = Zero  -- Zero : Nat  
  | Succ Nat -- Succ : Nat -> Nat
\end{verbatim}

-- generic list
\begin{verbatim}
data List a -- List : Type -> Type  
  = []  -- [] : List a  
  | (::) a (List a) -- (::) : a -> List a -> List a
\end{verbatim}

-- 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

\textbf{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**

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

**Concept Parameters**

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

```csharp
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
  compare :: a -> a -> Ordering
  (<=) :: a -> a -> Bool ...

instance Ord Int where
  ... -- Ord functions’ implementation

findMax :: Ord a => [a] -> a
findMax (x:xs) = ... if mx < x ...

Figure: Examples of Haskell type classes

Figure: The use of the Ord type class
Haskell Type Classes

```haskell
class Eq a where
  (==) :: a -> a -> Bool ...

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

instance Ord Int where
  ...
```

Figure: Examples of Haskell type classes

```haskell
findMax :: Ord a => [a] -> a

findMax (x:xs) = ...
```

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
Traits are used in Scala instead of interfaces.

```scala
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)
**Traits** are used in Scala instead of interfaces.

```scala
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)

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

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

```scala
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
```