MICROSOFT RESEARCH, .NET GENERICS RESEARCH PROJECT - GENERIC C# SPECIFICATION, v16, AS OF 12 DEC 2001

Authors: Don Syme, Andrew Kennedy, Claudio Russo

{dsyme,akenn,crusso}@microsoft.com

This document is designed to be read as an addendum to the C# 1.0 specification.

18. Generics

C# permits classes, structs, interfaces and methods to be parameterized by the types of data they store and manipulate, through a set of features known collectively as *generics*. C# generics will be immediately familiar to users of generics in Eiffel or Ada, or to users of C++ templates, though they do not suffer many of the complications of the latter.

Generics are useful because many common classes and structs can be parameterized by the types of data being stored and manipulated – these are called *generic classes* and *generic structs*. Similarly, many interfaces define contracts that can be parameterized by the types of data transacted – these are called *generic interfaces*. Examples of generic classes in the .NET Framework are the collection classes `System.Collections.Generic.ArrayListG` and `System.Collections.Generic.HashtableG`, and examples of generic interfaces are `System.Collections.Generic.IListG` and `System.Collections.IDictionaryG`. Methods may also be parameterized by type in order to implement “generic algorithms”, and these are known as *generic methods*. Often the static methods in a generic class will be parameterized in this way, for example some of the overloads of `System.Array.Sort`.

Comment [DRS1]: *Note: no decision has been made on the exact names for generic collection classes or which classes will be supported.*

The purpose of this chapter is to describe precisely the extensions to C# to provide support for generic programming.

Notation

Throughout this chapter we use the following notation:

`C`, `C1`, `C2`, … refer to classes, either generic or non-generic (§18.4).

`I`, `I1`, `I2`, ... refer to interfaces, either generic or non-generic (§18.7).

`V`,`U`, `W`, ... refer to *type-parameters* (§18.2.1)

`T`, `T1`, `T2`, ... refer to types, including basic C# types (§MainSpec), as well as constructed types (§18.2.2) and type parameters used as types (§18.2.1).

`B`, `B1`, `B2`, ... refer to types used as *explicit-type-parameter-constraints* (§18.8.3).

<*inst*>, <*inst2*>, … refer to *type instantiations* (§18.2.4).

18.1 Generics by examples

Generic collections represent the simplest and most common use of generic classes, interfaces and structs. The code below shows a very simple generic class that wraps an array as a read-only generic vector (the indexer does not allow elements to be updated):

```
class Vector<V>
{
   private V[] data;
   public virtual V this[int n]  { get { return data[n]; }  }
   public Vector(V[] init)  { data = init; }
   public int Length { get { return data.Length; } }
}
```

The code below provides arrays that automatically expand as values are assigned at indices:

```
class ExpandingArray<V>
{
   private V[] data;
   public virtual V this[int n]
   {
      get { return data[n]; }
      set
      {
         if (data == null)
            data = new V[n+1];
         else if (n >= data.Length)
         {
            int oldlen = data.Length;
            int newlen =  Math.max(n+1,oldlen * 2);
            V[] newdata = new V[newlen];
            for (int i = 0; i < oldlen; i++)
               newdata[i] = data[i];
            data = newdata;
         }
         data[n] = value;
      }
   }
   public ExpandingArray () { data = null; }
}
```

Generic methods can be used to implement generic algorithms over generic collections. For example, the following code checks if the given vector is sorted:

```
interface IComparer<V>
{
   int Compare(V v1,V v1);
   ...
}
class VectorMethods
{
   public static bool IsSorted<V>(IComparer<V> comparer, Vector<V> inp)
   {
      for (int i=1; i<inp.Length; i++)
         if (comparer.Compare(inp[i-1], inp[i]) > 0)
            return false;
      return true;
   }
}
```

Generic classes and methods often use additional methods that help them "interpret" type parameters. For example, the class may require a way of comparing values of type `V`. These methods are typically provided in one of the following ways:

(a) Explicitly, by requiring that the client of the class pass extra parameters to the class that provide the functionality. For example, an instance constructor for the class may require a value of type `Icomparer<V>` to be passed in. This value can then be used to compare values of type `V`, as show in the example above.

(b) Explicitly, by using *explicit constraints* (§18.8.3). In this case, the signature of the generic class specifies that any type used to instantiate the class must support particular types. If there is only one such constraint it can be written directly after the type parameter, e.g. `class C<V : IComparable>`. The example below shows how explicit constraints can be specified for a generic method.

(c) Implicitly, by casting values of type `V` to an appropriate type within the body of the class (e.g. casting value to type `IComparable`) and then calling a method. This technique is known as *implicit constraints*. A similar way to implicitly access functionality from a type parameter is to use reflection.

Comment [crusso2]: We should state somewhere that a constraint can also be a **class** type, not just an interface --- I don't think any of the examples illustrate this.

The following code illustrates the use of explicit constraints (§18.8.3) to define a method that checks if the given vector is sorted.

```
interface IComparable<V> { int CompareTo(V); }
class VectorMethods
{
   public static bool IsSorted<V : IComparable<V>>(Vector<V> inp)
   {
      for (int i=1; i<inp.Length; i++)
         if (inp[i-1].CompareTo(inp[i]) > 0)
            return false;
      return true;
   }
}
```

Comment [PS3]: *Currently this will not compile, (1) because >> is not handled correctly, and (2) because constraints are not supported in generic methods.*

18.2 Constructed types, type arguments, type parameters and type instantiations

Generics extend the set of types available to the C# programmer in two ways, by adding ***constructed types*** and ***type parameters***. Constructed types can be used in both generic and non-generic code to declare values whose types involve generic classes, structs and interfaces.

18.2.1 Type parameters

Type parameters can be used only in generic code, i.e. type parameters are visible within the bodies of *generic-class-declarations* (§18.4), *generic-struct-declarations* (§18.5), *generic-method-declarations* (§18.6), *generic-interface-declarations* (§18.7) and *generic-delegate-declarations* (§18.8). Properties, indexers and events may not be generic themselves, although instance properties, indexers and events may utilize the type parameters of their surrounding class.

type-parameter:
  simple-identifier

The following rules apply to all type parameters:

- A type parameter has a scope, and within that scope may be used to form a *type*. The scope of a type parameter depends on the kind of declaration to which it is attached.
- When used to form a type, the accessibility of a type parameter is `public`.

18.2.2 Constructed types

A constructed type `C<T1,…,Tn>` is formed by applying a type name `C` to an appropriate number of ***type arguments*** `T1,…,Tn`. The type arguments `T1,…,Tn` together form a ***type instantiation***. C# supports four different kinds of constructed types: *constructed-class-types*, *constructed-struct-types*, *constructed-interface-types*, and *constructed-delegate-types*.

class-type:
  constructed-class-type
  type-name
  `object`
  `string`

constructed-class-type:
  type-name < *type-arguments* >

struct-type:
  constructed-struct-type
  type-name
  simple-type

constructed-struct-type:
  type-name < *type-arguments* >

interface-type:
  constructed-interface-type
  type-name

constructed-interface-type:
  type-name < *type-arguments* >

delegate-type:
  constructed-delegate-type
  type-name

constructed-delegate-type:
  type-name < *type-arguments* >

A constructed type C<T_1,…,T_n> is only ***valid*** if all the types T_1, …, T_n are well formed, the class name `C` refers to a generic class accessible from the current context, the generic class expects exactly *n* type parameters, and the instantiation <T_1, …, T_n> satisfies the constraints (§18.9.2) for the generic class. The validity of constructed interface types, delegate types and struct types is defined in a similar fashion.

18.2.2.1 Accessibility of constructed types

A constructed type `C<T1,…,Tn>` is accessible when all its parts `C`, `T1`, …, `Tn` are accessible. For instance, if the generic type name `C` is `public` and all of the *type-arguments* `T1`,…,`Tn` are accessible as `public`, then the constructed type is accessible as `public`, but if either the *type-name* or one of the *type-arguments* has accessibility `private` then the accessibility of the constructed type is `private`. If some part of the

Comment [DRS4]: *Aside: The above syntax does not permit the use of ">>" in a type instantiation or declaration of the constraint. The C# grammar needs to be modified to permit this.*
The basic way to do this is to "unroll" all the rules that permit a ">" in the final position, so that the second copy of them expects two ">"'s. Then a new rule can be added that can take a ">>".
For example, the grammar

```
type = C < type >
     | int
```

becomes

```
type = C < type1
      | int
type1 = type >
      | C < type2
type2 = type >>
```

or the grammar

```
type = C < types >
     | int
types = types , type
      | type
```

becomes

```
type = C < types1
      | int
types = types, type
       | type
type1 = type >
      | C < types2
types1 = types, type1
       | type1
type2 = type >>
types2 = types, type2
       | type2
```

This is gross, of course, especially for a large grammar, but in the existing recursive descent C# compiler we would implement this by keeping a count of the number of ">" that are expected at the end of aconstruct. For the grammar specification there isn't a lot of choice but to give all the rules, or a general way of deriving the rules.

Comment [crusso5]: Note the various ***-type productions are ambiguous when combined. The main C# spec is also written in this way, or was when I last checked.

constructed-type has accessibility `protected`, and another part has accessibility `internal`, then the constructed-type is accessible only in this class and its subclasses in this assembly.

More precisely, the accessibility domain for a constructed type is the intersection of the accessibility domains of its constituent parts. Thus if a method has a return type or argument type that is a constructed-type where one constituent part is `private`, then the method must have an accessibility domain that is `private`; see §3.5.

For example:

```
public class B<T,U> { }

internal class C

{

  // Because C is internal,all the following types
  // have their given accessibility domains intersected with
  // "internal"

  protected internal class CProInt { } // i.e.internal
  public class CPub { } // i.e. internal
  protected class CPro { } // i.e. intersect(protected,internal)
  internal class CInt { } // i.e. internal
  private class CPri { } // i.e. private

  // Because C is internal,all the following methods
  // really have an accessibility domain of "internal"

  public B<CPub,CPub> m11() { ... }    // Ok
  public B<CPub,CProInt> m12() { ... } // Ok
  public B<CPub,CInt> m13() { ... }    // Ok
  public B<CPub,CPro> m14() { ... }    // Error, CPro is protected
  public B<CPub,CPri> m15() { ... }    // Error, CPri is private

  // Because C is internal,all the following methods
  // really have an accessibility domain of "internal"

  protected internal B<CProInt,CPub> m21() { ... }  // Ok
  protected internal B<CProInt,CProInt> m22() { ... }  // Ok
  protected internal B<CProInt,CInt> m23() { ... }  // Ok
  protected internal B<CProInt,CPro> m24() { ... } // Error, CPro prot.
  protected internal B<CPro,CPri> m25() { ... }   // Error, CPri private

  internal B<CInt,CPub> m31() { ... }    // Ok
  internal B<CInt,CProInt> m32() { ... } // Ok
  internal B<CInt,CInt> m33() { ... }    // Ok
  internal B<CInt,CPro> m34() { ... }    // Error, CPro protected
  internal B<CInt,CPri> m35() { ... }    // Error, CPri is private

  // Because C is internal,all the following methods
  // really have an accessibility domain that is the intersection of
  // "protected" and "internal"

  protected B<CPro,CPub> m41() { ... }    // Ok
  protected B<CPro,CProInt> m42() { ... } // Ok
  protected B<CPro,CInt> m43() { ... }    // Ok
  protected B<CPro,CPro> m44() { ... }    // Ok
  protected B<CPro,CPri> m45() { ... }    // Error, CPri is private

  private B<CPri,CPub> m51() { ... }    // Ok
  private B<CPri,CProInt> m52() { ... } // Ok
  private B<CPri,CPro> m53() { ... }    // Ok
  private B<CPri,CInt> m54() { ... }    // Ok
  private B<CPri,CPri> m55() { ... }    // Ok

}

public class D

{
```

Comment [DRS6]: Sorry for the forribly long example – but I thought it worth going through every case, and once I did I thought I would record the results to help your testers construct the test cases.

Test incompleteness: Peter Sestoft says all this works, but I haven't checked.

```
    protected internal class DProInt { }
    public class DPub { }
    protected class DPro { }
    internal class DInt { }
    private class DPri { }

    public B<DPub,DPub> m11() { ... }    // Ok
    public B<DPub,DProInt> m12() { ... } // Error, DProInt not public
    public B<DPub,DInt> m13() { ... }    // Error, DInt not public
    public B<DPub,DPro> m14() { ... }    // Error, DPro not public
    public B<DPub,DPri> m15() { ... }    // Error, DPri not public

    protected internal B<DProInt,DPub> m21() { ... }  // Ok
    protected internal B<DProInt,DProInt> m22() { ... }  // Ok
    protected internal B<DProInt,DInt> m23() {...}// Error, DInt not prot.
    protected internal B<DProInt,DPro> m24() {...}// Error, DPro not int.
    protected internal B<DPro,DPri> m25() { ... } // Error, DPri is private

    internal B<DInt,DPub> m31() { ... }    // Ok
    internal B<DInt,DProInt> m32() { ... } // Ok
    internal B<DInt,DInt> m33() { ... }    // Ok
    internal B<DInt,DPro> m34() { ... }    // Error, DPro not internal
    internal B<DInt,DPri> m35() { ... }    // Error, DPri not internal

    protected B<DPro,DPub> m41() { ... }    // Ok
    protected B<DPro,DProInt> m42() { ... } // Ok
    protected B<DPro,DInt> m43() { ... }    // Error, DInt not protected
    protected B<DPro,DPro> m44() { ... }    // Ok
    protected B<DPro,DPri> m45() { ... }    // Error, DPri not protected

    private B<DPri,DPub> m51() { ... }    // Ok
    private B<DPri,DProInt> m52() { ... } // Ok
    private B<DPri,DPro> m53() { ... }    // Ok
    private B<DPri,DInt> m54() { ... }    // Ok
    private B<DPri,DPri> m55() { ... }    // Ok
}
```

18.2.2.2 Using aliases

Using aliases are extended to constructed types:

```
namespace N
{
   using Ints = List<int>;
   class List<V> { ... }
   class C {
       void m(Ints x);
   }
}
```

Using aliases may themselves be parameterized:

```
namespace N
{
   using RedirectTable<V> = Hashtable<V,V>;
   class C {
       void m(RedirectTable<int> x);
   }
}
```

Using aliases cannot refer to type parameters other than those declared for a parameterized using alias.

18.2.3 Type arguments

Each type argument is simply a type or the special type argument `void`.

Comment [DRS7]: *Implementation incompleteness: This is not implemented.*

Comment [DRS8]: *Implementation incompleteness: This is not implemented. Good error messages become slightly harder to produce in the presence of this kind of alias, so a little care and thought should be given before implementing it.*

Comment [DRS9]: Comment from Claudio: You should make precise that aliases, ultimately, can't be cyclic, and what the scoping rules for aliases, and their bodies, are.
Response: this is already covered in the main C# spec (I presume).

type-arguments:
  type-argument
  type-arguments `,` *type-argument*

type-argument:
  type
  `void`

Type arguments may in turn be constructed types. In unsafe code (§MainSpec) the *type-arguments* may include pointer types. Every constructed type must satisfy any constraints on the corresponding type parameters of the *type-name* (§18.8.3).

18.2.3.1 Using "void" as a type argument

The use of `void` as a type argument is simply a short-hand for the use of the empty struct type `System.Empty`. Thus when an instantiation causes a generic class, method, interface, struct or delegate to possess a method or property with a return type `void` then an access to the method will result in a method with return type `System.Empty`, and when an instantiation causes a generic class, method, interface, struct or delegate to expect an argument of type `void` then a value of type `System.Empty` will be passed by the C# compiler instead.

The replacement of `void` by `System.Empty` is essentially transparent. However, the use of `void` and `System.Empty` are not equivalent or completely interchangeable – the C# compiler applies special rules for the former (§18.4.7.1, §18.8.1.1) but not the latter.

Comment [DRS10]: *Implementation incompleteness. This is not implemented in the C# compiler.*
The GCLR does NOT support the use of type "void" as a type argument. This is because "void" has such a special status in the IL, e.g. you do not need to emit a "pop" when a method returns a value of type "void". However, it is still possible to support the use of type "void" as a type argument by systematically translating it to an empty value type, e.g. let's call it System.Empty. However, when a call is made to a method that actually returns something of type "System.Empty" in the underlying IL the compiler will HAVE to emit an extra "pop" instruction where it wouldn't have to if the type was "void". The generic code does not change at all – only code that instantiates at type "void". Note also that this may lead to the creation of arrays of type System.Empty[].

18.2.4 Type instantiations

A ***type instantiation*** <$V_1 \rightarrow T_1, \ldots, V_n \rightarrow T_n$> for a class `C` with type parameters $V_1, \ldots, V_n$ specifies a type T_i for each type parameter V_i of `C`. Type instantiations are a notion that are used only within this specification for the purpose of succinctness and clarity – they do not occur syntactically in a C# program.

A constructed type may simply be written `C<`*inst*`>` where `C` is a type name and *inst* is a type instantiation. Thus when we write a constructed type as `C<`*inst*`>`, it should be understood that *inst* can be used to refer not just to the vector of types $T_1, \ldots, T_n$ but also to the instantiation <$V_1 \rightarrow T_1, \ldots, V_n \rightarrow T_n$> when the class `C` has formal type parameters $V_1, \ldots, V_n$. Likewise, an instantiation <$V_1 \rightarrow T_1, \ldots, V_n \rightarrow T_n$> may just be written <$T_1, \ldots, T_n$> when it is clear which class and type parameters are being referred to e.g. the formal type parameters $V_1, \ldots, V_n$ for the class `C` when writing `C`<$T_1, \ldots, T_n$>.

Type instantiations may also be built for generic structs, interfaces, methods and delegates. An ***empty class type instantiation*** has no entries at all, and corresponds to an instantiation for a non-generic class, struct, interface, method or delegate.

Except where otherwise indicated, `C<`*inst*`>` covers both the cases where the class is a non-generic class (e.g. a non-generic class such as `string`) and *inst* is the empty type instantiation, and the case where `C` is a generic class and *inst* is non-empty.

For example, in the code

```
class B<U1,U2> { ... }

class C<V1,V2>
{
   public B<V1,string> f1() { ... }

   public B<V1,V2> f2() { ... }

   public B<string,string> f3() { ... }
}

class D
{
   public B<string,string> f1() { ... }
```

```
    public D f2() { ... }
}
```

the following *type instantiations* occur:

```
<U1 → V1,     U2 → string>    (maps U1 to V1,     U2 to string)
<U1 → V1,     U2 → V2>        (maps U1 to V1,     U2 to V2)
<U1 → string, U2 → string>    (maps U1 to string, U2 to string)
<>                            (contains no mappings, e.g. for the type D)
```

Note that the instantiation `<U1 → string, U2 → string>` occurs twice in the code.

A notion that is useful on occasion is the ***formal type instantiation*** for a class. For a class `C` with type parameters V_1, …, V_n this simply assigns each type parameter to itself, e.g. V_1 to V_1 etc.

An instantiation may be ***applied to a type*** by simultaneously substituting types for type parameters throughout the type. An instantiation may also be ***applied to a signature*** by applying the instantiation to all the types in the signature. An instantiation may be ***composed*** with another instantiation, e.g. composing *inst1* with *inst2* means generating a new instantiation with the same domain as *inst2* by applying *inst1* to all the types (right-hand sides of arrows `V → T`) in *inst2*.

For example, the composition of the following two instantiations *inst1* and *inst2*

```
<W → string>           (maps W to string)
<U → W, V → List<W>>  (maps U to W, V to List<W>)
```

is

```
<U → string, V → List<string>>  (maps U to string, V to List<string>)
```

18.3 Conversions

18.3.1 Reference conversions to and from constructed reference types

Constructed reference types support implicit reference conversions. These rules replace the corresponding rules listed in (§6.1.4 - Implicit reference conversions).

- From any *reference-type* `S` to any *reference-type* `T` if `S` is derived from `T`, i.e. `T` is one of the *base types* of `S` (§18.12.3)

Note that this single general rule includes the following as special cases:

- From any *class-type* `S` to any *class-type* `T` from which it is derived, i.e. `T` is one of the *base class types* of `S` (§18.12.1.1)
- From any *class-type* `S` to any *interface-type* `T`, provided `S` implements `T`, i.e. `T` is one of the *base interface types* of `S` (§18.12.1.2)
- From any *interface-type* `S` to any *interface-type* `T`, provided `S` is derived from `T`, i.e. `T` is one of the *base interface types* of `S` (§18.12.2)

Constructed reference types support the following explicit reference conversions. These rules replace the corresponding rules listed in (§MainSpec - Explicit reference conversions).

- From any *reference-type* `S` to any *reference-type* `T` if `T` is derived from `S`, i.e. `S` is one of the *base types* of `T` (§18.12.3)

Note that this single general rule includes the following as special cases:

- From any *class-type* `S` to any *class-type* `T` if `T` is derived from `S`, i.e. `T` is one of the *base class types* of `S` (§18.12.1.1)
- From any *class-type* `S` to any *interface-type* `T`, provided `S` implements `T`, i.e. `T` is one of the *base interface types* of `T` (§18.12.1.2).

- From any *interface-type* `S` to any *interface-type* `T`, provided `T` is derived from `S`, i.e. `T` is one of the *base interface types* of `S` (§18.12.2).

Constructed reference types also support the conversions common to all reference types, e.g. from the "null" type, and to the type `object`, and from any *constructed-delegate-type* to `System.Delegate` (§MainSpec).

18.3.2 Conversions to and from constructed value types

Constructed value types also support the conversions common to all value types, e.g. to and from the types `object` and `System.ValueType` (§MainSpec).

18.3.3 User defined conversions to and from constructed types

TBD

18.3.4 Conversions to and from type parameters

An implicit, unchecked conversion exists from a type parameter `V` to type `T` under the following conditions:

- If `T` is the type `object`. This is executed as a boxing conversion whenever `V` is instantiated to a value type.
- If `T` is a reference type and `T` is one of the *explicit-type-parameter-constraints* of `V`. This is executed as a boxing conversion whenever `V` is instantiated to a value type.

The method `Wrap` in following code takes a value of type `V` and returns it as a value of type `object`.

```
public class Main
{
    static object Wrap<V>(V x)   {  return x; }
    static void Main()
    {
         object s = “string”;
         object obj1 = Wrap<string>("string");  // Ok
         string s1 = (string) obj1; // Ok, cast succeeds
         object obj2 = Wrap<int>(3);     // Ok
         int i2 = (int) obj2; // Ok, unbox
         object obj3 = Wrap<object>((object) 3);  // Ok
         string s3 = (string) obj3; // InvalidCastException
    }
```

An explicit, checked conversion exists from type `T` to a type parameter `V` under the following conditions:

- If `T` is the type `object`. This is executed as an unboxing conversion whenever `V` is instantiated to a value type.
- If `T` is a reference type and `T` is one of the type constraints of `V`. This is executed as an unboxing conversion whenever `V` is instantiated to a value type.

These conversions are checked at runtime.

The following code takes an object and attempts to return it as type `V`.

```
public class Main
{
    static V Cast<V>(object x)   {  return (V) x; }
    static void Main()
    {
            object s = “string”;
            string v1 = Cast<string>(s);  // Ok
            string v2 = Cast<int>((object) 3);  // Ok, 3 is boxed then unboxed
            string v3 = Cast<string>((object) 3);  // InvalidCastException
    }
```

18.3.5 No “co-variance” for constructed types

No special conversions exist between constructed reference types other than those listed above. In particular, constructed reference types do not exhibit “co-variant” conversions, unlike C# array types. This means that a type `List<B>` has an (identity) reference conversion to `List<B>`, but no reference conversion (either implicit or explicit) exists to `List<A>` even if `B` is a derived from `A`. In particular, no conversion exists from `List<B>` to `List<object>`.

The rationale for this is simple: if a conversion to `List<A>` is permitted, then apparently one can store values of type `A` into the list. But this would break the invariant that every object in a list of type `List<B>` is always a value of type `B`, or else unexpected failures may occur when assigning into collection classes.

One choice is to support a runtime check every time a generic data structure is modified. For example, C# does support co-variant array types, and therefore typically performs a runtime check on every store into an array of reference type. However, one of the aims of generics is to avoid the performance costs associated with this kind of check, and hence co-variance is not supported. In addition, when co-variance is not supported, the programmer can be assured that assignments into collection classes will not raise exceptions at runtime. Also note that if the generic class `List` is itself derived from some class (e.g. a non-generic class `Collection`) then there will be reference conversions from types such as `List<B>` to that class.

The behavior of conversions and runtime type checks is illustrated below:

```
    class A { ... }
    class B : A { ... }
    class List<V> { ... }
    public static void MyMethod(List<A> argl) { ... }
    public static void Main() {
       List<A> al = new List<A>();
       List<B> bl = new List<B>();
       if (al is List<A>)                         // true
          Console.WriteLine(“al is List<A>”);
       if (al is List<B>)                         // false
          Console.WriteLine(“al is List<B>”);
       if (bl is List<A>)                         // false
          Console.WriteLine(“bl is List<A>”);
       if (bl is List<B>)                         // true
          Console.WriteLine(“bl is List<B>”);
       MyMethod(al); // Ok
       MyMethod(bl); // Error, bl is not List<A>
    }
```

The last method call causes a compile time error. If the last method call in the above program were omitted, the program would produce:

```
    al is List<A>
    bl is List<B>
```

18.3.6 "Default" values for type parameters

No implicit conversion exists from the null type to a type parameter `V`. This is because `V` may be instantiated as a value type, and "null" does not have the same, intuitive meaning that may be expected for those types.

However, the expression `default<V>` is guaranteed to produce a "default" value for the type corresponding to `V`. This can be considered an explicit, unboxing conversion from the "null" type to a type parameter.

The default value for a type is guaranteed to have the following properties:

- If `V` is a reference type, then the value is `null`.
- If `V` is a struct type, then the value is an element of the struct type where every field is set to its corresponding default value.
- If `V` is a built-in value type, then its value is the default value for that type.

Comment [DRS11]: *Implementation incompleteness: the expression ((V) null) is not implemented, apart from the bit about local values giving a "hack" way to generate "zero" values. When implemented the code generated should just generate a dummy local value as there is no "ldnull.any V" instruction on the CLR.*

Default values are useful in a number of circumstances, e.g. for initializing slots in data structures. Another use is to "blank out" slots of data structures once the data previously held in the slot is no longer required. Blanking out data can help eliminate a source of bugs in garbage collected programs known as "memory leaks", which occur because the garbage collector cannot reuse space if data structures continue to contain handles to objects, even if those objects will no longer be used.

Default values may also be generated by creating an instance variable of the type `V` and declaring it `readonly,` so that it is never written to, as its initial value will also be the default value for `V`.

Comment [DRS12]: However, note that this causes the compiler to emit an irritating warning for each such declaration.

18.4 Generic classes

A ***generic-type-declaration*** is either a ***generic-class-declaration*** (§18.4), a ***generic-struct-declaration*** (§18.5), a ***generic-interface-declaration*** (§18.7), or a ***generic-delegate-declaration*** (§18.8).

```
generic-type-declaration:
    generic-class-declaration
    generic-struct-declaration
    generic-interface-declaration
    generic-delegate-declaration
```

As with any *type-declaration*, a *generic-type-declaration* can occur as a top-level declaration in a compilation unit or as a member declaration within a namespace, class, or struct.

A ***generic-class-declaration*** declares a new class parameterized by one or more types.

```
generic-class-declaration:
    attributes_opt   class-modifiers_opt
      class   identifier  <  class-type-parameters >
        class-base_opt   class-body   ;_opt
```

It is identical to a *class-declaration* except for the inclusion of the *class-type-parameters*. These may be constrained (see §18.8.3 for the specification of *explicit-type-parameter-constraint* and *extra-type-parameter-constraints*).

```
class-type-parameters:
    basic-class-type-parameters extra-type-parameter-constraints_opt
```

Comment [DRS13]: *Unimplemented: extra-type-parameter-constraints are not yet supported syntactically. The .NET CLR metadata does support these.*

```
basic-class-type-parameters:
    class-type-parameter
    class-type-parameters  ,  class-type-parameter

class-type-parameter:
     identifier  explicit-type-parameter-constraint_opt
```

A *class-type-parameter* specifies a name and an optional *explicit-type-parameter-constraint* (§18.8.3). The following generic class has one type parameter `V` with no constraint.

```
class LinkedList<V>
{
   public V hd;
   public LinkedList<V> tl;
}
```

Generic class declarations follow the same rules as normal class declarations except where noted, and particularly with regard to naming, nesting and the permitted access controls. Generic class declarations may be nested inside other class and struct declarations.

Generic classes may not be "overloaded" by arity, that is a generic class must be uniquely named within a scope in the same way as ordinary classes.

```
class C { }
class C<V> { } // Error, C defined twice
class C<U,V> { } // Error, C defined twice
```

The *class-type-parameters* of a *generic-class-declaration* can be used to form types in certain parts of the class, according to the following rules:

- A *class-type-parameter* may be used to form a type in every non-static declaration in the class, as well as the instance constructors, the specification of the *explicit-type-parameter-constraints* and the *class-base* of the *generic-class-declaration*.
- A *class-type-parameter* may not be used to form a type in the attributes, static methods, static fields, operators, destructors, the class initializer or nested types of the *generic-class-declaration* An attempt to do so is a compile-time error, and thus the *class-type-parameter* effectively hides any other visible type names. This means that class type parameters are in scope in static members, but illegal, rather than not in scope at all.
- It is a compile time error to have a nested class with the same name as a class type parameter of the enclosing class.
- A *class-type-parameter* may be used to form a type in the field initializers of all non-static fields.

Comment [DRS14]: *Testing incompleteness: I've never tested generic classes being nested inside non-generic classes. I guess it will just work.*

Comment [DRS15]: I think it's strongly worth considering changing this rule. The really useful case is for families of generic classes that take variable numbers of type parameters, e.g. Tuple and DataSet. DataSet is a good example: "DataSet" would be untyped, "DataSet<T>" would have one column of elements of type T (and perhaps other columns of unspecified data type), "DataSet<T1,T2>" would have at least two columns and so on. This really becomes much more painful if you always have to write "Tuple2,Tuple3,Tuple4, …" and "DataSet2", "DataSet3", etc.

However, there is no way we can modify the CLR to make arity significant, so the C# compiler would have to name mangle. This is not _that_ bad, and could form part of a standard CLS-like naming convention, e.g. if a generic type is part of a family of generic types up to some maximum N then each member of the family should be named "C$0" etc.

We would have to consider the ramifications on other aspects of the spec – but I believe there aren't many.

Comment [DRS16]: *Testing incompleteness: I've not tested the scoping and visibility rules for type parameters quite as widely as I should have, e.g. it's possible type parameters from the base class or from outer classes are somehow visible. I guess I've covered 95% of the cases but I remember the code confused me so I may have missed one.*

Comment [DRS17]: *Testing completeness: I have no idea if nested class names currently hide type parameter names or vice-versa – a conflict here should be a compile time error.*

Comment [DRS18]: *Testing incompleteness: I've never tested this thoroughly – just a few adhoc tests and it seemed to work.*

18.4.1 Class base specification

The specification of the base class of any *class-declaration* is a class type, and in particular this might be a *constructed-class-type* (§18.2.2). In a *generic-class-declaration* it may not be a *class-type-parameter* on its own, though it may involve the *class-type-parameters* that are in scope.

The specification of each base interface of any *class-declaration* is an interface type, though note some of these may be *constructed-interface-types* (§18.2.2). In a *generic-class-declaration* it may not be a *class-type-parameter* on its own, though it may involve the *class-type-parameters* that are in scope.

The following code illustrates how a class can implement and extend constructed types:

```
class C<U,V> { }
interface I1<V> { }
class D : C<string,int>, I1<string> { }
```

Methods in a class that overrides or implements methods from a base class or interface must provide appropriate methods at specialized types.

The following code illustrates how methods are overridden and implemented. This is explained further in §18.4.4.1.

```
class C<U,V>
{
   public virtual void m1(U x,List<V> y) { ... }
}
```

```
interface I1<V>
{
   V m2(V);
}
class D : C<string,int>, I1<string>
{
   public override void m1(string x, List<int> y) { ... }
   public string m2(string x) { ... }
}
```

The base class and base interfaces of a class may not be type parameters:

```
class Extend<V> : V { ... }  // Error, the base class may not be a
                             // type parameter
```

Comment [DRS19]: I think the compiler gives a bad error message or AVE on this.

18.4.1.1 Uniqueness of interface types

The set of base interface types of a class (§18.12.1.2) must not contain two *constructed-interface-types* with the same *generic-interface-name* and different instantiations. This simplifies the implementation of dynamic dispatch mechanisms and prevents ambiguities arising amongst methods and ambiguities when generic interfaces are instantiated. Because of this restriction, the following class hierarchy is illegal:

```
interface I<V> { }
class C : I<string>, I<object> { }
```

This restriction may be lifted in later versions of C#.

Comment [DRS20]: *Unimplemented: The check for this condition is not yet implemented.*

Comment [DRS21]: *NOTE: This issue would require some fairly substantial work in the .NET CLR to fix, but my guess is that it could be done within the Whidbey timeframe if needed. The spec for some related issues is included in the footnote below.* [1]

18.4.2 Members in generic classes

Generic classes can contain essentially the same kinds of members as specified in §MainSpec, although particular rules apply in each case.

§18.12.4 defines when the *set of all function members* of a class is well formed. In particular, it checks rules such as the fact that an "override" method must have the correct signature to override a virtual method in a parent class. If this set is not well formed an error is reported at compile-time.

The static constructor of a generic class is executed once only, and ***not*** once for each different instantiation of that class. Destructors are executed once for each instance object created for each instantiation.

18.4.3 Fields in generic classes

18.4.3.1 Instance variables in generic classes

The instance variables of a generic class may have types and variable initializers that include any type parameters from the enclosing class. For example:

```
class C<V> {
  public V f1;
  public C<V> f2 = null;
  public C(V x) { this.f1 = x; }
}

class Application {
  static void Main() {
    C<int> x1 = new C<int>(1);
    Console.WriteLine(x1.f1);  // Prints 1
    C<double> x2 = new C<double>(3.1415);
    Console.WriteLine(x2.f1);  // Prints 3.1415
  }
}
```

18.4.3.2 Static variables in generic classes

The types of static variables in a generic class may not refer to any type parameters from the enclosing class.

A static variable in a generic class is shared amongst all the types constructed from it. There is no way to specify fields where a new storage location is created for each instantiation of the class.

For example:

```
class C<V> {
  static int count = 0;
  public C() { count++; }
  static public int Count { get { return count; } }
}

class Application {
  static void Main() {
    C<int> x1 = new C<int>();
    Console.WriteLine(C.Count);  // Prints 1
    C<double> x2 = new C<double>();
    Console.WriteLine(C.Count);  // Prints 2
    C<object> x3 = new C<object>();
    Console.WriteLine(C.Count);  // Prints 3
  }
}
```

18.4.4 Methods in generic classes

This section discusses *method-declarations* within generic classes. In passing it is worth noting that the signatures of all methods may involve *constructed types* of some kind, e.g. *constructed-class-types*, whether generic or non-generic, or whether in generic classes or non-generic classes.

Methods within a generic class can be overloaded. See §18.10.6 for a description of how overloading is resolved at call sites.

§18.4.7 below describes how virtual methods declared within generic classes can be overridden, whether the overriding method occurs in a generic class or in a non-generic class. §18.6 describes method declarations which are themselves generic, and §18.4.4.4 below describes how these methods are allowed to themselves occur within generic classes.

18.4.4.1 Instance, abstract and virtual methods in generic classes

For non-static methods inside generic classes the signature may also involve one or more of the *class-type-parameters*. The body of such a method may also use these type parameters.

The following example shows an instance method, an abstract method and two virtual methods within a generic class:

```
abstract class C<V>
{
   private string name;
   private V data;
   public bool CheckName(C<V> x)
```

```
         { return (this.name == x.name); }
      ...

      public virtual V GetData()
         { return data; }
      public abstract C<V> CopyData();
      public virtual void ReplaceData(C<V> x)
         {  this.data = x.data; }
   }
```

18.4.4.2 Static methods in generic classes

Static methods do not automatically acquire the type parameters of a generic class in which they are used. For this reason neither the argument types nor the return types of a static method can include any type parameters from the enclosing class. Similarly the body of such a method cannot use the type parameters from the enclosing class. In this way type parameters are very similar to instance fields of the class being defined – they are only accessible in instance methods.

The following sample shows how you can make a static method generic in order to manipulate generic data:

```
   class C<V>
   {
      private string name;
      private V data;
      public static bool CheckName(C<V> x, C<V> y) // Error, cannot access V
         { return (x.name == y.name); }
      public static bool CheckName<V>(C<V> x, C<V> y) // Ok, CheckName generic
         { return (x.name == y.name); }
      public bool CheckName(C<V> y)          // Ok, instance method can access V
         { return (this.name == y.name); }
   }
```

18.4.4.3 Param-array methods and type parameters

Type parameters may be used in the type of an argument array expected for a `params` method. For example, given the declaration

Comment [DRS22]: *Testing completeness: This appears to work, but has not been thoroughly tested*

```
   class C<V>
   {
      void F(int x, int y, params V[] args);
   }
```

the following invocations of the expanded form of the method

```
   (new C<int>).F(10, 20);
   (new C<object>).F(10, 20, 30, 40);
   (new C<string>).F(10, 20, "hello", "goodbye");
```

correspond exactly to

```
   (new C<int>).F(10, 20, new int[] {});
   (new C<object>).F(10, 20, new object[] {30, 40});
   (new C<string>).F(10, 20, new string[] {"hello", "goodbye"} );
```

18.4.4.4 Generic methods in generic classes

Methods in both non-generic and generic classes may themselves be *generic-method-declarations* (§18.6). This applies to all kinds of methods, including static methods, instance methods, virtual methods, and methods declared with the `abstract`, `new` or `overrides` keywords.

Non-static generic methods within a generic class are rare, but are permitted. These can use *both* the type parameters of the class and the type parameters of the method.

```
class C<V>
{
   public static bool f1(C<V> x, C<V> y)     // Error, cannot access V
     { ...}
   public static bool f2<V>(C<V> x, C<V> y)  // Ok
     { ... }
   public bool f3(C<V> x)                    // Ok
     { ...  }
   public bool f4<U>(C<V> x, C<U> y)         // Ok
     { ...  }
   public virtual bool f5(C<V> x)            // Ok
     { ...  }
   public virtual bool f6<U>(C<V> x, C<U> y) // Ok
     { ...  }
}
```

18.4.4.5 Uniqueness of signatures

The method signatures within a generic class must be unique *prior to any instantiations*. However, the following additional rules apply in determining when two signatures are identical:

- The names of all type parameters are ignored, and instead their numerical position in a left-to-right ordering of the type parameters is used.
- The number of type parameters accepted by a generic method is significant.
- Any constraints on type parameters accepted by generic method are ignored.

Thus in the following class the indicated methods cause conflicts with those of the same name:

```
class C<V>
{
   public V f1() { ... }
   public string f1() { ... }  // Error, return types differ
   public void f2(V x) { ... }
   public void f2(object x) { ... }  // Ok, unique prior to instantiation
}
class D
{
   public static void g1<V>(V x) { ... }
   public static void g1<U>(U x) { ... }  // Error, identical ignoring names
   public static void g2() { ... }
   public static void g2<V>() { ... }  // Ok, number of params significant
   public static void g3<V>() { ... }
   public static void g3<V,U>() { ... }  // Ok, number of params significant
   public U g4<U>(object x) { ... }
   public U g4<U>(U x) { ... } // Ok, differ by formal argument type
   public static void g5<V : I1>() { ... }
   public static void g5<V : I2>() { ... }  // Error, constraints are
                                            // not signifcant
}
```

Comment [DRS23]: Imlementation incompleteness: Currently this is illegal: a method cannot appear with different degrees of genericity.

Comment [DRS24]: I think this rule should probably be changed – we're looking at changing it in the Generic CLR.

18.4.5 Nested types in generic classes

A *generic-class-declaration* can itself contain type declarations, just as with ordinary classes. However, the type parameters of the *generic-class-declaration* are ***not*** implicitly visible within the nested class declaration, i.e. the nested class declaration is not implicitly parameterized by the type parameters of the outer class declaration. Thus the type parameters play a similar role to the "this" value and the instance fields of the outer class declaration, which are not accessible in the nested type. The rationale for this is that it is frequently the case that the programmer wishes to define nested classes that take more or fewer type parameters than the outer class, and given this possibility it is better to be explicit about the exact degree of type parameterization desired.

Nested types may themselves be parameterized. The *class-type-parameters* of the enclosing class are not visible in the nested class, but within the enclosing class constructed types involving the nested class may involve the type parameters of the enclosing class. Nested types may also be parameterized by type parameters with the same names as the type parameters of the enclosing class.

For example:

```
public class LinkedList<V>
{
   Node<V> first, last;
   private class Node<V>
   {
      public Node<V> prev, next;
      public V item;
   }
}
```

18.4.6 Properties, Events and Indexers in generic classes

This section discusses *property-declarations*, *event-declarations* and *indexer-declarations* within generic classes. In passing it is worth noting that the signatures of all properties, indexers and events may contain *constructed types,* regardless of whether these are in a generic or non-generic class.

18.4.6.1 Instance properties, events and indexers in generic classes

Within a *generic-class-declaration* the type of an instance property may include the *class-type-parameters* of the enclosing class. For example, the following shows an instance property in a generic class:

Comment [DRS25]: *GC# Testing incompleteness: I havent thoroughly tested virtual properties, events etc. in generic classes, though they are implemented.*

```
class C<V>
{
   private V name;
   private V name2;
   public V Name
   {
      get { return name; }
      set { name = value; }
   }
   public V this[string index]
     { get { if (index == "main") return name; else return name2 }   }
}
```

18.4.6.2 Static properties and events in generic classes

Because class type parameters are not accessible at static property declarations (§18.4), the type of a static property cannot include any type parameters.

```
delegate D<V>(T x);  // Ok, a generic delegate
class C<V>
{
   public static V Name // Error, static properties can't use type parameter
      { ... }
   static event D<V> e; // Error, static events can't use type parameters
}
```

18.4.6.3 No generic properties, events or indexers

Properties, events and indexers may not themselves be generic. If a property-like construct is required that must itself be generic then a static or virtual generic method should be used instead.

Comment [DRS26]: We could allow generic properties. There aren't that many such useful properties, but there are a few.

For example:

```
class C<V>
{
   public C<V> p1         // Ok
      { get { return null; } }
   public C<U> p2<U>     // Error, properties may not be generic
      { ... }
   public event D<V> fire1;  // Ok
   public event D<V> fire2<V>;  // Error, events may not be generic
   public V this[int index] { ... }  // Ok
   public V this<U>[int index] { ... } // Error, syntax error
}
```

18.4.7 Overriding and generic classes

Abstract and virtual members can only be overridden according to the rules of §18.12.4 which define when the *set of all function members* for a class (generic or non-generic) is well formed. In particular, a member in a derived class with the `override` attribute must respect the instantiations specified in the inheritance chain. That is, the resulting signature must be identical to the signature of a method with the `virtual` or `abstract` attribute after the instantiations implied by the inheritance chain are taken into account in the manner described in §18.12.4.

The following example shows some examples of this:

```
abstract class C<V>
{
   private string name;
   private V data;
   public bool CheckName(C<V> x)  { ... }
   public virtual V GetData()  { ...   }
   public abstract C<V> CopyData();
   public virtual void ReplaceData(C<V> x) { ... }
}
class D : C<string>
{
   public override string GetData()  { ... }
   public override C<string> CopyData()  { ...  }
   public override void ReplaceData(C<int>) // Error, incorrect override,
                                            // should be C<string>
      { ... }
}
class E<U,W> : C<U>
{
   public override U GetData()  { ... }
```

```
      public override C<W> CopyData()  // Error, incorrect override,
         { ... }                      // should be C<U> as we derive
                                      // from C<U>
      public override void ReplaceData(C<U>) { ... }
   }
```

Note that the signatures required for the virtual methods as we override them are the same as the signatures for these methods in the class `C<V>` once we have substituted `string` for `V` (in the case of class `D`) and `U` for `V` (in the case of generic class `E`).

18.4.7.1 Returning void when overriding a method that originally returned a variable type

As specified in §18.2.3.1, when the keyword `void` is used as a type argument, the C# compiler replaces this with a type argument `System.Empty`, which is a value type containing no instance fields. In addition, when a method is used to override a virtual or abstract method *m* from a constructed base class or interface type `C<`*inst*`>`, and *m* is declared in `C` to return a type made from a type variable `V`, and furthermore *inst* instantiates `V` to `void` (interpreted as `System.Empty` by the C# compiler), then the overriding method must also return type `void`. In particular, the C# compiler will insert a new virtual method that returns the type `void`, and will override the old virtual method via a method that immediately calls the new method and then returns `System.Empty`.

Comment [DRS27]: *Implementation incompleteness: using "void" as a type parameter is not implemented, and no System.Empty class currently exists in the BCL as far as I know.*

For example, the following code is permitted:

```
   class C<U>
   {
      public virtual U PeekData()  { ... }
   }
   class E : C<void>
   {
      public override void PeekData()  { ... }
   }
```

Conceptually the "void" translation gives:

```
   class C<U>
   {
      public virtual U PeekData()  { ... }
   }
   class E : C<System.Empty>
   {
      private sealed override System.Empty C<System.Empty>.PeekData()
      {
         this.PeekData();
         return new System.Empty();
      }
      public new virtual void PeekData()  { ... }
   }
```

Note, however, that this translation cannot always be directly expressed in C# source code, because signatures may be duplicated when only argument types are considered, and private implementations of virtual methods are not permitted.

18.4.8 Operators in generic classes

Generic-class-declarations can include operator declarations. However, because class type parameters are not in scope at static member declarations (§18.4), the type of a static operator annot use any of the type parameters of the enclosing class. Instead, an operator within a *generic-class-declaration* must explicitly specify the type parameters accepted by the operator, if any. [1]

generic-operator-declaration:
  attributes$_{opt}$ *operator-modifiers* *operator-declarator type-parameters operator-body*

Operators may accept more or fewer type parameters than the enclosing class. When operators are called, there is no way to explicitly specify the actual type arguments being supplied at the point of call, and these must always be *inferred* (see §18.10.5).

The following example shows two operators. The * operator takes 2 type parameters, both of which will be inferred from the arguments provided to the operator. The ! operator takes no type parameters, and only works on values of type `Set<byte>`.

```
class Set<V>
{
   ...
   public static Set<U> operator * <V, U> (Set<V> s, IDictionary<V, U> h)
   {
      Set<U> ret = new Set<U>();
      foreach (V el in s)
       if (h.ContainsKey(s))
         ret.Add(h[el]);
```

[1] FOOTNOTE

TODO: Explicit declarations of type parameters by operators is NOT implemented. Instead, I first implemented "implicit acquisition" of type parameters as specified below, just for completeness. I now MUCH prefer explicitly declaring the type
TODO: Explicit declarations of type parameters by operators is NOT implemented. Instead, I first implemented "implicit acquisition" of type parameters as specified below, just for completeness. I now MUCH prefer explicitly declaring the type
parameters, and would suggest throwing out implicit acquisition in favour of the explicit declarations. I and Peter Sestoft keep running across examples where it makes sense to have either more or fewer type parameters on an operator, just as we do for static methods, which is why static methods do not automatically acquire their type parameters.

Operators implicitly accepting the same type parameters as the enclosing class.

An example of using operators that implicitly accept the same type parameters as the enclosing class is shown below. The example also uses "implicit constraints" (i.e. a combination of runtime casts and conversions) to access some functionality provided by the type arguments (§TBD).

```
interface IPlus<T> {   T Plus(T x); }
interface IMinus<T> {    T Minus(T x); }

// nb. Explicit constraints are not being used
// T and U should implicitly support IPlus<T> if the "+" method is used
// T and U should implicitly support IMinus<T> if the "-" method is used
class NumPair<T,U> {
  private T x;
  private U y;
  public NumPair<T,U> operator +(NumPair<T,U> a, NumPair<T,U> b)
  {
     T x2 = ((IPlus<T>) a.x).Plus(b.x);
     U y2 = ((IPlus<U>) a.y).Plus(b.y);
     return new NumPair<T,U>(x2,y2);
       }
  public NumPair<T,U> operator -(NumPair<T,U> a, NumPair<T,U> b)
  {
     T x2 = ((IMinus<T>) a.x).Minus(b.x);
     U y2 = ((IMinus<U>) a.y).Minus(b.y);
     return new NumPair<T,U>(x2,y2);
}
```

```
            return ret;
        }
        public static Set<byte> operator ! (Set<byte> s)
        {
           Set<byte> ret = new Set<byte>();
           for (byte b = 0; b < 256; b++)
             if (!s.Contains(b))
               ret.Add(b);
        }
     }
```

For all "well-behaved" sets of overloaded operators the inference process (§18.10.5) can infer unique type parameters. However this process may be ambiguous (e.g. see examples in §18.10.5.5). For normal generic method this is acceptable, as explicit type arguments can always be provided by the user to specify the exact type instantiation required. However for operators this is not the case, as there is no syntax to provide explicit type parameters when calling operators. Thus the designer of a class library must be careful to choose a set of operators that will not result in irresolvable ambiguous method calls.

The various rules of section §MainSpec apply in a modified form to operators in generic classes. In particular, when a rule states that one of the arguments (or return type) of an operator in a *class-declaration* of a class `C` must be the type `C` itself, there is instead a requirement that the corresponding argument (or return type) of an operator in a *generic-class-declaration* of a class C must be a constructed type $C<T_1,...T_n>$ for some types $T_1,...,T_n$.

The exact rules are listed in the sections that follow.

18.4.8.1 Unary operators

The rules for unary operators in a generic class $C<V_1,...,V_n>$ are:

- A unary +, -, !, or ~ operator must take a single parameter of type $C<T_1,...,T_n>$ for some types $T_1,...,T_n$ and can return any type.
- A unary ++ or -- operator must take a single parameter of type $C<T_1,...,T_n>$ for some types $T_1,...,T_n$ and must return the same type.
- A unary `true` or `false` operator must take a single parameter of type $C<T_1,...,T_n>$ for some types $T_1,...,T_n$ and must return type `bool`.

18.4.8.2 Binary operators

A binary operator in a generic class $C<V_1,...,V_n>$ must take two parameters, at least one of which must be a type $C<T_1,...,T_n>$ for some types $T_1,...,T_n$.

18.4.8.3 Conversion operators

The rules for conversion operators in a generic class $C<V_1,...,V_n>$ between two types `S` and `T` are:

- `S` and `T` are different types.
- Either `S` or `T` a type $C<T_1,...,T_n>$ for some types $T_1,...,T_n$.
- Neither `S` nor `T` is `object` or an *interface-type*.
- No implicit or explicit reference conversions exist between `S` and `T` apart from the conversion due to the presence of this particular conversion operator.

18.4.8.4 Example

As a longer example, the following class combines delegates and operators to build a delegate-like class called `Action` that allows delegates (actions) to be pipelined using the >> operator, while keeping track of and checking the types generated in the intermediary stages of the pipeline. The `Action` class also supports the operators + to sequence delegates, as well as an implicit conversion operator to allow `PrimAction` delegates to be used as `Action` objects.

```
delegate U PrimAction<V,U>(V x)

class Action<V,U>
{
   private PrimAction<V,U> action;
   public Action(PrimAction<V,U> initial) { action = initial; }
   public U this[V value] { get { return action(value); } }
   public static Action<V,W> operator >> <V,U,W> (Action<V,U> a1,
                                                 Action<U,W> a2)
      { return new ComposedAction<V,W>(a1,a2); }
   public static Action<V,U> operator + <V,U> (Action<V,U> a1,
                                              Action<V,U> a2)
      { return new SequencedAction<V,V>(a1,a2); }
   public static implicit operator Action<V,U>(PrimAction<V,U> initial)
      { return new Action<V,U>(initial); }

   internal class ComposedAction<V,U,W> : Action<V,W>
   {
      private Action<V,U> a1;
      private Action<U,W> a2;

      private W fire(V x) { return a2[a1[x]]; }

      public ComposedAction(Action<V,U> _a1, Action<V,U> _a2)
          : base(new PrimAction<V,W>(this.fire))
        { a1 = _a1; a2 = _a2; }
   }

   internal class SequencedAction<V,U> : Action<V,U>
   {
      private Action<V,U> a1;
      private Action<V,U> a2;

      private U fire(V x) { a1[x]; return a2[x]; }

      public SequencedAction(Action<V,U> _a1, Action<V,U> _a2)
          : base(new PrimAction<V,U>(this.fire))
        { a1 = _a1; a2 = _a2; }
   }
}
```

The following example shows how the operators in the `Action` class can be used

```
public class Main
{
   private static void Print(string s) { Console.Write(s); return s; }

   private string string Copy(string s) { return s + s; }

   public static void Main() {
      Action<string,void> a = new PrimAction(Print);
      a["hello"];
      a[" "];
      a += new PrimAction(Copy) >> new PrimAction(Print);
      a["world\n"];
}
```

When run this program produces:

```
hello hello world
world
```

18.4.9 Instance constructors in generic classes

Instance constructors in a *generic-class-declaration* are "generic" in the sense that they must work for arbitrary *class-type-parameters*. Within the constructor this means that for a class `C` with *class-type-parameters* `V1,...,Vn` the "this" pointer is considered to have the constructed type `C<V1,...Vn>`, just as for any instance function member.

The *class-type-parameters* may be used in the argument expressions of a constructor initializer and in the field initializers of all non-static fields. The following example illustrates both of these:

```
class C<V>
{
   V[] x = new V[10];
   public C(V[] x) { this.x = x; }
}


class D<W> : C<W>
{
   public D(int x): base(new W[x]) {}   // Ok
   public D<int>(): base(new int[0]) {} // Error, syntax error
}
```

18.5 Generic structs

The rules for *class-declarations* and *generic-class-declarations* apply identically to *generic-struct-declarations*, except where the exceptions noted in §MainSpec for *struct-declarations* apply.

TODO: Do this later.

18.6 Generic methods

A *generic-method* is a method member that is abstract with respect to certain types. Generic methods can be used at two places: at a call site (§18.10.4) or to construct a delegate from the generic method (§18.10.11). Generic methods can only be used once an *instantiation* for the type parameters of the generic method is determined, either by inference (§18.10.5) or when explicitly specified by the user (§18.10.4, §18.10.11).

Generic methods are declared using the following syntax.

generic-method-declaration:
   generic-method-header method-body

generic-method-header:
   attributes$_{opt}$ *method-modifiers*$_{opt}$ *return-type member-name* < *method-type-parameters* >
   (*formal-parameter-list*$_{opt}$)

method-type-parameters:
   method-type-parameter
   mthod-type-parameters , *method-type-parameter*

method-type-parameter:
   identifier explicit-type-parameter-constraint$_{opt}$

Comment [DRS28]: *Implementation incompleteness: constraints on generic methods are not implemented.*

The *method-type-parameters* are in scope throughout the *generic-method-declaration*, and may be used to form types throughout that scope including the *return-type*, the *method-body*, and the *explicit-type-parameter-constraint* but excluding the *attributes*.

The following example finds the first element in an array, if any, that satisfies the given test delegate. Generic delegates are described in §0.

```
delegate bool Test<V>(V);
class Finder
{
   public V Find<V>(V[] inp, Test<V> p)
   {
      foreach (V x in inp)
         if (p(x)) return x;
      throw new InvalidArgumentException("Find");
   }
}
```

A generic method may not be declared `extern`.

A generic method must differ in formal signature from all other methods in its class. For the purposes of signature comparisons any *explicit-type-parameter-constraints* are ignored, as are the names of the *method-type-parameters* and any *class-type-parameters* that are accessible from the surrounding class definition, but the number of generic type parameters is relevant, as are the relative numeric positions of type-parameters in left-to-right ordering from the outermost declaration to the innermost.

18.6.1 Generic static methods

TODO: Do this later.

18.6.2 Generic virtual methods

Comment [DRS29]: *Generic virtual methods are not completely implemented in the GC# compiler, though they are in the GCLR.*

TODO: Do this later.

18.7 Generic interfaces

A *generic-interface-declaration* is a *generic-type-declaration* (§18.2.2.2) that declares a new generic interface.

generic-interface-declaration:
   attributes$_{opt}$ *interface-modifiers*$_{opt}$ `interface` *identifier* `<` *interface-type-parameters* `>`
   interface-base$_{opt}$ *interface-body* `;`$_{opt}$

It is identical to an *interface-declaration* except for the inclusion of the *interface-type-parameters*, which themselves follow an identical syntax to *class-type-parameters*:

interface-type-parameters:
   interface-type-parameter
   interface-type-parameters `,` *interface-type-parameter*

interface-type-parameter:
   identifier *explicit-type-parameter-constraint*$_{opt}$

An *interface-type-parameter* can be used as a *type-name* in certain parts of the interface, according to the following rules:

- An *interface-type-parameter* may be used to form a type in the signature of every member in the interface.
- The specification of each base interface of any *interface-declaration* is an *interface-type*, and in particular this might be a *constructed-interface-type*. It may not be an *interface-type-parameter* on its own, though it may involve the *interface-type-parameters* that are in scope.

18.7.1 Interface implementations

Interfaces may extend *interface-types*, and these may include *constructed-interface-types*. This may happen even for non-generic interfaces if they extend interfaces that themselves derive from *constructed-interface-types*.

18.7.1.1 Uniqueness of interface types

The set of base interface types of an interface (§18.12.3) must not contain two *constructed-interface-types* with the same *generic-interface-name* and different instantiations. This simplifies the implementation of dynamic dispatch mechanisms and prevents ambiguities arising amongst methods and ambiguities when generic interfaces are instantiated.

Comment [DRS30]: *Unimplemented: The check for this condition is not yet implemented.*

For example, consider the following interfaces:

```
interface I1<U> { ... }
interface I2<V> : I1<V[]> { ... }
interface I3<W> : I1<Object>, I2<W> { ... }
```

The *interface set* of `I3` is as follows:

```
{  I1<Object>, I1<W[]>, I2<W> }
```

This contains the interface `I1` at more than one instantiation, and is hence a compile-time error.

This restriction may be lifted in later versions of C#.

18.7.2 Explicit interface member implementations

Explicit interface member implementations may be provided for those interfaces which are *constructed-interface-types*. As in §MainSpec, an interface member implementation must be qualified by an *interface-type* indicating which interface is being implemented.

member-name:
  identifier
  interface-type `.` *identifier*

Comment [DRS31]: This leads to a little parsing problem – not strictly an ambiguity though. For example:

```
void IFoo<A,B,C>.m1()
void m2<A,B,C>()
void m2<A,B,C : IBar>()
```

The parser doesn't know if it parsing an explicit method impl or a generic method declaration until it reaches either the DOT (method impl), or an LPAREN (generic method) or something which lets it tell earlier, e.g. a COLON (generic method def). This is already sort of the case I believe, e.g. for long names

```
void NS1.NS2.IFoo.m1()
```

but there you only need finite lookahead. So if doing this with Yacc I think you need to parse the stuff on the left without committing to the two paths and decide which is which with a choice on the right.

This type may be a *constructed-interface-type*, as in the following example:

```
interface IList<V>
{
   V[] Elements();
}
interface IDictionary<K,D>
{
   D this[K];
   void Add(K x, D y);
}
class List<V>: IList<V>, IDictionary<int,V>
{
   V[] IList<V>.Elements() {...}
   V IDictionary<int,V>.this[int idx] { ... } // return the element at index
   void IDictionary<int,V>.Add(int idx, V y) { ... } // add y at the index
}
```

The *constructed-interface-type* must be a member of the *immediate base interface types* (§18.12.1.2) of the class C.

The algorithm in §MainSpec that maps class methods to the *set of base interfaces* (§18.12.1.2) of the class works essentially unchanged with the additional propagation of instantiations throughout the process.

18.8 Generic delegate declarations

A *generic-delegate-declaration* is a *generic-type-declaration* (§18.4) that declares a new generic delegate type.

generic-delegate-declaration:
  attributes$_{opt}$ *delegate-modifiers$_{opt}$* `delegate` *result-type* *identifier*
    delegate-create-type-parameters$_{opt}$
    delegate-invoke-type-parameters$_{opt}$
      `(`*formal-parameter-list$_{opt}$* `)` `;`

Comment [DRS32]: *Implementation incompleteness: Delegate-invoke type parameters are not yet implemented, either in the GC# compiler. A little work is also required in the .NET CLR to make them work, though the bulk of the work has essentially already been done for generic virtual methods. In MS-IL you would always create them via a "ldftn" or "ldvirtftn" on an uninstantiated generic method. I have included them in the design primarily for completeness and to make sure that delegates are as "expressive" as generic virtual methods, to which they are normally compared.*

It is identical to a *delegate-declaration* except for the inclusion of the *delegate-create-type-parameters*, which themselves follow an identical syntax to *class-type-parameters*, and the *delegate-invoke-type-parameters*, which follow the syntax and rules of *method-type-parameters*. If only one set of delegate type parameters are specified they are assumed to be *delegate-create-type-parameters*. If both sets of type parameters are given then the names of the type parameters in these sets must be distinct.

delegate-create-type-parameters:

  `<` `>`

  `<` *delegate-type-parameters* `>`

delegate-invoke-type-parameters: `<` *delegate-type-parameters* `>`

delegate-type-parameters:
  delegate-type-parameter
  delegate-type-parameters `,` *delegate-type-parameter*

delegate-type-parameter:
  identifier *explicit-type-parameter-constraint$_{opt}$*

Either kind of *delegate-type-parameter* can be used as a *type-name* in the rest of the signature of the delegate.

The definitions and rules for *combinable* delegates and *multi-cast* delegates carry over directly.

18.8.1 Specifying type parameters at delegate creation

Typically delegate type parameters are specified at the point where a delegate object is created. As specified in §18.10.11 the method provided for the delegate must have the signature that results after applying the specified instantiation throughout the delegate signature.

The following example specifies the arguments `<int,string>` at the point the instance of the delegate is created in the expression `new Test<int,string>(One)`. Note that the `One` method has the signature that corresponds to the signature required for `Test` delegates after the instantiation `<V` $\rightarrow$ `int, U` $\rightarrow$ `string>` is applied.

```
delegate bool Test<V,U>(V,U);

struct Pair<V,U> { V e1; U e2; }

class PairSearch
{

   public static U Find<V,U>(Pair<V,U>[] inp, Test<V,U> test)
   {

      foreach (Pair<V,U> p in inp)

         if (test(p.e1, p.e2)) return p.e2;

      throw new InvalidArgumentException("Find");
   }

   public static bool One(int n, string s)
      { return n == 1; }

   public static void Main()
   {

      Pair<int,string>[] a1 = { new Pair<int,string>(1,"one"),
                                new Pair<int,string>(2,"two"),
                                new Pair<int,string>(3,"three") };
```

```
        string three = Find<int,string>(a1, new Test<int,string>(One));
    }
}
```

Note that the same delegate object cannot be used at different types when all the type parameters are specified at the moment the delegate is created. That is, the delegate object cannot itself encapsulate a generic operation, only the generic operation specialized to a particular set of types. A new delegate object must be allocated for each different set of types.

18.8.1.1 Generic delegates and methods that return the type void

As specified in §18.2.3.1, when the keyword `void` is used as a type argument, the C# compiler replaces this with a type argument `System.Empty`, which is a value type containing no instance fields. However, this would appear to indicate that when a generic delegate type specifies that it returns a type parameter `V`, then instantiating `V` with `void` will then lead the delegate to require a method that returns the type `System.Empty`.

Comment [DRS33]: *Implementation incompleteness: using "void" as a type parameter is not implemented, and no System.Empty class currently exists in the BCL as far as I know.*

In this situation the C# compiler will insert fresh, uniquely named dummy methods and classes as necessary:

- If the delegate is to a static method, then the C# compiler will create a delegate to a fresh, dummy method returning a value of the `System.Empty` that calls the method returning nothing (i.e. `void`), and then returns a value of type `System.Empty`. The dummy method is placed in the class where the expression creating the delegate occurs.
- If the delegate is to an instance methods, then the C# compiler will create a a fresh, dummy class containing a single instance method returning a value of the `System.Empty` and a single assembly-level visibility private field. The instance field stroes the object being delegated to, and the instance methods calls the method being invoked for the particular object. This will return nothing (i.e. `void`), and the instance methods then returns a value of type `System.Empty`. The dummy class is placed as a nested class in the class where the expression creating the delegate occurs.

For example, the following shows the creation of two delegates to methods returning nothing (i.e. `void`).

```
delegate U Action<V,U>(V v);
class Print
{
    public static void Print(string s) { ... }
    public static void PrintLine(string s) { ... }
    public static void Main()
    {
        Action<string,void> a1 = new Action<string,void>(Print);
        Action<string,void> a2 = new Action<string,void>(PrintLine);
        a1("hello");
        a2("world");
    }
}
```

In general the insertion of these dummy methods and classes will be invisible. However, if later code then performs "stack-inspection" or other reflective actions then it is possible that the presence of these classes and methods may be detected.

Comment [DRS34]: Indeed, I strongly believe that the C# compiler should _always_ insert a dummy method or class when attempting to delegate to a method outside the current assembly or that has different permissions to the current context.

18.8.2 Delaying type parameters until delegate invocation

Comment [DRS35]: *Implementation incompleteness: See above – invoke-time type parameters are not supported yet in the GCLR.*

Delegates can be used to wrap generic operations, without committing to type parameters. Delegates of this kind must delegate to a generic method.

For example, the following example illustrates how you can write a component that is parameterized by a (generic) sort algorithm while late-binding the exact algorithm used, and without committing to the types at which you will need to invoke the algorithm.

```
interface int IComparer<U> { int Compare (U,U); }
delegate void Sorter<> <V> (V[] arr, IComparer<V> cmp);
class Algorithms
{
   public static void QuickSort<W>(W[] arr, IComparer<W> cmp) { ... }
   public static void MergeSort<W>(W[] arr, IComparer<W> cmp) { ... }
}


class MyComponent
{
   Sorter sorter;
   public void MyComponent(Sorter sorter)
   {  this.sorter = sorter; }
   public void Run()
   {
      int[] arr1 = { 5,4,3 };
      sorter<int>(arr1, new IntComparer());
      string[] arr2 = { "abc", "0", "987" }
      sorter<string>(arr2, new StringComparer());
   }
}


class Main
{
   public static void Main()
   {
      MyComponent c1 = new MyComponent(new Sorter(Algorithms.QuickSort));
      c1.Run();
      MyComponent c2 = new MyComponent(new Sorter(Algorithms.MergeSort));
      c2.Run();
   }
}
```

It is possible to combine the two kinds of delegate type parameters. For example:

```
delegate void F<U> <V> (V[] v, U[] u);
class Algorithms
{
   public void m1<W>(W[] arr1, int[] arr2) { ... }
   public void m2<W>(W[] arr1, string[] arr2) { ... }
   public void m3<W>(W[] arr1, int[] arr2) { ... }
   public void m4<W>(W[] arr1, string[] arr2) { ... }
}
class MyComponent
{
   F<int> f1;
   F<string> f2;
   public void MyComponent(Sorter sorter)
   {  this.sorter = sorter; }
   public void Run()
   {
      int[] arr1 = { 5,4,3 };
      string[] arr2 = { "abc", "0", "987" }
```

```
        f1<int>(arr1, arr1);
        f1<string>(arr2, arr1);
        f2<int>(arr2, arr1);
        f2<string>(arr2, arr2);
    }
}
class Main
{
    public static void Main()
    {
        MyComponent c1 = new MyComponent(new F<int>(Algorithms.m1),
                                         new F<string>(Algorithms.m2));
        c1.Run();
        MyComponent c2 = new MyComponent(new F<int>(Algorithms.m3),
                                         new F<string>(Algorithms.m4));
        c2.Run();
    }
}
```

18.8.3 Constraints and generic delegate types

Both the *delegate-create-type-parameters* and the *delegate-invoke-type-parameters* may specify constraints for their respective type parameters. The rules for delegate creation (§18.10.13) and delegate invocation (§18.10.4.3) specify the relevant conditions that must hold to satisfy these constraints.

TODO: need some discussion on the additional operations that delegates support (BeginInvoke EndInvoke etc).

18.9 Constraints

Each *type-parameter* (including *class-type-parameters*, *method-type-parameters* etc.) may be qualified by an *explicit-type-parameter-constraint*. The specification of an explicit constraint is optional. If given, a constraint is a *reference-type* that specifies a minimal "type-bound" that every instantiation of the type parameter must support. This is checked at compile-time at every use of the corresponding generic class, interface, method, delegate or struct.

explicit-type-parameter-constraint:
  reference-type

extra-type-parameter-constraints:
  | *identifier explicit-type-parameter-constraint further-type-parameter-constraints*

further-type-parameter-constraints
  empty:
  , *identifier explicit-type-parameter-constraint*

TODO: None of the examples actually illustrate the use of class constraints, just interface constraints

A single constraint is typically specified immediately after a type parameter. Values of the type constrained by the type parameter can be used to access the instance members, including instance methods, specified in the constraint.

```
interface IPrintable { void Print() ; }
class Printer<V : IPrintable>
{
    void PrintOne(V x) { x.Print(); }
}
```

Constraints may also be specified after all type parameters have been introduced. This allows multiple constraints to be specified for a single type parameter.

```
interface IPrintable { void Print() ; }
class Printer<V | V : IPrintable, V : IDisposable>  { ... }
```

Constraints may involve the type parameters themselves. This is used when the constraint specifies an operation that involves passing or returning values involving the type parameter. For example:

```
interface IComparable<V> { int CompareTo(V); }
class Sorter<V | V : IComparable<V>>  { ... }
```

Constraints may even involve the class, interface or method for which they are acting as a constraint:

```
interface I<V : I<V>>  { ... }  // A strange, but legal constraint
```

Constraints may also be class types, e.g. abstract base classes:

```
abstract class Printable { abstract void Print(); }
interface I<V : Printable>  { ... }
```

"Different" constraints may be attached to the type parameters declared on generic static methods within a generic class. This is because generic static methods must declare their own type parameters and do not automatically acquire the type parameters of the enclosing (generic) class. For example:

```
interface IPrintable { void Print() ; }
class List<V>
{
   ...
   static void Print<V : IPrintable>(List<V> list) { ... }
}
```

Constraints may not themselves be a type parameter:

```
class Extend<V, U | U : V> { ... }  // Error, a type parameter may not be
                                    // used as a constraint
```

Comment [DRS36]: I think the compiler currently chokes onthis, giving an AV or a bad error message.

Explicit constraints are most useful in two circumstances:

- Generic classes: to require class type parameters to support simple functionality such as an "equals" method or a "less than" method to implement an ordering, especially when the class implements an abstract data type that makes no sense without these methods (e.g. balanced binary trees require an ordering on the type used as the key). Note, however, that in many circumstances it is unreasonable to expect instantiating types to support exactly the right base classes, and it may be more appropriate and flexible to use "provider" functions (e.g. System.Collections.IHashCodeProvider or System.ICollections.IComparer) to provide the necessary functionality.
- Generic methods: constraints can be used to help define generic methods that "plug together" functionality provided by different types, thus defining "generic algorithms". This can also be achieved by subclassing and runtime polymorphism, but static, constrained polymorphism can in many cases result in more efficient code, more flexible specifications of generic algorithms, and more errors being caught at compile-time rather than run-time. However, constraints need to be used with care and taste, as code that does not implement the constraints will not be easily usable in conjunction with the methods.

In either case, constraints are most useful in the context of defining a *framework*, i.e. a collection of related classes, where it is an advantage to ensure that a number of types support some common signatures and/or base types. Constraints also allow types to be related when they may not been explicitly related via inheritance and/or implementation.

The .NET Framework Collection Classes do not make heavy use of explicit constraints. Rather, the constraints on type parameters are left implicit in the few cases were they are used. For example, System.Collections.Generic.SortedArray requires that either the user of the class provides a comparison delegate, or the objects placed in the class all support the IComparable interface. A runtime error will occur if this is not the case. Nearly all objects in the .NET Framework Libraries support the IComparable interface. Thus the .NET Framework Collection Classes strike a balance between the rigidity of a static constraint system and the ease-of-use of a more dynamic system.

Comment [DRS37]: *NOTE: I expect this to be true. I don't know if we want it in this spec, but it kind of seems worth mentioning as readers may be asking themselves if they need to know this section or not.*

18.9.1 Satisfying Constraints

Every time a type is supplied for a type parameter, e.g. at a method call to a *generic-method-declaration* or when building a *constructed-class-type* out of a *generic-class-declaration*, the actual type parameters must ***satisfy*** the constraints on the *type-parameters* of the declaration being used.

For example, the following constraint is satisfied by all of the types declared below it.

```
interface IPrintable { void Print() ; }

class C1 : IPrintable
{
    ...
    public virtual void Print() { ... }
}

class C2 : IPrintable
{
    ...
    public abstract void Print() { ... }
}

struct C3 : IPrintable
{
    ...
    public void Print() { ... }
}

class C4 : IPrintable
{
    ...
    void IPrintable.Print() { ... }
}
```

This means the following code fragments are accepted by the compiler:

```
public class Printer<V : IPrintable>
{
    ...
    public void Print(V x) { ...x.Print(); ...  }
}

... C1 x = new C1(); Printer<C1> p1 = new Printer<C1>(); p1.Print(x); // OK

... C2 y = new C2(); Printer<C2> p2 = new Printer<C2>(); p2.Print(y); // OK

... C3 y = new C3(); Printer<C3> p3 = new Printer<C3>(); p3.Print(y); // OK

... C4 y = new C4(); Printer<C4> p4 = new Printer<C4>(); p4.Print(y); // OK
```

However, the following type does not satisfy the constraint because the class does not explicitly implement the interface:

```
public class C5
{
    ...
    public void Print() { ... }
}

public static void Main()
{
    C5 y = new C5();

    Printer<C5> p5 = new Printer<C5>(); // Error

    p5.Print(y);
}
```

18.9.2 When and how an instantiation satisfies its constraints

A *constructed-class-type* `C<`*inst*`>` for a *generic-class-declaration* C with *class-type-parameters* `<`V_1`,...,`V_n`>` is only ***valid*** under certain conditions. In particular, for each V_i that is constrained, e.g. by a *constructed-constraint* `B<`T_1`,...,`T_m`>`, first form a new *constructed-constraint* `B<`S_1`,...,`S_m`>` by applying the instantiation *inst* to each type `Tᵢ` to form `Sᵢ`. Then the individual type that corresponds to `Vᵢ` in the instantiation *inst* must satisfy this constraint.

A type `T` (whether reference or a value type, constructed or non-constructed) ***satisfies a constraint*** S if there is an implicit non-user-defined conversion from the type `T` to the type `S`. The conversion may be a boxing conversion, which means that value types may satisfy constraints given by interface types, if the value types implement those interface types.

Similarly, constraints must be satisfied for *constructed-interface-types*, *constructed-delegate-types*, *constructed-struct-types*, *generic-method-invocations* and *generic-delegate-invocations*.

18.10 Expressions and statements

The semantics of many expressions and statements is affected by the presence of constructed types, variable types and their corresponding values. This section describes these effects. In particular the following expressions are affected:

- *member-accesses*
- *invocation-expressions*
- *element-access*
- *this-accesses*
- *base-accesses*
- *post-increment-expressions*
- *post-decrement-expressions*
- *new-expressions*
- *typeof-expressions*
- *sizeof-expressions*
- *local-variable-declarations*
- *foreach-statements*
- *specific-catch-clauses*

18.10.1 Expression classifications

The expression classifications of §MainSpec are extended in the following ways:

- Whenever expressions have a type, e.g. values, variables, property accesses, event accesses and indexer accesses, the type may involve *constructed-types*, and, if the expression occurs within a *generic-declaration* the type may also involve *type-parameters*.
- When an expression is itself a type, e.g. is an operand of the `as` operator, the type may be a *constructed-type*, and, if the expression occurs within a *generic-declaration,* the type may also involve any *type-parameters*of that declaration .
- In addition, the *class-name* associated with a *generic-class-declaration* may be used to form a type in the following situations: on the left hand side of a member-access and as the operand of the `typeof` operator. This is in order to access the static members contained within a *generic-class-declaration* and to pass an (uninstantiated) generic type to reflection libraries. The names associated with generic interface, struct and delegate types can be used in a similar fashion.
- When an expression has an associated instance expression, e.g. when the expression is an *invocation-expression* (§MainSpec, §18.10.4) or a *property-access-expression* (§MainSpec), the instance expression may have a type that is a *constructed type*.

- When an expression is classified as a method group, e.g. in an *invocation-expression* (§MainSpec, §18.10.4) or a *delegate-creation-expression* (§MainSpec, §18.10.13), then each method in the method group will be paired with a (possibly constructed) type that indicates the *reference-type* or *struct-type* related to the position of this method within an inheritance hierarchy that may contain constructed types. Namely, the method group is a subset of the *set of all function members* of a class, interface or struct (§18.12.2, §18.12.3), which annotates members with type information according to the inheritance hierarchy.

  The use of these pairs is explained in the rules for *invocation-expressions* (§18.10.4) and *delegate-creation-expressions* (§18.10.13).

- Similarly, when an expression is a property, event or indexer access, then the member is paired with a (possibly constructed) type that indicates the *reference-type* or *struct-type* related to the position of this method within an inheritance hierarchy that may contain constructed types.

18.10.2 Member lookup

The rules for member lookup are unchanged (§MainSpec) with the following exceptions:

- Member lookups for classes return member groups that contain pairs of members and governing types, drawn from the *set of all function members of a class*, as defined in §18.12.2 and elsewhere.
- Similarly for member lookups for interfaces and structs.

In addition, member lookup for an expression that has a type that is a type variable `V` returns the union of all the member groups from all the constraints of `V`. These will again be groups containing pairs of members and governing types.

For instance, in the following example the expression "`x`" has type `V`. The method group for the expression "`x.Print`" will contain two members, the first being the type `IPrintable` twinned with the method "`void Print();`", the second the type `IBatik<string>` twinned with the method "`U Print(U);`". The second method will be the one selected by the process of overload resolution.

```
public void IPrintable { void Print(); }
public void IBatik<U> { U Print(U); }
public void Print<V | V: IPrintable, V: IBatik<string>>(V x)
{
    x.Print(“abc”);
}
```

18.10.3 Member access

Constructed types are not allowed to appear as types on the left of a *member-access*. An uninstantiated generic type may appear on the left of a *member-access*. In this case the accessed member must be static.

18.10.4 Invocation expressions

An *invocation-expression* is used to invoke a method. At the point of invocation a *generic-instantiation* may be specified:

invocation-expression:
  primary-expression generic-instantiation$_{opt}$ `(` *argument-list*$_{opt}$ `)`

Comment [DRS38]: *This introduces the one place where an ambiguity arises in the whole grammar, in particular*
`g(f<a,b>(d))`
This can be parsed either as a generic method application or an expression. It should be resolved as the former, which in theory means some existing code may no longer compile. But the case above is SO UNLIKELY in existing code I think it's OK.

As before, the *primary-expression* of an *invocation-expression* must be a method group or a value of a *delegate-type*.

18.10.4.1 Instance method invocation

An instance method invocation has:

- An instance expression *obj* associated with the *primary-expression.*

- A method group arising from the *primary-expression*. This method group will contain pairs of methods and governing types as explained in §18.10.1.
- An optional *generic-method-instantiation* specified in the invocation expression.

Resolving an instance method invocation involves the following steps:

1. Incorporate the type of *obj* with the governing types of the method group, which initially do not take the detailed type of *obj* into account. In particular, if *obj* has type `D`<*inst1*> and the governing type in the method group is `C`<*inst2*>, then replace the governing type with `C`<*inst*> where *inst* is the composition of *inst1* and *inst2*.

   Note that:

   - `C` and `D` may be generic classes taking type parameters $U_1,...,U_m$ and $V_1,...,V_n$ respectively.
   - Under our notational conventions this also covers the case where `D` is non-generic and *inst1* is the empty instantiation, and also where `C` is non-generic and *inst2* is the empty instantiation.
   - Note also that `C` and `D` may be identical, or the class `C` will appear somewhere in the class hierarchy above `D`.
   - The type `C`<*inst2*> will appear in the *base type set* for the class `D`.
   - The type `C`<*inst*> will appear in the *base type set* for the type `D`<*inst1*>.
   - The instantiation *inst1* will map type parameters $V_1,...,V_n$ of `D` to types that may involve the *type-parameters* that are accessible at the point where the overall *invocation-expression* occurs. These types will contain no other type parameters.
   - The instantiation *inst2* will map type parameters $U_1,...,U_m$ of `C` to types that may involve the *type-parameters* $V_1,...,V_n$, but no other type parameters.

2. Apply *method type parameter inference* (§18.10.5). In particular this is applied to each method in the method group. The process may exclude some methods but will yield exactly one *method-type-instantiation* for those methods that remain, or else report an error.

   If a *method-type-instantiation* has already been specified in the invocation expression then this will be used for every method in the method set, and those methods for which the *method-type-instantiation* is not valid (because it does not satisfy the appropriate constraints or specifies the wrong number of type parameters) are excluded.

   After *method type parameter inference* the method group can now be considered to contain triples consisting of methods, governing types and method instantiations.

3. Apply overload resolution (§MainSpec, §18.10.6) to the method group that results from step 2. In the absence of an error this will yield a single method, a single governing type and a single method instantiation.

We now explain how the method, governing type and method instantiation are combined to determine the exact instantiations of type parameters used for the expression, and how the return type of the expression can be determined.

Let us suppose that the steps above produce a method `R` *m* $(A_1,...,A_q)$, a governing type `C`<*inst*> and a method instantiation *minst*. Then the ***overall class type instantiation*** is *inst*, and the ***overall method type instantiation*** is *minst*.

Comment [DRS39]: *Testing incompleteness: calling generic methods in generic classes is not particularly well tested, especially in combination with overloading. This should be tested thoroughly especially when method type inference is also completed.*

Note that:

- `C` may be a generic class taking type parameters $V_1,...,V_n$.
- *m* may be a generic method accepting type parameters $W_1,...,W_p$.
- Under our notational conventions this also covers the case where `C` is non-generic and *inst* is the empty instantiation, and/or *m* is non-generic and *minst* is the empty instantiation.

- The instantiations *inst* and *minst* will map type parameters `V`$_1$`,...,V`$_n$ and `W`$_1$`,...,W`$_p$ respectively to types that may involve the *type-parameters* that are accessible at the point where the overall *invocation-expression* occurs. These types will contain no other type parameters.

Furthermore:

- The constraints that *inst* must satisfy are found by substituting *inst* through the formal constraints associated with `V`$_1$`,...,V`$_m$.
- The constraints that *minst* must satisfy are found by substituting the combination of *inst* and *minst* throughout the formal constraints associated with `W`$_1$`,...,W`$_p$.
- The actual argument types and return type can be found by substituting the combination of *inst* and *minst* throughout `A`$_1$`,...,A`$_q$ and `R`.

Comment [DRS40]: *Implementation incompleteness: method type parameter constraints are not yet implemented.*

In combination with the instantiations and eventual method called via virtual dispatch, the overall class and method instantiations determine the exact types assigned to type parameters when the code for the invoked method is executed.

TODO: explain virtual dispatch in the presence of inheritance hierarchies that involve constructed types. Do this after the virtual dispatch section of the main spec is written. Essentially the object records the actual types for all the type parameters up the inheritance hierarchy, and the appropriate type parameters are used as the "actual" class type parameters.

18.10.4.2 Static method invocation

Static method invocation proceeds along essentially the same lines as instance method invocation, except that no governing types need be considered. Both overload resolution and method type parameter inference are applied.

18.10.4.3 Delegate invocation

Delegate invocation may accept type parameters. Inference is applied according to essentially the same rules as instance method invocation. No overloading resolution is applied since it is not required.

18.10.5 Type parameter inference

The aim of type parameter inference is to allow programmers to omit type instantiations in certain commonly occurring, highly predictable situations. Type parameter inference is applied when a generic method or operator is called but no explicit instantiation is given at the call site, or when an instance constructor on a generic class is called with no explicit instantiation.

For example, given the class and statemements below, the type instantiations "string" and "int" are essentially obvious given the arguments to the constructor.

```
class List<V> : ICollection
{
   List(V[] x1) { ... }
}
...  ICollection c1 = new List<string>(new string[] { "abc", "def" });
...  ICollection c2 = new List<int>(new int [] { 3, 4 });
```

Type parameter inference allows these arguments to be omitted:

```
class List<V> : ICollection
{
   List(V) { ... }
}
...  ICollection c1 = new List(new string[] { "abc", "def" });
...  ICollection c2 = new List(new int [] { 3, 4 });
```

Type parameter inference may be ambigious, in which case an error is reported and the user must insert explicit type parameters.

Type parameter inference consists of an algorithm that attempts to determine an appropriate instantiation for the method or constructor call. The algorithm is applied to each candidate entry in a method group, and either excludes the method, determines a unique method instantiation for the call, or reports an error.

We assume the entry in the input method group contains

- a method *m*; and
- a governing type `C<`*inst*`>`.

We also assume *m* is a generic method accepting type parameters $W_1,...,W_n$. Note that if any of the names of these type parameters conflict with those currently in scope at the point where type inference is applied they must be renamed appropriately. We also assume the method has formal argument types $A_1,...,A_m$ and a return type `R`, and the actual argument expressions that match these have types $X_1,...,X_m$.

Inference applies the following steps:

1. First, generate more refined versions of the expected actual types by applying *inst* to each formal argument type $A_1,...,A_m$. This produces expected argument types $E_1,...,E_m$. In the case of generic instance methods this eliminates the class type parameters from the inference process and allows the information from the instance expression to be used in the inference process.
2. For each actual argument/formal argument pair, generate a *candidate type instantiations* as specified below. Each such type instantiation specifies bindings for none, some or all of $W_1,...,W_n$.
3. Take the union of these type instantiations, and choose a unique sub-instantiation of the union that is *consistent*, *complete* and *valid* as defined below. Not all the individual bindings generated in step 2 need be used.
    a. If no such choice exists, then a compile-time error is reported. In rare cases it may still be possible for a programmer to provide explicit type arguments to the method invocation that will get around this problem, and the compiler will report when this may be possible. In particular, it may be possible if choices of subsets exist that are *complete* and *consistent* but not *sufficient*, as defined below.
    b. If more than one choice exists, then a compile-time error is also reported and the ambiguity reported.

18.10.5.1 Generating candidate type instantiations

Given an expected argument `E` and an actual argument type `X` we generate a type instantiation *A*, which is a set `{ V1 -> T1, ... }` of ***bindings*** of types to type parameters. Note that, on entry to this procedure, `E` may be an "out" of "ref" argument but in subsequent recursive invocations of step (2) it will always be just a type.

Comment [crusso41]: One reviewer said: Perhaps it would be better to have an outer loop for dealing with byref arguments, and then descending into pure types.

1. Set *A* to be the empty set.
2. Recursively perform the following:
    a. If `E` contains no type parameters at all then add nothing to *A*.
    b. If `E` is a type parameter `W` drawn from $W_1,...,W_n$ then add the binding (`W -> X`) to *A*.
    c. If `E` is a constructed type $C<E_1,...,E_n>$ then determine the unique type with the form $C<X_1,...,X_n>$ for some $X_1,...,X_n$ such that there exists an implicit conversion from `X` to this type.

       Then for each E_i/X_i pair run step 2 with `E` set to E_i and `X` set to X_i, with the restriction that only the identity conversion should be considered at step 2(c) for this and subsequent recursive invocations.

        i. If no such type $C<X_1,...,X_n>$ exists then a compile-time error is reported, as it will not be possible to find a suitable type instantiation for the call site. Alternatively, exit

Comment [crusso42]: One reviewer asked: *It is too early to report an error here since all you are doing is generating candidate assignments. Perhaps you should just exit., with a perhaps partial assignment.*
The fact that the assignment is incomplete, or invalid will be caught later?

the procedure and report an error at a later stage, as no valid type assignment will be possible.

ii. $C<X_1,...,X_n>$ will be either amongst the *set of base types* (§18.12.3) of X, or else there will be an implicit user-defined conversion between X and $C<X_1,...,X_n>$. It is not possible that more than one such type may exist. In particular, the restrictions on user-defined conversions (especially the lack of co-variance between constructed types) and the fact that classes and interfaces may not implement/extend interfaces at more than one instantiation prevent this from occurring.[2]

d. If E is an array type E_1[] then X must also be an array type X_1[], otherwise report a compile-time error. Run step 2 with E set to E_1 and X set to X_1, without any particular restriction on the kinds of conversions used at step 2(c) unless a restriction already exists. Similarly with other array types.

e. If E is an "out" argument of type E_1 then run step 2 with E set to E_1 and X set to X_1, with the restriction that only the identity conversion should be considered at step 2(c) for this and subsequent recursive invocations.

3. Return the set *A* as the candidate type instantiation for the argument.

Comment [crusso43]: This algorithm is rather different from matching as we would implement it. In particular , an implementation should apply the substitution computed at step i to Ei+1 to detect errors earlier? You could also reject the candidate type assignment earlier by checking that A is actually E (or implicitly convertible).

18.10.5.2 Consistency, completeness and validity of type instantiations

A type instantiation is *consistent* if every type parameter is assigned at most one type.

A type instantiation is *complete* if it assigns a type to each and every type parameter $W_1,...,W_n$.

A type instantiation is *valid* if it both satisfies the constraints (§18.9.2) for the type parameters $W_1,...,W_n$ and after applying the type instantiation to the expected argument types $E_1,...,E_m$, (producing types $F_1,...,F_m$), the actual argument types provided are sufficient to constitute a successful method call according to the standard rules for method calls. For example, in the absence of ref-parameters and `params` types, this means that an implicit conversion must exist from each of the actual argument types $X_1,...,X_m$ to each expected argument type $F_1,...,F_m$.

18.10.5.3 Inference and "params" argument arrays

If the method has a "params" argument then this is only taken into account if one or more actual arguments are provided for that argument. The element type of the `params` array is then duplicated once for each argument and the inference algorithm proceeds as if there was an equal match between formal argument types and actual argument types.

For example

```
class M
{
  static void f<V,U>(V x, params U[] ps) { ... }
}
class Application
{
  public static void Main()
  {
```

[2] If this restriction were lifted then we would have to either (a) backtrack or (b) choose one of the types at random or (c) return with an empty type assignment.

```
      M.f<string,int>("abc")    // Ok, no inference needed
      M.f("abc")                // Error, type cannot be inferred
      M.f("abc", 1)             // Ok, infer <V,U> = <string,int>
      M.f("abc", "def", "hij")  // Ok, infer <V,U> = <string,string>
      M.f("abc", "def", 3)      // Error, no valid instantiation determined
      M.f("abc", (object) "def", (object) 3) // Ok, <V,U> = <string,object>
      M.f("abc", "def", (object) 3)          // Ok, <V,U> = <string,object>
      M.f("abc", (object) "def", 3)          // Ok, <V,U> = <string,object>
    }
  }
```

18.10.5.4 Examples

The following code shows several examples of the type parameter inference algorithm and its results:

```
  class List<V>
  {
     List(V) { ... }

     public static List<V> f1<V>(V x) { ... }

     public static List<V> operator + <V> (List<V> x, List<V> y) { ... }

  }
  class M
  {
    public static List<V> f1<V>(List<V> x) { ... }

  }
  class Application
  {
    public static void Main()
    {
      List<int> x1 = M.f1<int>(new List<int>(3)); // Ok, no inference
      List<int> x2 = M.f1(new List(3));           // Ok, <V> = <int>

      List<int> x3 = List.f1<int>(x2);      // Ok, no inference
      List<int> x4 = List.f1(x2);           // Ok, inferred <V> = <int>
      List<int> x5 = x1 + x2;               // Ok, inferred <V> = <int>

    }
  }
```

The following code shows several more examples where the inference process reports errors:

```
  class C { ... }

  class M
  {
      public static U f2<V,U>(V x) { ... }

      public static void f3<V>(out V x, out V y) { ... }
  }
  class Application
  {
    public void Main()
    {
      M.f2(new C()); // Error, no type instantiation could
                          // be inferred for the type parameter ‘U’

      int x;
      long y;

      M.f3(x,y); // Error, no valid type instantiation could
                      // be inferred for the type parameter ‘V’
                      //    Considered V = int and V = long
    }
  }
```

18.10.5.5 Cases where ambiguities are possible

It is possible to write generic methods which will cause the inference algorithm described above to report an error even when an acceptable explicit instantiation may exist. These cases almost always involve the use of "naked" type parameters as arguments:

```
class C { ... }

class D { ... }

class M
{
   public static V f1<V>(V x, V y) { ... }
}

class Application
{

   public static void Main()
   {
      ... M.f1(new C(), new D()); // Error, no valid type instantiation
                                  // determined. An explicit type
                                  // parameter may be required when calling
                                  // f1,   e.g. V = Object
   }

}
```

It is also possible the inference algorithm will infer type parameters that are more precise than those desired. For example:

```
class C { ... }

class D : C { ... }

class List<V> { ... }

class M
{
   public static List<V> f1<V>(V x) { .. }

   public static void f2(List<C> x) { .. }
}

class Application
{

   public static void Main()
   {
     M.f2(M.f1<C>(new D()));  // Ok

     M.f2(M.f1(new D())); // Error, cannot pass value of type
                          // List<D> to value of type List<C>. Constructed
                          // types are invariant. Note that "D" in List<D>
                          // is an inferred type.  An explicit type
                          // parameter may be required when applying method
                          // f1.

}
```

In rare cases, generic methods may dynamically examine their type parameters (e.g. via `typeof` or via reflection). In these cases the exact type parameter may be very important, even if this is not apparent from the signature of the method. While this kind of programming is not common, it can on occasion be useful.

For example:

```
class M
{
   public static void f1<V>(V x)
   {
      if (typeof(V) != typeof(object)) throw new ArgumentException();
       ...
   }
}
```

```
class Application
{
   public static void Main()
   {
      M.f1<object>("abc");  // Ok
      M.f1<string>("abc");  // Would raise an exception
      M.f1("abc");  // Error, exception, <V> = <string> inferred
   }
}
```

18.10.5.6 Rationale for not "guessing" a "best common supertype"

The inference algorithm uses one-by-one parameter matching to determine candidate type instantiations. In particular, it does not "guess" instantiations that are not immediately obvious from matching the types of the arguments against the types expected by the method.

```
class C { ... }
class M
{
  public static void f<V,U>(V x, params U[] ps) { ... }
}
class Application
{
  public static void Main()
  {
    f("abc", "def", 3);     // Error, no valid instantiation easily inferred
  }
}
```

The rationale for this can be seen from the fact that it is a similar rule to that used for the conditional operator `(?:)`. In particular, the type for the expression `(b ? x : y)` is *either* the type of x or the type of `y`, never a proper supertype of both.

In particular, this greatly simplifies type inference in the presence of user-defined implicit conversions, for otherwise the compiler would have to consider *every* type that may be reachable by implicit conversions, including conversions in classes never before referenced.

18.10.5.7 Resolving inference ambiguities by explicit instantiations

Inference ambiguities can always be resolved by giving an explicit instantiation. If no instantiation is desired, i.e. overloading is occurring between a generic and a non-generic method, then no instantiation should be given, as explained in §18.10.6.1.

```
class C { ... }
class M
{
  public static void f<V>(V x) { ... }
  public static void f(object x) { ... }
}
class Application
{
  public void Main()
  {
      M.f<object>("abc");    // Ok, no inference needed
      M.f("abc");            // Ok, no inference needed
   }
}
```

18.10.5.8 Constructor type parameter inference

Type parameters are inferred at calls to instance constructors in much the same way as for calls to static methods. In particular, if the instance constructor is treated as a generic static method accepting the same type parameters as the class, then the inference process can be used essentially without modification.

For example:

```
class C<V>
{
   public C(V) { ... }
}
class Application
{
  public void Main()
  {
    C<object> x1 = new C<object>(4);  // Ok, no inference
    C<int> x1 = new C(4);             // Ok, inferred <V> = <int>
    C<object> x1 = new C((object) 4); // Ok, inferred <V> = <object>
  }
}
```

18.10.5.9 Inference for expressions of rigid type

For C# designers: I think we can make quite a late decision if this is what we want in the spec or not. I think there are many situations where this will save a fair bit of typing, but too much inference may confuse users and/or intellisense.

However, this particular bit of inference may make Intellisense work _much_ better. Users tend to write code-left-to-right (how observant of me..) so will often type

*Hashtable<int,string> table = new Hashtable****

*With the cursor now at the *** they have the option of typing "<" and getting "<int,string>" offered as a completion, OR typing "(" and getting the overloaded signatures offered as a completion with the substitutions made on the signature. Once I learnt to do the latter I would always use it, but the latter is only really valid for all signatures if the source language includes a rule like the one below.*

Basically, my intuition says that left-to-right inference rules like the one below will be more useful for intellisense than right-to-left rules like the general method-type-parameter inference up above.

When invocation and construction expressions are used at particular points within the syntax of C# statements and expressions, extra formal argument/actual argument information is supplied to the method type inference (§18.10.5) and constructor type inference (§18.10.5.8) processes. In particular if the formal return type of the candidate method for the invocation is `R` (or, respectively, the formal type of the value being constructed is `R`), then:

Comment [DRS44]: *Implementation incompleteness: Nothing in the entire inference section is implemented*

- If an expression occurs on the immediate right of a *variable-declarator* (§8.5.1 – Local variable declarations) or a field variable initializer (§10.4.5 *variable-initializer*s) where the declared type of the variable is `T`, then an extra actual/formal argument pair `T`/`R` is added to the inference processes specified in §18.10.5.1. However, for the interpretation of this pair the matching process is reversed, as the formal type `R` will contain unknown type variables, and `R` may be a subtype of `T` instead of vice-versa.
- If the expression is one of the expressions in an array construction expression (§7.5.10.2 – Array creation expressions), and the type of the array creation expression is an array type `T[...]` (for some array type, e.g. `T[,]`), then an extra expected/formal argument pair `T`/`R` is added to the inference processes specified in §18.10.5.1.

For example:

```
public class C<V>
{
   public C() { ... }
}
public class Pair<V,U>
{
   public Pair(V x, U y) { ... }
}
public class Application
{
  public void Main()
  {
    C<object> x1 = new C<object>();  // Ok, no inference
    C<int> x2 = new C();             // Ok, inferred <V> = <int>
    C<object> x2 = new C();          // Ok, inferred <V> = <object>
    C<int>[] arr1 =
        new C<int>[] { new C(), new C(), new C() };
    Pair<int,object>[] arr2 =
        new Pair<int,object>[] { new Pair(1,2), new Pair(2,"a") };
  }
}
```

Using this information allows type inference to determine the correct type in a number of useful situations, and never causes existing inference to fail where type checking would have succeeded overall.

```
class C { ... }
class D : C { ... }
class List<V> { ... }
class M
{
   public static List<V> f1<V>(V x) { .. }
}
class Application
{
   public static void Main()
   {
     List<C> list = M.f1(new D())); // Ok, <V> = <C> inferred.
   }
}
```

18.10.6 Overload Resolution

The overload resolution rules defined in §MainSpec apply to both the members of a constructed type (§18.2.2) and the functionality available via a constraint (§18.9). Instead of beginning with the members of a class (including its inherited members), the process begins after *method type parameter inference* has been applied, resulting in a method group that contains triples of:

- a method *m*;
- a governing type C<*inst*>; and
- a method instantiation *minst*.

For each entry in the method group, the combination of *inst* and *minst* is applied to the signature of the method, forming a group of methods. The standard method overloading rules of §MainSpec are then applied.

For example:

```
public class Foo { }
public class C<U,V>
{
```

Comment [DRS45]: *Implementation incompleteness: overloading on generic methods with both some type parameters and no type parameters is not supported. Furthermore, the CLR does NOT currently consider the number of type parameters signficiant for a generic method when determining signature matching. This is a current work item for MSR though we may not get around to it before code transfer.*
Also, I don't think I test for the rules above in the GC# compiler or produce any errors when the rules above are violated.
Also, we must also test the cases where we override methods, e.g. to make sure you can't override with a different signature, including a different constraint.

```
        ...
        public void m(U x, V y, object z) { ... }   // method A
        public void m(U x, V y, string z) { ... }   // method B
        public void m(object x,Foo y, V z) { ... }   // method C
    }

    public void Main
    {
       C<string,string> c1 = new C<string,string>() ;
       c1.m("a", "b", new object());          // calls method A
       c1.m("a", "b", new Foo());             // calls method A
       c1.m("a", "b", 1);                     // calls method A, boxing the int
       c1.m("a", "b", "c");                   // calls method B
       c1.m(new object(), new Foo(), "abc")   // calls method C
       c1.m(new Foo(), new Foo(), "abc")      // calls method C
       c1.m(new Foo, new Foo(), new Foo())    // error in last parameter
    }
    // Note that C<string,string> is considered to have methods:
    //    public string m(string x,string y, object z) { ... }   // method A
    //    public string m(string x,string y, string z) { ... }   // method B
    //    public string m(object x,Foo y, string z) { ... }   // method C
```

In the above example, the type being activated is `C<string,string>`, and because all the candidate methods come from this class, the governing type for each is `C<string,string>`. The full set of methods available for this type is determined by applying the instantiation `<string,string>`, which is the instantiation associated with the governing type `C<string,string>`. This makes the individual resolutions performed above apparent using the standard rules for overload resolution.

TODO: Longer example where overloads span across a hierarchy.

One exception applies:

- If a tie occurs between a generic and a non-generic method, and no explicit type instantiation has been given (i.e. one has been inferred), then the tie is resolved in favor of the non-generic method. See §18.10.5.7 for the rationale.

Particular substitutions may result in overload resolution errors where others may not.

```
    C<Foo,Foo> c2 = new C<Foo,Foo>();
    // Note that C<Foo,Foo> has methods:
    //    public object m(Foo x,   Foo y, object z) { ... }   // method
    //    public object m(Foo x,   Foo y, string z) { ... }   // method
    //    public object m(object x,Foo y, Foo z) { ... }   // method C

    public void Main()
    {
       c2.m(new Foo(), new Foo(), new Foo());   // Error, ambiguous
    }
```

This situation can be avoided easily by restrained and sensible use of overloading, just as in normal class design.

Comment [DRS46]: *Testing incompleteness: Overloads spread across multiple classes up the inheritance chain in combination with generics has not been well tested. In particular, the member lookup rules specify "all methods with the same signature as `M` declared in a base type of `S` are removed from the set" – I've not particularly tested if I'm determining "same-signature" here after applying the relevant instantiations as I should be.*

Comment [DRS47]: *Implementation incompleteness: This tie-breaking is not implemented, because method inference is not implemented..*

Comment [crusso48]: Onew reviewer said: On e way to resolve overloading for generic methods might be to give a full instantiation as a substitution with explicit domain:
o.m<W=U>(A[W] a,…)
o.<V=T>m<W=U>)(A[T,U] a,…)
Ugly, but at least if would be there if you need it.

18.10.6.1 Preferential selection of non-deprecated members

For C# designers: The problem highlighted below needs to be resolved somehow. What's mentioned below is what I've implemented (or had implemented – I've actually forgotten if I chopped it out of the codebase). I actually don't care which of the mechanisms are chosen, but some kind of general "Preferred" or "Prefer-non-deprecated" seems to make sense to me, in a wider context than generics. But I realize "Deprecated" doesn't carry quite the same intent as "Not-Preferred", so maybe a different attribute is required.

We could also take the following routes to fixing this kind of problem:

(a) Make the indexer in the derived class `private`. However in C# such overriding methods must have `public` accessibility, and in any case the presence of the two indexers would violate the duplicate-signature rule.

(b) If `Vector` were an interface, then we could use explicit method implementations. However, these are not permitted for base classes.

As a new rule, the C# compiler will select non-deprecated methods in preference to deprecated methods. The mechanism is a general one, allowing new versions of classes to provide more general functionality than previous ones and yet still maintain binary compatibility and versioning properties with respect to the interface expected by old clients of the class. This can be done even when there would be overloading errors or signature duplication errors stemming from the existence of the new methods.

To see why this rule is useful, we must look at a common pattern for generic collection classes. It is common to derive them from existing non-generic collection classes, where the functionality of the existing classes is exposed via virtual methods. Perhaps the simplest example of this pattern is as follows:

```
class Vector
{
   private object[] data;
   public virtual object this[int n]
   {
      get { return data[n]; }
      set { data[n] = value; }
   }
   public Vector(object[] init) { data = init; }
}


class GVector<V> : Vector
{
   private V[] data2;


   [Deprecated]
   public override object this[int n]
   {
      get { return (object) data2[n]; }
      set { data2[n] = (object) value; }
   }
   public virtual V this[int n]
   {
      get { return data2[n]; }
      set { data2[n] = value; }
   }
   public GVector(V[] init) : Vector(null)
     { data2 = init; }
```

```
    }
```

This pattern allows objects belonging to the constructed types associated with the generic collection class to be used where the non-generic classes are expected, and thus these objects can be passed to non-generic code.

However, note that in order to implement the `Vector` semantics of its base class, the `GVector` class must provide an indexer that returns values of type `object`. In addition, it is natural that in `GVector` we wish to provide another indexer that returns the more precise type `V`. However, this leads to a problem: at places where the indexer is used it will be impossible to distinguish which should be preferred.

The solution we use is to mark the new version of the old accessor as `Deprecated`. This attribute means both that a warning will be emitted if the method is ever selected and that the compiler should always prefer the use of a non-deprecated method over a deprecated one.

18.10.7 Element access expressions

The basic rules for *element-access* expressions are described in §MainSpec and these may be either array accesses or indexer accesses.

An array access expression may access an array whose element type is either a *type-parameter* or a *constructed-type*. The rules carry over unchanged, for example, if the element type is a *type-parameter* `V`, and the array type is `V[]`, then the overall type of the element access expression will be `V`.

An indexer access expression may access an object whose type is either a *type-parameter* or a *constructed-type*. When accessing an object whose type is a *type-parameter*, the *type-parameter* must be constrained by one or more types that have indexers, according to the rules of §18.9 and the rules for determining the set of members for an expression of variable type. The set of indexers for an arbitrary type is the set of members determined by the rules of inheritance modified to take into account inheritance through structured types (§18.12.5).

Comment [DRS49]: Testing incompleteness: I haven't tested indexers in interfaces as far as I recall.

18.10.8 This access expressions

As before a *this-access* is permitted only in the *block* of a constructor, an instance method, or an instance accessor, and itt has different meanings in these situations according to the rules of §MainSpec. These rules carry over directly when the *this*-access occurs within a *generic-class-declaration* or a *generic-struct-declaration*. In all cases, a *this*-access expression within the instance members of a *generic-class-declaration* of a type `T` with *class-type-parameters* `V1,...,Vn` is considered to have the type `T<V1,...,Vn>`, i.e. the most general type possible in the given generic class.

18.10.9 Base access expressions

A "base" access expression is used to access functionality of the base class within a subclass. The rules for base access expressions are essentially unmodified from §MainSpec. Note that the base class may be a *constructed-class-type*. Thus, at compile-time, *base-access* expressions of the form `base.I` and `base[E]` are evaluated exactly as if they were written `((B<C,D>)this).I` and `((B<C,D>)this)[E]`, where `B<C,D>` is the *constructed-class-type* that forms the base class of the class or struct in which the construct occurs.

18.10.10 Post-increment and post-decrement expressions

The expressions are essentially unchanged, except that they use slightly modified rules for operator overloading. The set of operators can include operators defined in *generic-class-declarations*. The process of choosing the appropriate operator and the instantiation of the class is the algorithm described in §MainSpec modified to include the operators available because they are part of the inherited set of members of a type according to the rules of §18.12.4.

18.10.11 Object creation expressions

The type `T` in a "new" expression `new T(args)` may be a *constructed-class-type* or a *constructed-struct-type*.

The method group of overloaded constructors will be made up of pairs of methods and *governing types* (§18.12.4). Once the method group of constructors has been determined, processing proceeds as follow:

- If the type `T` is simply a generic class or struct name `C` and no type arguments are given, then constructor type parameter inference (§18.10.5.8) is applied to the method group, potentially eliminating some of the entries in the method group and for those that remain returning an instantiation *inst*. For each method in the method group overload resolution then proceeds as if the expression were `new C<`*inst*`>(args)`.
- Overloading is resolved according to the rules in above (§18.10.6), applying the instantiations to the relevant signatures and resolving overloading as normal.

Comment [DRS50]: Constructor type parameter inference is not implemented.

The type `T` may not be a *type-parameter*, and thus it is not permitted to simply write `new V()`. However, the expression `new V[]` is permitted, as is accessing the constructors of `V` by using a "typeof" expression and reflection. An alternative design pattern is to require that clients pass in a factory object capable of producing values of type `V`.

For C# designers: This is the final major issue in constraints that requires resolution. Either we

(a) take exactly what I've spec'd and make people pass in extra factory objects and/or delegates; or

(b) we hack the CLR so that every class with a default constructor automatically supports an interface AND allow interfaces to contain constructors or

(c) we emit a call to reflection at this point (YUCK) or

(d) we hack the CLR to go look for a method when it sees this pattern in generic IL… Take your pick – I really can't decide on this one or

(e) we just support it in C# (not the CLR) and translate it out by synthesizing and passing in additional delegate parameters to the places where they are needed. This would also require some kind of explicit constraint indicating that a constructor is required, and leads us back down the merry path of structured constraints, though in this case C# would have more flexibility as it would just be being implemented in the language.

I guess I would go for (a) or (e) …. That seems to be the consensus of everyone I've spoken to as well.

18.10.12 Array creation expressions

An array creation expression includes a *type* for the array which may be a constructed type. Special rules for potentially inferring the type parameters for calls to generic constructors and generic methods immediately contained in the elements of the object creation expression apply (§18.10.5.9).

The type of the array may be a *type-parameter*, and thus it is permitted to simply write `new V[`*size*`]`. The elements of the array are assigned the "default" value for the type `V`.

18.10.13 Delegate creation expressions

An delegate creation expression includes a *type* for the delegate which may be a constructed type. The type arguments for the construction expression may also be inferred (§18.10.5.8).

The compile-time processing of a *delegate-creation-expression* of the form `new T(E)` where `T` is a *constructed-delegate-type* or a *delegate-type* that accepts *delegate-invoke-type-parameters* is augmented by the following steps:

If `E` is a method group:

- If `T` is a generic delegate type `D` accepting *delegate-create-type-parameters* and no type arguments are given, then constructor type parameter inference is applied (§18.10.5.8), potentially eliminating some of the entries in the method group, and for each that remains inferring an instantiation *inst*. Processing now proceeds as if the delegate type `D<`*inst*`>` had been specified for `T`.
- We can now assume that `T` has the form `D<`*inst*`>` for some delegate type `D` and instantiation *inst*. Note that *inst* must satisfy the constraints on the type parameters for `D` (§18.9.2).
- The method group for `E` must include exactly one entry whose signature is compatible with `D<`*inst*`>`. The notion of compatibility from §15.1 is extended in two ways: if `D` does not accept *delegate-invoke-type-parameters* then the rules of §15.1 are simply applied after applying the instantiation *inst* to the signature

Comment [DRS51]: Constructor type parameter inference is not implemented.

of `D`. If `D` accepts *delegate-invoke-type-parameters* then a method is compatible with `D`<*inst*> if all the following are true. The rules are checked after the instantiation *inst* has been applied to the signature of `D`, removing all *delegate-create-type-parameters*, creating a new signature *sig*.

- The method is generic and it accepts the same number of *delegate-invoke-type-parameters* as *sig*;
- The corresponding type parameters have identical constraints, up to renaming of type parameters.
- The method has the same number or parameters as *sig*, with the same types, in the same order, with the same parameter modifiers, up to renaming of type parameters.
- Their return type of the method is the same as the return type of *sig*, up to renaming of type parameters.
- The result is a value of type `D`<*inst*>, namely a newly created delegate that refers to the selected method and target object.

Comment [DRS52]: Testing incompleteness: This should be implemented, but I'm not sure I've tested every feasible bad combination of delegate types and instantiations.

Comment [DRS53]: Delegate invoke type parameters are not implemented at all.

Otherwise, if `E` is a value of a *delegate-type*:

- If `T` is a generic delegate type `D` accepting *delegate-create-type-parameters* and no type arguments are given, then constructor type parameter inference is applied (§18.10.5.8), potentially eliminating some of the entries in the method group, and for each that remains inferring an instantiation *inst*. Processing now proceeds as if the delegate type `D`<*inst*> had been specified for `T`.
- We can now assume that `T` has the form `D`<*inst*> for some delegate type `D` and instantiation *inst*. Note that the type arguments must satisfy the constraints on the type parameters for `D` (§18.9.2).
- `D`<*inst*> and `E` must be compatible (§MainSpec); otherwise, a compile-time error occurs. The notion of compatibility is extended in a similar manner to that immediately above to cope with *delegate-create-type-parameters* and *delegate-invoke-type-parameters*.
- The result is a value of type `D`<*inst*>, namely a newly created delegate that refers to the same invocation list as `E`.

Comment [DRS54]: Constructor type parameter inference is not implemented.

Note that no method type arguments may *ever* be specified in a delegate creation expression even if the argument passed is a generic method. Similarly, type parameters may not be specified even if the argument is a value of *delegate-type* and it expects *delegate-invoke-type-parameters*. That is, you can't use delegate creation to "partially apply" a generic method or generic delegate to a set of type parameters.

Comment [DRS55]: However, if a generic method occurs as an instance method within a generic class then by creating an instance of a generic class you can create a delegate to the instance method such that the generic classes type parameters are effectively fixed. Wild stuff...

Consequently this means that method type parameter inference is never applied to the argument passed to a delegate creation expressions, though constructor type parameter inference (§18.10.5.8) is applied.

18.10.14 "typeof" expressions

A "typeof" expression takes a type as its argument. The type may be a *constructed-class-type* or a *constructed-struct-type* or a *constructed-delegate-type*. The actual type object will be different for different constructed types. The type may also be a *type-parameter*. The type object that results will depend on the instantiation under which the generic code is being used. The type may be an uninstantiated generic type.

Comment [PS56]: Implementation incompleteness: typeof(List) where List is a generic type is not currently implemented.

TBD. Examples.

18.10.15 "sizeof" expressions

The type may specify a *constructed-struct-type*. Different instantiations of a *generic-struct-declaration* may have different sizes. As usual, `sizeof` can only be used in unsafe code.

18.10.16 Local variable declarations

A type involved in a local variable declaration may be a *constructed-type*. Local variable declarations and field initializers can also affect the type parameter inference process (§18.10.5.9), as the type written on the left of the declaration can be used in the inference process for the expression on the right.

18.10.17 foreach-statements

TBD. Currently supports only non-generic IEnumerable but DOES support presence of methods that give back constructed types.

Comment [PS57]: CLR and/or C# optimization: There is an implicit conversion (or checkcast) involved in every iteration of a C# foreach (T x in C) statement, which can be avoided when the type of the Current property (of the object E returned by C.GetEnumerator) is a subtype of T, as will usually be the case when the type of C is a generic collection. Hopefully the JIT compiler does not generate that conversion (or checkcast)?

TBD. Examples

18.10.18 specific-catch-clauses

TBD. Basically you can catch a constructed exception type.

TBD. Examples

18.11 Generics and attributes

TBD. Basically attribute classes may be generic. Attributes themselves may be contructed types.

Comment [DRS58]: This is not tested, and may not even be implemented.

18.12 Supporting definitions

Informally, the *set of base types* of a reference type is the set of all those types from which it is derived, i.e. all those types that occur in the inheritance chain of the reference type, taking into account any constructed types that occur "along the way". For example if a class `C<V>` inherits from a class `D<List<V>>`, then `D<List<int>>` is one of the base types of `C<int>`.

This section makes this and other notions precise by fully defining the following notions:

- the ***base class types of a class***;
- the ***base interface types of a class***.

We then lift these definitions to apply not just to class definitions, but also all class types, including constructed class types:

- the ***base class types of a class type***;
- the ***base interface types of a class type***;

Similarly for interfaces we define the following:

- the ***base interface types of an interface***;
- the ***base interface types of an interface type***.

TBD. Similarly for structs and delegates.

We then combine these to define

- the ***base types of a reference type***.

Finally we also define:

- the ***set of all function members of a class***;
- the ***set of all function members of an interface***;
- and the ***set of all function members of a struct***.

18.12.1 Base types of classes

18.12.1.1 Base class types of a class

We now define the ***base class types of a class*** `C`, i.e. the set of types from which a class `C` is derived. If `C` is a generic type then these types will involve the type parameters for the class `C`, and these type parameters only.

1. Start with *S* empty
2. Set *Current* to be `C` and set *inst* to be the *formal type instantiation* (§18.2.4) for `C`.
3. Insert *Current<inst>* into the set *S*, except on the first iteration.
4. If *Current* has no base type then terminate. Otherwise if *Current* has base type `C2`*<inst2>*:
    a. Set *inst* to be the composition of *inst* and *inst2*, set *Current* to `C2`, and go to step 3.

This recurses over the entire interface hierarchy of a class, computing the transitive closure of base types, some of which will be constructed. The iteration takes into account instantiations as we progress through constructed base classes.

For example, given the following inheritance hierarchy:

```
class C<U> { }
class D<V> : C<V> { ... }
class E<W> : D<List<W>> { ... }
```

then the set of base class types for class `E<W>` is { `C<List<W>>`, `D<List<W>>` }. The set of base class types for class `D<V>` is { `C<V>` }. The set of base class types for class `C` is empty.

18.12.1.2 Base interface types of classes

The set of ***base interface types of a class*** `C` is defined as follows:

1. Start with S empty.
2. Set *Current* to be the class `C` and *inst* to be the *formal type instantiation* (§18.2.4) for `C`.
3. For each supported interface *Int* of *Current*, apply *inst* to all the types in the *set of base interfaces types* (§18.12.2) for *Int* and add these items to *S*.
4. If *Current* has no base class or is `object`, then finish.
    a. Otherwise if the base class of *Current* is `C2`*<inst2>*, then set *inst* to be the composition of *inst* and *inst2*, set *Current* to `C2`, and go to step 3.

In the following example, the class `C` supports the interfaces `IList<Widget>` and `IWidgets`.

```
interface IList<V>  { V[] Elements(); }
interface IWidgets : IList<Widget>  {  void RedrawAll(); }
class C : IWidgets
{
   public Widget[] Elements() {...}
   public void RedrawAll() { ... }
}
```

The set of interfaces supported by a *generic-class-declaration* may involve the *class-type-parameters* of the *generic-class-declaration*. In the following example, the generic class `List<V>` supports both `IList<V>` and `IDictionary<int,V>`.

```
interface IList<V> { V[] Elements(); }
interface IDictionary<K,D> { D Find(K key); }
class List<V>: IList<V>, IDictionary<int,V> {   ... }
```

Consider the following classes.

```
interface I1<U> { U[] foo(U); }
interface I2<V> : I1<V[]>   { V[] bar(V); }
class B<W> : I2<W> { ... }
class C<X> : B<X>, I2<X> {  ... }
```

The interface set of `C<X>` is then { `I2<X>` , `I1<X[]>` }

18.12.2 Base types of interfaces

Just as the definition of the set of types from which a *class-declaration* or *generic-class-declaration* is derived requires special attention when the inheritance chain includes *constructed-class-types* (§18.2.2) so the set of interfaces supported by a *generic-interface-declaration* requires special attention when the inheritance chain for interface types includes *constructed-interface-types*. The purpose of this section is to make this intuition precise. We follow this by a specification of the set of interfaces supported by a class, and then by a specification of the set of interface members that a class must implement.

We now define the set *S* of ***base interface types for an interface*** I:

1. Start with *S* empty and *ICurrent* to be *I*.
2. Set *inst* to be the *formal type instantiation* (§18.2.4) for *I*.
3. Apply *ICurrent* to *inst* and insert it into the set *S* if not already present.
4. For each base interface *I2<inst2>* of *ICurrent*:
    a. Set *inst* to be the composition of *inst* and *inst2*, set *ICurrent* to *I2*, and recursively apply steps 2-4.

This recurses over the entire interface hierarchy of a class, computing the transitive closure of implemented interfaces, taking into account the instantiations as we progress through constructed base interfaces.

For example, consider the following interfaces:

```
interface I1<U> { ... }
interface I2<V> : I1<V[]> { ... }
interface I3<W> : I2<W> { ... }
```

The base interface types of I3 are as follows:

```
{  I1<W[]>; I2<W> }
```

The ***immediate base interface types*** of an interface are computed by the same process as above limited to one iteration.

18.12.3 Base types

We have defined the ***base class types of a class*** (§18.12.1.1). The ***base class types of a type*** is now defined as:

- If the type is a constructed class type `C<`*inst*`>` then:
    1. Find the set of base class types of the class `C`
    2. Apply *inst* to each type in this set;
- If the class type is a non-constructed class type `C`, just find the set of base types of the class `C`.

We similarly define the ***base interface types of a class type*** and the ***base interface types of an interface type*** by taking the base interface types of the relevant class or interface respectively and, if the given type is a constructed type, applying the relevant instantiation to these sets.

Take together, these yield the ***base types of a reference type***.

For example, given.

```
interface I1<U> { U[] foo(U); }
interface I2<V> : I1<V[]>   { V[] bar(V); }
class B<W> : I2<W> { ... }
class C<X> : B<X>, I2<X> {  ... }
```

The base types of the class type `C<string>` are { `I2<string>` , `B<string>` , `I1<string[]>` }, the base interface types are { `I2<string>` , `I1<string[]>` } and the base class types are { `B<string>` }.

18.12.4 All function members of a class

We now define the ***set of all function members*** of a class. This notion is, for example, is used in the definition of member lookup (§18.10.2). However, unlike §MainSpec we now return not just a set of members, but each member is paired with a type, called the ***governing type*** for the member. The types reflect the presence of constructed class types and constructed interface types in inheritance hierarchies.

In particular, the ***set of all function members*** of a class `C` is computed as follows.

1. Start with *S* empty
2. Set *Current* to be the class `C`. Set *inst* to be the *formal type instantiation* (§18.2.4) for `C`.
3. For each immediate member *m* of *Current* add the pair (*Current<inst>*, *m*) to the set *S*. Here *Current<inst>* is the governing type for *m*. If *m* has the `static` attribute then just add (*Current*, *m*). Note that the instantiation is not applied to the signature of *m* in either case, but is recorded in the governing type.

   Note: do not add a member *m* if another member *m'* already exists in *S* with the same signature as *m* (after the application of *inst* to *m*) and m' has the `new` attribute Replace a member *m'* by a member *m* with the same signature as *m'* (after the application of *inst* to *m*) if *m'* has the `override` attribute and *m* has the `virtual` or `abstract` attribute – if *m* does not have the `virtual` or `abstract` attributes then this is an error. If a member *m'* already exists with the same argument types as *m* (after the application of *inst* to *m*) and does not have the `new` attribute or the `override` attribute then this is an error.
4. If *Current* has no superclass specified, then finish with step 5

   Note: if *S* still has any members with the `override` attribute, then this is an error, as they do not correspond to any `virtual` or `abstract` methods.

   a. If *Current* has a base class `C2`<*inst2*>, then set *inst* to be the composition of instantiations *inst* and *inst2*, set *Current* to `C2`, and go to step 3.
5. Add the members for the additional base types `object`, `System.Delegate`, and `System.Array` as specified in §MainSpec, pairing the members of each type with the type itself.

Note that in the absence of generics the types are always just simple (i.e. non-constructed) reference types reflecting the position of the members in the inheritance hierarchy.

For example, consider the following classes.

```
class B<U> { }

class C<V>
{
   V[] foo(V x) { ... }
   virtual V bar(V x) { ... }
   static void baz(int x) { ... }
}
class D<W> : C<B<W>>
{
   new W[] foo(W) { ... }
   override B<W> bar(B<W>) { ... }
}
```

Excluding the members arising from the type `object`, the set of formal members of `D` is as follows:

```
{
   (D<W>,     new     W[]  foo(W))
   (C<B<W>>, virtual V bar(V))
   (C<B<W>>, static  void baz(int))
}
```

That is, the set of formal members of a class `D` tells us the what the members of the base classes of `D` “look like” when viewed from `D`, taking into account the inheritance hierarchy along the way. Note that the governing types for the members may involve the type parameters of `D` but no others.

18.12.5 All function members of an interface

TBD. Similarly for structs and delegates.

19. When and how to use generics

TODO: More to do here – This probably won't go in this spec in the end, but MSDN or some other material….

19.1 When should I use generics?

Before deciding to adopt generics for your project, you should carefully consider a range of issues.

The simplest scenario for using generics is if you will only be using them internally within a library or an application. For example, generic collection classes can be used internally in an application or algorithm without exposing generic objects or types to other components.

19.1.1 What if I interact with non-generic libraries?

All C# applications typically call libraries for operations such as I/O, performing networking calls, transacting data with databases and so on. These calls can all naturally be made regardless of whether you are using generic classes for some of your data structures. However, occasionally a little care is required:

- Some libraries will require inputs arranged as collections, e.g. a "list of widgets" or a "dictionary mapping handles to strings". The types expected by the library for these data structures will typically not be generic, constructed types, and instead will either be "hand-specialized" collection types such as `WidgetList` or they will be interfaces such as `IList` that transact values via the type `object`. In the former case you should be careful not to use generics and instead to create input of the appropriate kind.

19.1.2 What if I am writing new libraries?

If your data structures will be passed between components that you are developing, then you must also consider if these components will all be written in C#, and/or some other .NET language that supports the standard model of generics. If so, generic types can be used across your component boundaries.

A more complicated scenario is where you are writing a component that is designed to be used by all languages on the .NET platform, including those that do not understand generics. If so, you should not expose constructed types, generic classes or generic methods as part of the interface to your component. Thus the types that should appear in your component contracts should not contain constructed types, and generic code should not be exported from your component.

You may certainly, however, use generics for internal data structures within your component. However, you should be aware that you may inadvertently be exposing generic objects externally if they may be accessed via non-generic interfaces or base classes – this may be intended, but just as with any use of inheritance where subclasses are not public it is possible that non-generic code may not be able to access all the functionality or reflect on the type of the objects returned.

19.1.3 What about languages that don't support generics?

The model of generics provided by C# is the standard model of generics supported on the .NET Common Language Runtime. However, not all languages on the .NET platform support generics, and not all implementations that support some related feature (e.g. templates, generics or "parametric polymorphism") will necessarily utilize the standard implementation. Programmers using languages that do not understand generics directly can always access the functionality of generic code via the .NET Framework reflection libraries.

19.2 When and where should I use the different feature of generics?

The great majority of users of generic libraries need only learn how to instantiate and call generic collection classes, and how to call generic methods. As such, for most users there is no need to learn how to write new generic classes or code.

19.2.1 When should I use generic classes?

Generic code is often used to replace code that manipulates values of type `object`. The `object` based code will typically be less efficient as it will perform extra casts, boxing and unboxing operations. While such code sometimes provides greater flexibility (because objects of any type can be manipulated unchecked), it typically has inferior performance characteristics than the corresponding generic code, and the incorrect use of such classes may result in runtime errors that could be detected earlier when using generic classes.

Not all code that manipulate values of type `object` should be replaced by a generic code. Typically non-generic code transacting values of type `object` can be replaced by generic code if you want to enforce tighter uniformity across a range of values of type `object`. Such uniformity is often desirable, e.g. in the case of collection classes, but not always – for example heterogeneous collections of “widgets” accessed via interfaces or abstract base classes are frequently used in GUI framework libraries, and in this case there are few compelling reasons to enforce uniformity of type across sets of widgets, as the heterogeneous case is more common (i.e. frames containing many kinds of widgets, rather than uniform kinds of widgets) and few programmer errors are related to “putting the wrong widget inside the wrong collection of widgets”, i.e. the problem that generics would solve in this case.

Note that some data structures are deliberately designed in a way such that it is impossible to determine at compile time a uniform type for the elements of the data structure, for example consider a list that is designed to hold integers, strings and/or floats, or a method that generates values of these different types. In these cases it is sensible to either use the non-generic classes such as those provided in `System.Collections` of the .NET Framework, or to use generic collection classes instantiated at the type `object`. The latter approach explicitly signals the fact that a heterogeneous collection is desired.

The kind of uniformity imposed by generics is also very useful when the types of the inputs to a component are directly related to the types of the outputs. This can frequently be seen in the most useful applications of generic methods, interfaces and delegates.

19.2.2 When should I use generic methods?

For generic algorithms.

TBD

19.2.3 When should I use generic interfaces?

TBD

19.2.4 When should I use generic delegates?

When you want to use delegates to act as the generic operations over collection classes. Typically you will not need to define new generic delegate types yourself unless you are writing generic collection classes yourself.

19.2.5 When should I define new generic virtual methods?

Rarely. Only use these when you wish to dynamically switch between different generic algorithms, e.g. dynamically switch between sorting algorithms of the same name, defined in inheritance-related classes. However, some interesting applications are possible, for example consider a cascading hierarchy of n-ary vector types with specialed sorting algorithms where dynamic dispatch picks the appropriate sort method for each vector size.

Comment [DRS59]: A reviewer suggested this example – I haven’t tried it yet.

19.2.6 When should I define new generic delegate types that accept type parameters at invocation?

Rarely. Only use these when you wish to dynamically switch between different generic algorithms, e.g. dynamically switch between sorting algorithms of different names, or when the algorithms are not related by inheritance-related virtual methods.

19.3 How do I interface between generic and non-generic code?

The following code illustrates the use of conversions to convert a generic list to a non-generic list. This demonstrates one common technique to interface between generic and `object` based code.

```
public IList AdaptList<V>(IListG<V> inp) {
   ArrayList res = new ArrayList();
   foreach (V x in inp) {
      res.Add(x); // Here “x” is implicitly cast to the type “object”
   }
   return res;
}
```

19.4 What generics are not

Generics should not be confused with attempts to cover a very wide scope of possible applications of “generative programming.” The .NET Framework offers rich dynamic code generation facilities through its reflection libraries, and for advanced generative programming these libraries can be used in conjunction with generics.