Concept-C#

(Type)Classes for the Masses

Claudio Russo Matt Windsor

Wednesday 25 October 2017

Concepts are a proposed way to structure polymorphic C# code. Like interfaces, they constrain generic type arguments based on the presence of certain functions over those types. Unlike interfaces, but like extension methods, concepts separate the implementation of the functions from the types themselves, and do not tie the functions to existing objects. This lets programmers connect existing types to concepts, and abstract over static methods like constructors and type-level properties.

Introduction

NB: This document is still being written.

When writing generic code, we often rely on a handful of facts about the types used with that code. Most modern languages include *bounded parametric polymorphism*, or the ability to do this ahead of time in a type-safe manner[1]. There are two main approaches:

- *subtype polymorphism*: a bound on type `T` says that `T` is a subtype of some other type `U`, giving code access to all the methods of, or able to accept a, `U`[2];
- *typeclasses*: a bound on type `T` says that `T` is a member of some group ('class') `C`, giving code access to all methods that are in the definition of, or able to accept members of, `C`.

Both have the same idea—they assign functionality to types—but different strengths and weaknesses. Subtype polymorphism is a natural fit for object-oriented languages, but bakes whether or not a type satisfies certain bounds indelibly into that type's definition. Typeclasses are more flexible, and scale easily to bounds relating multiple types in complex ways, but are more difficult to weave into an object-oriented system.

.NET, Microsoft's managed code ecosystem, is object-oriented at its core. Since .NET 2.0, it has had a high-quality generic type system with subtype polymorphism based on classes—bundles of data and methods to operate on them—and *interfaces*—packs of method signatures that can be treated as subtypes for polymorphism. Interfaces deal with many of the same problems in bounded polymorphism as typeclasses, but inherit[3] many limitations from subtype polymorphism:

- They can only constrain the members of one type—their subtype.

[1] Except *Go*.
[2] If the polymorphism system supports variance, sometimes the bound is flipped: `T` is a _supertype of `U`.
[3] Pun intended.

- In .NET, they cannot abstract over constructors or other static members.
- It is hard to use the actual eventual type of an interface implementation in the interface method specification. This usually results in unnecessary *boxing* of said type into its interface form, which affects performance.

In fact, modern object-oriented programming languages are increasingly experimenting with a move towards typeclass-based bounded polymorphism. Languages like *Rust* have adopted typeclasses as the core of their generic type system. Other languages, like *Swift*, have hybrid systems, taking cues from both interfaces and typeclasses. At least one language, *Scala*, has both interfaces *and* typeclasses!

With this in mind, we propose *concepts*, a C# implementation of typeclass-style bounded parametric polymorphism. At first, this model seems alien to C#. However, as a recent language design proposal suggested, concepts find a home in the existing C# model as an extension of its existing *extension methods* system. Not only do concepts help us gain the expressiveness of powerful typeclass based bounded polymorphism, but they also let us tap into the specialisation power of modern .NET just-in-time compilers to give performance rivalling hand-specialised code. To assess the impact of concepts, we use *Concept-C#*, a fork of the *Roslyn* C# compiler,

This document

In this document, we:

- Look at the key parts of the C# type system, including extension methods;
- Give our prototype design for concepts by example, starting with the basics and proceeding to more exotic features available in the `Concept-C#` prototype;
- Discuss how we implement `Concept-C#` on top of the existing Common Language Runtime and its subtype-based generics system;
- Conclude by discussing related and future work.

The C# 7.2 type system

As one of the key languages of the .NET ecosystem, C#'s type system is heavily based on the underlying .NET Common Language Runtime (CLR). The CLR supports generic methods and types, with a rich system of constraints, variance, and subtyping.

Classes

Following in the Simula tradition, classes contain all data and code inside a C# program. While most classes must be instantiated as objects to be used, C# also supports static members, which can be used without an object reference.

Following in another tradition, we begin exploring C#'s object orientation by looking at how we might store shapes in C#. A `Rectangle` class might look as follows:

```
public class Rectangle
{
```

```
    // Data (fields)
    private int _length;
    private int _height;

    // Constructor
    Rectangle(int length, int height)
    {
        _length = length;
        _height = height;
    }

    // Instance method
    public int Area()
    {
        return _length * _height;
    }

    // Properties
    public int Length => _length;
    public int height => _height;

    // Static method
    public static Rectangle Square(int length)
    {
        return new Rectangle(length, length);
    }
}
```

Interfaces

If we have a method that works with the area of `Rectangles`, we may also need it to handle the area of `Triangles`, `Circles`, and other shapes. If each defines a method

```
public int Area();
```

then we can call upon C#'s interface-based subtyping system as follows:

```
public interface Shape
{
    int Area();
}

public class Rectangle : Shape
{
    // ...
    public int Area()
    {
        return _length * _Height;
```

```
    }
}

public class RightAngledTriangle : Shape
{
    // ...
    public int Area()
    {
        return (_opposite * _adjacent) / 2;
    }
}
```

The subtyping relationship between these classes is baked into the definitions of the types themselves. This means we run into problems if, say, we use a library that contains a nice implementation of `Circle`, but doesn't already implement `Shape`.

Generics

In the C# syntax, we mark generic methods and types with an angle-bracket-delimited *type parameter list*. To use generics, we *construct* the generic entity with a mapping of type arguments to type parameters. For example, the .NET Framework ships with a generic class, `Lazy<T>`, which lets a value of type `T` be lazily initialised when, and only when, its actual value is needed. `Lazy<int>` is a possible construction of `Lazy<T>` for lazy initialisation of integer values.

Each type parameter corresponds to an extra obligation when building a type deduction for its generic. For example, we deduce `Lazy<int>` as follows:

$$\dfrac{\dfrac{}{T \vdash \text{int}}}{T \vdash \text{Lazy}<\text{int}>}\ \text{Lazy}$$

Operator overloading and generic arithmetic

When we calculate the area of our `Shape`s, we use C#'s builtin operators `*` and `/` to multiply and divide integers. Perhaps, however, we need to express the dimensions of our `Shape`s in some other type. The builtin operators have us covered if we move to floating point numbers, or some other size of integer—but what of `Lazy<int>`, or any custom integer type we make or stumble across in the future?

To let programmers define their own numeric types, and use the familiar numeric notation, C# allows for *operator overloading*. Classes can define static methods with certain syntax, and C# will call into them when it encounters the corresponding operator.

This is useful as a way of making numeric code cleaner and easier to read, but can we abstract over the presence of such operators? Can we make a `Shape<TArea>`, where `TArea` supports `*` and `/`? Alas, no: operator overloads sit outside the subtyping system—there is no interface or class that represents having certain operators.

To abstract over generic arithmetic, we would need to abandon operator overloads and create a new interface. Even then:

- We can't make our arithmetic generic over existing classes, as they can't make themselves subtypes of the interface;
- We can't make it generic over primitive types, as they can't subtype at all;
- Even if we restrict ourselves to our own classes, we can't represent all of the operations we would need for practical generic arithmetic as interface methods.

Let's explore the third point. Most types that act like numbers work with a common set of operations: addition, subtraction, multiplication, splitting into absolute value and sign, and conversion from an integer[4]. We could try to make this set into a C# interface, but quickly run into trouble:

```
public interface INum<T>
{
    T Add(T other);
    T Sub(T other);
    T Mul(T other);
    T Abs();
    T Signum();
    // ...we can't implement conversion from integer:
    // we don't have an INum<T> to call it from!
}
// How do we make 'int' an INum<int>?
```

Extension methods

A key problem of system is that it is closed for extension. If we want to make a class implement an interface, we must modify the class's code. If we can't, we must resort to workarounds like static methods and type wrappers. This problem extends to adding new methods to existing types. For example, maybe we really need to be able to rotate a `Rectangle` 90 degrees, but aren't allowed to add a new `Rotate` method to the class itself.

To fix this, C# 3.0 introduced *extension methods*—static methods that can be invoked as if they were native methods on some target type:

```
public static class ShapeExtensions
{
    public static Rectangle Rotate(this Rectangle me)
    {
        return new Rectangle(length: me.Height, height: me.Length)
    }
}

var rec = new Rectangle(27, 53); // length 27, height 53
var rot = rec.Rotate();          // height 27, length 53
```

Extension methods are a powerful way to extend existing classes, but do have limitations. We can't constrain type parameters based on whether they have certain extension methods available, and certainly can't implement interfaces using them.

[4] We discuss this set in particular as it corresponds to Haskell's `Num` typeclass.

Concepts

In our tour of C# 7.1, we found some pain points with subtyping polymorphism:

- We can't add interfaces to existing types, including primitive types;
- We can't fit static methods, type-level properties, operator overloads, and other such exotica inside interfaces;
- Extension methods let us add methods to existing types, but we can't use them in bounds;

In this section, we explore our design for concepts in C#, showing how concepts can address all of these points.

Basic concepts

A concept names several methods and properties, and groups together types based on whether an implementation exists for each. Unlike interfaces, the methods and properties exist *at the type level*—they are not called on objects.

To underline the difference, here is a concept version of the `INum` interface we attempted earlier:

```
public concept CNum<T>
{
    T Add(T x, T y);
    T Sub(T x, T y);
    T Mul(T x, T y);
    T Abs(T x);
    T Signum(T x);
    T FromInteger(int x);
}
```

Concepts, like interfaces and subtyping relations, are constraint on types. However, we constrain types with concepts in a different way from usual. In Concept-C#, we make generic methods and classes dependent on concepts by adding `implicit` type parameters, which carry the concept requirement through a C# constraint. We can use `CNum` as follows:

```
public A SomePoly<A, implicit NumA>(A x, A c)
    where NumA : CNum<A> =>
    NumA.Add(NumA.Add(NumA.Mul(x, x), x), c);
```

Formally, when we introduce the `implicit NumA` type parameter, we create a proof obligation that `CNum<T>` holds before we get to use `SomePoly` for `T`. We can write this as a proof tree:

$$\frac{T \vdash \text{CNum}< \text{A} > \qquad T \vdash \text{A}}{T \vdash \text{SomePoly}< \text{A} >}$$

To use a concept, we give evidence that for some refinement of the concept type parameters (in this case, `T`), an implementation of the functions in the concept body exists. We do this with an `instance` declaration. For example, `int` has the following instance!

```
public instance Num_Int : CNum<int>
{
```

```
    int Add(int x, int y)  => x + y;
    int Sub(int x, int y)  => x - y;
    int Mul(int x, int y)  => x * y;
    int Abs(int x)         => Math.Abs(x);
    int Signum(int x)      => Math.Sign(x);
    int FromInteger(int x) => x;
}
```

Unlike interface implementations, we can make instances for types that already exist—including primitive types like `int`.

Formally, instances correspond to deduction steps in concept proof trees:

$$\cfrac{\cfrac{}{T \vdash \text{int}}}{T \vdash \text{CNum}<\text{int}>}\ \text{Num\_Int}$$

In most languages with concepts—including Concept-C#—the compiler solves these obligations through type inference. We discuss Concept-C#'s concept inference scheme later.

Derived instances

Suppose we want to perform `CNum` operations on values that are slow to compute and not guaranteed to be used. It makes sense to make `CNum` instances over `Lazy`—but making a new `Lazy<T>` for each `T` with which we want to do lazy calculations scales poorly.

In Concept-C#, we can give instances `implicit` type parameters. This means we can make instances conditional on other instances existing. We call such instances *derived*. A derived instance for any `Lazy<A>` for which we have an instance for `A` may look like this:

```
public instance Num_Lazy<A, implicit NA> : CNum<Lazy<A>>
    where NA : CNum<A>
{
    Lazy<A> Add(Lazy<A> x, Lazy<A> y) =>
        new Lazy<A>(() => Add(x.Value, y.Value));
    Lazy<A> Sub(Lazy<A> x, Lazy<A> y) =>
        new Lazy<A>(() => Sub(x.Value, y.Value));
    Lazy<A> Mul(Lazy<A> x, Lazy<A> y) =>
        new Lazy<A>(() => Mul(x.Value, y.Value));
    Lazy<A> Abs(Lazy<A> x)            =>
        new Lazy<A>(() => Abs(x.Value));
    Lazy<A> Signum(Lazy<A> x)         =>
        new Lazy<A>(() => Signum(x.Value));
    Lazy<A> FromInteger(int x)        =>
        new Lazy<A>(() => FromInteger(x));
}
```

Derived instances correspond to intermediate stages in a proof tree:

$$\cfrac{\cfrac{}{\text{CNum}<\text{int}>}\ \text{Num\_Int}}{\text{CNum}<\text{Lazy}<\text{int}>>}\ \text{Num\_Lazy}$$

Operator overloading

Writing numeric code in an functional style, like we did above, is tedious and quickly marches off the right margin of one's text editor. To solve this, we extend C#'s operator overloading support to concepts.

In Concept-C#, concepts can define their own operators. We can rewrite `CNum` to use operators:

```
public concept CNum<T>
{
    T operator +(T x, T y);
    T operator -(T x, T y);
    T operator *(T x, T y);
    T Abs(T x);
    T Signum(T x);
    T FromInteger(int x);
}
```

Concept operators are picked up when no builtin operator or operator overload is available, and a suitable concept instance is in scope. When we use `+`, `-`, or `*` on two values of type `T`, there is not already a valid operator or operator overload, and some `CNum<T>` is available, Concept-C# will pick up its definitions.

With concept operator overloading, we can rewrite `SomePoly` to this:

```
public T SomePoly<T, implicit NT>(T x, T c) where NT : CNum<T> =>
    x * x + x + c; // NT is in scope, so use its operators
```

We can also, finally, make our shapes library generic over numeric types[5]:

```
public class Rectangle<T, implicit NT> where NT : CNum<T>
{
    private T _length;
    private T _height;

    // Constructor
    Rectangle(T length, T height)
    {
        _length = length;
        _height = height;
    }

    public T Area()
    {
        return _length * _height;
    }

    public T Length => _length;
```

[5] `NT` is part of the class type parameters, which guarantees that the user of the class can't modify how to calculate the area, but means that two `Rectangles` with the same `T` but different `NT` will have different types. This is a difficult ergonomic problem with our model.

```
    public T height => _height;

    // Static method
    public static Rectangle Square(T length)
    {
        return new Rectangle(length, length);
    }
}
```

Concept extension methods

Operator overloads let us write concept-based generic arithmetic in a highly idiomatic way. However, if we want to take the absolute value or sign of a `Num<T>`, we still have to write awkward, functional code:

```
var abs = NumT.Abs(x);
var sig = NumT.Signum(x);
```

Ideally, we would like to call `Abs` and `Signum` on `x`, as if they were actual methods of the `T` class.

Just like we extended operator overloads to concepts, we extend extension methods: by prefixing the first parameter of a concept method with `this`, the method becomes a *concept extension method*. Applying this change to `Abs` and `Signum` means we can refactor:

```
var abs = x.Abs();
var sig = x.Signum();
```

Autofilling

At this stage, our `Num_Int` instance looks as follows:

```
public instance Num_Int : CNum<int>
{
    int operator +(int x, int y)  => x + y;
    int operator -(int x, int y)  => x - y;
    int operator *(int x, int y)  => x * y;
    int Abs(this int x)           => Math.Abs(x);
    int Signum(this int x)        => Math.Sign(x);
    int FromInteger(int x)        => x;
}
```

The first three methods are tedious to write, and likely to introduce copy-paste errors when implemented:

```
    int operator -(int x, int y)  => x + y; // oops!
```

To help out, Concept-C# infers trivial parts of instances. If an instance method is an operator overload, and there already exists a valid statically defined operator overload or builtin operator for the same types, Concept-C# will fill in the obvious definition of that operator. The above is, thus, equivalent to[6]:

```
public instance Num_Int : CNum<int>
{
    int Abs(this int x)           => Math.Abs(x);
    int Signum(this int x)        => Math.Sign(x);
    int FromInteger(int x)        => x;
}
```

Similarly, suppose we implement `CNum` for a class with methods already called `Abs` and `Signum`. Concept-C# can automatically forward concept extension methods to instance methods on the same class, saving us from needing to write:[7]

```
int Abs(this Clazz x) => x.Abs();
```

Defaults

Another operation we often do with numeric types is negation. We left this out of `CNum` since it can be defined entirely in terms of `-` and `FromInteger`. We may want to add it for two reasons: first, most numeric types will let us implement negation in a more performant way, and second, for some types it may be easier to define `-` in terms of `+` and negation. However, this means that we must implement negation in all of our `CNum` instances, even if it is just `x => FromInteger(0) - x`.

Concept-C# allows concepts to provide bodies for methods they specify. These bodies are then used as the *default implementation* of the method in any instance that doesn't either directly implement the method, or implicitly forward it based on the rules above.

The syntax for this is straightforward:

```
public concept CNum<T>
{
    // ...
    T operator -(int x) => FromInteger(0) - x;
}
```

Multi-parameter concepts

C# supports generic collection enumeration through the `IEnumerable<T>` interface, where `T` is the type of elements in the collection. The interface looks like this:

```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
```

[6] Future versions of Concept-C# will probably prevent the programmer from overloading builtin operators anyway, to prevent surprises.
[7] We don't forward static methods, but maybe we should.

```
public interface IEnumerator<out T> : IDisposable, IEnumerator
{
    T Current;
    void Dispose();
    bool MoveNext();
    void Reset();
}
```

If we have a collection `xs` whose type implements this interface, we can use C#'s `foreach` syntax, which looks like this:

```
foreach (var x in xs)
{
    DoSomething(x);
}
```

This pattern also appears at the core of *LINQ to Objects*, where operators such as `select` and `where` transform enumerables into other enumerables. An example of LINQ is:

```
var ys = from x in xs where x > 5 select x * 2;
```

This setup works well—if the author of `xs`'s type had the foresight and patience to implement `IEnumerable<T>`, `IEnumerator<T>`, and the non-generic equivalents `IEnumerable` and `IEnumerator`. We lose any compile-time type information about the `IEnumerator` of `xs` *is*, which hits performance both by introducing virtual calls and preventing compile-time specialisation. Worse, `IEnumerable<T>` is a hassle to implement properly. Indeed, C# is flexible in the case of `foreach`, and accepts types that implement the correct methods, but not the full interface.

For a satisfying concept implementation of enumerables, we must make the concept generic not only on the type of the collection, but also the type of the enumerator. Similarly, the concept for enumerators must be generic both on the enumerator and the element. This needs a more general idea of concepts than previously seen. C# already has multi-parameter interfaces and structs, so our implementation of multi-parameter concepts is simple.

Using C#7.2's support for by-reference extension methods, we implement enumerators and enumerables in Concept-C# as follows:

```
public concept CEnumerator<TEnum, [AssociatedType]TElem>
{
    TElem Current(ref this TEnum e);
    void Dispose(ref this TEnum e);
    bool MoveNext(ref this TEnum e);
    void Reset(ref this TEnum e);
}
public concept CEnumerable<TColl, [AssociatedType]TEnum>
{
    TEnum GetEnumerator(TColl c);
}
```

When we ask for the enumerator of a `CEnumerable`, we know its concrete type *at compile-time*. We sacrifice some of the abstraction of the interface version, but get possibilities to optimise performance.

Lightening the load

Implementing `CEnumerable` directly is still overkill for many collections. For example, collections that contain a bounded, ordered number of elements with `O(1)` lookup given a position in the collection have trivial enumerators, which must be wired up separately for each instance.

Derived instances mean we can break down the implementation of a large concept by using smaller concepts. Suppose we have the following concept for looking up items in a collection by their index:

```
public concept CIndexable<TColl, TElem>
{
    TElem At(this TColl c, int i);
}
```

Suppose we also have the following concept for taking length of a collection:

```
public concept CCountable<TColl>
{
    int Count(this TColl c);
}
```

We can create a generalised `CEnumerable` instance capturing the standard pattern of implementing enumerables for these types of collection, which looks as follows:

```
public struct ICEnumerator<TColl>
{
    public TColl src;
    public int pos;
    public int len;
}
public instance Enumerable_IC<TColl, [AssociatedType]TElem, implicit I, implicit C>
    : CEnumerable<TColl, ICEnumerator<TColl, TElem>, TElem>
    where I : CIndexable<TColl> where C : CCountable<TColl>
{
    ICEnumerator<TColl, TElem> GetEnumerator(this TColl c) =>
        new ICEnumerator<TColl, TElem>
            { src = c, pos = 0, len = C.Count(c) };

    TElem Current(ref this ICEnumerator<TColl, TElem> e) => e.src[e.pos];

    void Dispose(ref this ICEnumerator<TColl, TElem> e) => {};

    bool MoveNext(ref this ICEnumerator<TColl, TElem> e)
    {
        if (e.len <= e.pos) return false;
        e.pos++;
        return (e.pos < e.len);
    };

    void Reset(ref this ICEnumerator<TColl, TElem> e) => e.pos = -1;
```

```
}
```

Now, type creators can write instances for the much simpler `CCountable` and `CIndexable` concepts, and the `CEnumerable` instance (and, through inheritance, the `CEnumerator` instance) write themselves. For example, we can define list enumeration as possible:

```
public instance Indexable_List<TElem> : CIndexable<List<TElem>>
{
    TElem At(this List<TElem> c, int i) => c[i];
}
public instance Countable_List<TElem> : CCountable<List<TElem>>
{
    int Count(this List<TElem> c) => c.Count;
}
```

Specialisation

We now have an instance for `CEnumerable<List<T>, ..., T>` without actually implementing `CEnumerable` itself. What if, for performance reasons, we instead want to create a bespoke `CEnumerable` instance for `List<T>`? In fact, we might want to use the same enumerator `List<T>` already exposes—`List<T>.Enumerator`—instead of making our own.

We can write a working instance over `List<T>.Enumerator` as follows:

```
public instance Enumerable_List<TElem>
    : CEnumerable<List<TElem>, List<TElem>.Enumerator, TElem>
{
    List<TElem>.Enumerator GetEnumerator(this List<TElem> c) =>
        c.GetEnumerator(c);

    TElem Current(ref this List<TElem>.Enumerator e) => e.Current;
    void Dispose(ref this List<TElem>.Enumerator e) => e.Dispose();
    bool MoveNext(ref this List<TElem>.Enumerator e) => e.MoveNext();
    void Reset(ref this List<TElem>.Enumerator e) => e.Reset();
}
```

This poses a problem: there are now two suitable instances for the concept `CEnumerable<List<TElem>>`. By default, Concept-C# refuses to infer an instance when this happens. This is to help programmers diagnose unintentional overlaps in their instances.

We can still use lists wherever we need `CEnumerable`, by manually specifying which instance we want to use. For example,

```
var general =
    CEnumerable<List<int>,
                Enumerable_IC<List<int>,
                              int,
                              Indexable_List<int>,
                              Countable_List<int>>
    .GetEnumerator(xs);
```

```
var specific =
    CEnumerable<List<int>,
                Enumerable_List<int>>.GetEnumerator(xs);
```

In most cases, we want Concept-C# to pick our more specialised instance over the general one, and don't want to write out the entire instance. Concept-C# has heuristics for *tie-breaking* when multiple instances are available. To use them, we must tell it that the less suitable instance can be *overlapped* by any more suitable instance, or the more suitable instance can be *overlapping* less suitable instances[8].

We can turn tie-breaking on by making the following change to our general instance:

```
[Overlappable]  // Enumerable_List can overlap this
public instance Enumerable_IC<TColl, TElem, implicit I, implicit C>
    : CEnumerable<TColl, ICEnumerator<TColl, TElem>, TElem>
    where I : CIndexable<TColl> where C : CCountable<TColl>
```

Now, concept inference will pick `Enumerable_List`, if it is in scope.

From interfaces to concepts, and back again

Existing C# code only uses interfaces, not concepts. As a result, it would be hard to use concepts for common patterns such as enumerables if both systems were incompatible.

Adapting interfaces to concepts is as easy as creating a catch-all instance for types implementing that interface, using C#'s existing support for interface constraints. We can port `IEnumerable` code to `CEnumerable` as follows:

```
[Overlappable]
public instance Enumerable_IEnumerable<TElem>
    : CEnumerable<IEnumerable<TElem>, IEnumerator<TElem>, IEnumerator<TElem>>
{}
[Overlappable]
public instance Enumerator_IEnumerator<TElem>
    : CEnumerator<IEnumerator<TElem>>
{
    bool MoveNext(ref this IEnumerator<TElem> e) => e.MoveNext();
    TElem Current(ref this IEnumerator<TElem> e) => e.Current;
    void Dispose(ref this IEnumerator<TElem> e) => e.Dispose();
}
```

This pattern generalises to other concepts. We can port `IComparable` code to `CComparable` as follows:

```
[Overlappable]
public instance Comparable_IComparable<TLhs, TRhs>
    : CComparable<TLhs, TRhs>
    where TLhs : IComparable<TRhs>
{
```

[8] This system is based on the Glasgow Haskell Compiler's *overlapping instances* extension.

```
    int Compare(TLhs l, TRhs r) => l.CompareTo(r);
}
```

TODO: this example is probably wrong, we need to check what refkinds do to CEM lookup.

Going the other way—creating an `IEnumerable` from `CEnumerable`—is harder. Unlike interfaces, concepts have no tangible type representation. Interfaces implicitly 'box' their actual implementation, so we must do a similar job with the concept witness. This works as follows:

```
class EnumeratorShim<TElem, TState, implicit N> : IEnumerator<TElem>
    where N : CEnumerator<TElem, TState>
{
    private TState _state;

    public EnumeratorShim(TState state)
    {
        _state = state;
    }

    public E Current => N.Current(ref _state);
    object IEnumerator.Current => N.Current(ref _state);
    public bool MoveNext() => N.MoveNext(ref _state);
    public void Reset() { N.Reset(ref _state); }
    void IDisposable.Dispose() { N.Dispose(ref _state); }
}
```

Associated types

Multi-parameter concepts can be hard to infer, as some of the type parameters may not correspond to parameters of the functions we want to call through them. However, it is often the case that one or more of those parameters can be determined precisely from the instance we choose to fulfil the concept.

In the `CEnumerable` example, `TElem` is one such type: it appears only as a return type, and C# cannot infer return types. `TEnum` can also be tricky to work out if we enter the concept through a call to `GetEnumerator`, for the same reason.

Languages that implement multi-parameter concepts have various solutions to this, such as Haskell's type families and functional dependencies, and Rust's associated types. Concept-C# has a simple system approximating associated types. Type parameters marked `[AssociatedType]` will be filled in by the concept witness inferrer as soon as it fixes an instance for a concept constraint that mentions that type.

We can redefine our `CEnumerator` and `CEnumerable` concepts as follows:

```
public concept CEnumerator<TElem, [AssociatedType] TEnum>
{ /* ... */ }
```

```
public concept CEnumerable<TColl,
                           [AssociatedType] TElem,
                           [AssociatedType] TEnum>
: CEnumerator<TElem, TEnum>
{ /* ... */ }
```

Now, whenever we try to infer an instance for `CEnumerator` or `CEnumerable`, we will pick up the `TElem` and `TEnum` of the inferred instance.

Standalone instances

Concepts can access a large amount of features normal extension methods don't have. However, we can also use instances as a powerful direct alternative to standard extension classes. If we make a *standalone instance*—one that inherits from no concepts—we can attach concept extension methods and operator overloads[9] to a type without creating a concept to hold them.

For example,

```
public instance GooPlus
{
    public Goo operator +(Goo x, Goo y) => /* ... */
}
```

will add an overload for `+` to all `Goo` as long as `GooPlus` is in scope.

Example: a self-specialising *LINQ to Objects*

TODO

Implementation

We require no changes to the Common Language Runtime (CLR). We need only modest changes to the compiler and an extension to the core library. Still, native CLR support for concepts would be useful: it would enable more flexibility in implementing, optimising, and using them.

Attributes

To let us access concepts and instances from compiled assemblies, while not changing the CLR, most of the new types of declaration in Concept-C# map to custom attributes. Concept-C# knows, when reading in metadata from compiled assemblies, to treat this attribute encoding in the same way as the new concept syntax.

[9] It might be a good idea to constrain operator overloads in standalone instances, eg. they can't be used unless there's a concept constraint, or by adding type parameter restrictions.

This means that other CLR languages can interoperate with our concepts, either by mapping it to new syntax or just exposing the attributes directly.

Concepts are interfaces

Our prototype compiles `concepts` directly to interfaces, with the following rules:

- We apply the `System.Concepts.ConceptAttribute` attribute to the interface;
- Each method gains a `public` modifier;
- Operators are translated to methods through their usual special naming convention—unlike actual C# operators, they remain instance methods;
- Extension methods gain the `System.Concepts.ConceptExtensionAttribute` attribute, and then translate as if they were *normal* methods.

Our `CNum` example from earlier becomes:

```
using System.Concepts;

[Concept]
public interface CNum<T>
{
    public T op_Addition(T x, T y);
    public T op_Subtraction(T x, T y);
    public T op_Multiply(T x, T y);
    [ConceptExtension] public T Abs(T x);
    [ConceptExtension] public T Signum(T x);
    public T FromInteger(T x);
}
```

Instances are structs

Similarly, we translate each `instance` to a `struct`, with the following rules:

- We apply the `System.Concepts.InstanceAttribute` attribute to the struct;
- Autofilled methods are expanded by the compiler into calls into whichever method or operator they are autofilled by;
- As above, each method becomes `public` and each concept extension method gains `ConceptExtension`.

Our autofilled instance of `CNum<int>` becomes:

```
using System.Concepts;

[Instance]
public struct Num_Int : CNum<int>
{
    [ConceptExtension] public int Abs(int x)     => Math.Abs(x);
    [ConceptExtension] public int Signum(int x) => Math.Sign(x);
    public int FromInteger(int x)                => x;
```

```
    [CompilerGenerated] public int op_Addition(int x, int y)    => x + y;
    [CompilerGenerated] public int op_Subtraction(int x, int y) => x - y;
    [CompilerGenerated] public int op_Multiply(int x, int y)    => x * y;
}
```

`implicit` type parameters

To make concept inference work properly across the separate compilation boundary, the `implicit` keyword becomes the `System.Concepts.ConceptWitnessAttribute` attribute in the translation. Also, any `implicit` type parameter gains a `struct` constraint, since concept instances are always value types.

The signature of `SomePoly` translates as follows:

```
using System.Concepts;

public T SomePoly<T, [ConceptWitness]NT>(T x, T c) where NT : struct, CNum<T>;
```

Concept instance calls

Any attempt to access a member on a concept instance of type `T` is translated in one of two ways:

- If the call is inside a block, we generate an uninitialised local variable for `T` at the outermost block, and turn the call into an instance call on that variable[10];
- Otherwise (for example, in field initialisers), we call into `default(T)`.

To demonstrate, let's translate `SomePoly`:

```
public T SomePoly<T, [ConceptWitness]NT>(T x, T c) where NT : struct, CNum<T>
{
    NT nt;
    return
        nt.op_Addition(
            nt.op_Addition(
                nt.op_Multiply(x, x),
                x
            ),
            c
        );
}
```

If this body were inside an initialiser, it would instead become:

```
    default(NT).op_Addition(
        default(NT).op_Addition(
            default(NT).op_Multiply(x, x),
```

[10]We re-use the same variable for subsequent calls into `T`, too.

```
            x
        ),
        c
    ) ;
```

Defaults are also structs

Whenever a concept has default implementations, we generate a nested struct inside the concept to hold them. This struct is generic on one type parameter: the instance that is calling into the default implementation. This makes sure the default can call back into the original instance.

If we added negation to `CNum` as suggested earlier, this would give us:

```
public interface CNum<T>
{
    // ...
    public T op_UnaryNegation(T x);

    [CompilerGenerated]
    public struct _default<W> where W : CNum<T>
    {
        public T op_UnaryNegation(T x)
        {
            W w;
            return w.op_Subtraction(w.FromInteger(0), x);
        }
    }
}
```

Inference

We extend C#'s type inference algorithm in two ways:

- Infer missing concept-witness and associated-type parameters with a new, Haskell-style inference strategy;
- Allow partially specified type arguments when the missing arguments are concept witnesses or associated types, and *part-infer* the rest;

TODO

Related Work

Other languages

The rise of typeclass-based polymorphism in other languages motivates us greatly. While languages such as Haskell have always had it, recently there has been an explosion of use in languages closer to C#:

- Rust has *traits*: concepts with defaults, unnamed witnesses (*trait bounds*) and instances (*impls*), and C#-style syntax. Rust uses traits for zero-cost abstraction, generating specialised code for trait-bound methods where possible. When it needs to, it can use boxed '*trait objects*', containing type-erased data and vtable pointers.
- Swift has *protocols*: C#-style interfaces, but with support for writing extensions to add existing classes to new interfaces. These behave very much like Concept-C# instances. Swift has optional requirements, which are similar to defaults (but put the onus of specifying the default behaviour on the caller).
- Scala has *implicit parameters*: concepts whose instances are singleton objects passed as implicit formal parameters (not type parameters). This is very close in design, expressivity, and feature set to our implementation. However, we pass witnesses as types rather than singleton objects, and use a special type declaration for concepts (which are just abstract classes in Scala). This gives stronger performance guarantees through JIT specialising.
- C++ has theoretical (documentation-level) concepts with no language support beyond the much more general template system. However, this does not give use-site typechecking or separate compilation. Language support for concepts is slowly progressing.

Shapes proposal

TODO

F#

As a quick experiment, a team at Microsoft Research Cambridge (ourselves, as well as Don Syme, James Clarke, and Rupert Horlick) built a prototype fork of Visual F# adding a subset of concepts to F#.

Conclusions

TODO

Future work

There are many unanswered questions that we must answer before making C# concepts a reality. These are not just syntactic—'is `concept` the best keyword?', 'should witnesses be explicit or implicit?'—, but also semantic—'what should the tie-breaking rules be?', 'when should concept extension methods appear in scope?'—and even relating to usability and discoverability—'are concepts too much of a cognitive burden on developers?'. We do not have answers to many of these questions, but look forward to the challenge of finding them.

Support for other languages

While we have tried to keep concepts as portable as possible—the only C# specifics are how inference and binding works, to our knowledge—extending this proposal to other languages in a user-friendly manner would require a nontrivial amount of porting.

The Roslyn compiler project targets both C# and Visual Basic. If we implement concepts for C#, it would be natural to also implement them for Visual Basic. This would need extra work on creating idiomatic syntax for Visual Basic concepts, and also extending the language compiler.