Concept C#

Prototype Implementation Report

Matt Windsor <t-mawind@microsoft.com>;Claudio Russo<crusso@microsoft.com>

# Contents

[0 Contents 1](#_Toc458771637)

[1 Introduction 3](#_Toc458771638)

[2 Features in Prototype Concept C# 3](#_Toc458771639)

[2.1 Overview 3](#_Toc458771640)

[2.1.1 Status 3](#_Toc458771641)

[2.1.2 Examples 4](#_Toc458771642)

[2.2 Concepts 5](#_Toc458771643)

[2.2.1 Overview 5](#_Toc458771644)

[2.2.2 Syntax 6](#_Toc458771645)

[2.2.3 Implementation 6](#_Toc458771646)

[2.2.4 Case study: generic arithmetic 8](#_Toc458771647)

[2.3 Associated Types 9](#_Toc458771648)

[2.3.1 Overview 9](#_Toc458771649)

[2.3.2 Syntax 9](#_Toc458771650)

[2.3.3 Implementation 9](#_Toc458771651)

[2.3.4 Case study: concept enumerators 9](#_Toc458771652)

[2.4 Default Implementations 12](#_Toc458771653)

[2.4.1 Syntax 12](#_Toc458771654)

[2.4.2 Implementation 12](#_Toc458771655)

[2.4.3 Case study: Prelude 12](#_Toc458771656)

[2.5 Operator Overloads 13](#_Toc458771657)

[2.5.1 Syntax 13](#_Toc458771658)

[3 Motivation and Comparison 14](#_Toc458771659)

[3.1 Existing Languages 14](#_Toc458771660)

[3.2 Static Interface Methods and *IArithmetic* 14](#_Toc458771661)

[4 Known Issues and Future Work 15](#_Toc458771662)

[4.1 Language Interactions 15](#_Toc458771663)

[4.1.1 Concept witnesses vs. constraints on non-concepts 15](#_Toc458771664)

[4.1.2 Defaults vs. member access on a direct instance 15](#_Toc458771665)

[4.1.3 Defaults vs. interfaces 15](#_Toc458771666)

[4.1.4 Type inference vs. variance 15](#_Toc458771667)

[4.1.5 Type inference vs. coercions 15](#_Toc458771668)

[4.1.6 Operator overloads vs. existing operator overloads 16](#_Toc458771669)

[4.1.7 Scoping rules vs. existing scoping rules 16](#_Toc458771670)

[4.2 Future Work 16](#_Toc458771671)

[5 Language Interoperability 17](#_Toc458771672)

[5.1 F# 17](#_Toc458771673)

[5.2 Visual Basic 18](#_Toc458771674)

[6 Bibliography 18](#_Toc458771675)

# Introduction

This document discusses the work by us (Matt Windsor and Claudio Russo), during the former’s internship at Microsoft Research Cambridge, to prototype an implementation of concepts (also known as traits, typeclasses, protocols, and by other names) for C#. It identifies:

* The set of features implemented, with brief notes on syntax, implementation, example case studies, and status;
* The motivation for implementing concepts in C#;
* Potential problematic interactions with C#’s existing features, and known issues in the prototype implementation;
* Brief notes on language interoperability;
* Future work we propose between now and a hypothetical feature request.

# Features in Prototype Concept C#

## Overview

Our progress can be divided roughly into four feature areas: concepts, associated types, operator overloads, and defaults.

### Status

Table 1 outlines the status of the design, syntax, and implementation of the features, as well as recommendations for a C# language proposal going forward.

Table 1. Summary of feature status at end of internship.

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | Design and Syntax | Implementation | Recommendation |
| Concepts | Mostly solid, could be improved | Complete, needs refactoring and fixing | **Adopt with clean rewrite** |
| Associated Types | Implementation work and call site only. Definition site needs heavy design work | Call site mostly complete, but type inference needs work. Definition site blocked by lack of design | **Consider adopting with new design** |
| Operator Overloads | Needs redesign: rough proof of concept only | Partially implemented for basic operators. Full implementation easy but not pursued due to design being poor. | **Perhaps omit unless a good design is forthcoming** |
| Defaults | Mostly solid, could be improved | Rough proof of concept only. Needs complete rework | **Perhaps omit: implementation hard, benefits small** |

### Examples

To demonstrate the above features, we have a sizeable collection of examples exercising all features. These examples include:

* Concepts
  + An implementation of a small cross-section of the Haskell Prelude, including generic arithmetic (including lazy arithmetic via Lazy<T>);
  + Generic arithmetic versions of both the Quickhull algorithm and Conal Elliot’s Beautiful Differentiation.
* Associated Types
  + CEnumerable, a concept-based implementation of both IEnumerable and IEnumerator as one concept;
  + Tuple2, a concept abstracting over pairs, and implemented for Tuple, ValueTuple, and KeyValuePair;
  + A concept-based implementation of the graph traversal library from Garcia et al.'s 'An Extended Comparative Study of Language Support for Generic Programming (<http://www.osl.iu.edu/publications/prints/2005/garcia05:_extended_comparing05.pdf>) and their earlier Haskell code (<http://www.osl.iu.edu/research/comparing/haskell_readme.html>).
* Operator Overloads
  + An operator-overloaded version of the Haskell Prelude;
  + Operator-overloaded versions of both Quickhull and Beautiful Differentiation.
* Defaults
  + Default-implemented methods on both versions of the Prelude.

## Concepts

### Overview

The concepts feature contains the core functionality of C# concepts. It introduces the following new constructs:

* A concept, which is an abstract type declaration that contains a series of function signatures over one or more type parameters that describe a pattern of functionality over which generic code can abstract;
* An instance, which is a concrete and standalone type declaration that provides implementations for a concept’s function signatures for a specific instantiation of that concept’s type parameters, thus serving as evidence that those types implement the concept;
* A concept witness, which is a type parameter in the signature of a method or class that is constrained to implement a concept, and thus accepts any instance that implements said concept for the specific construction of the constraining concept’s type parameters at call site.

(The F# implementation uses different terminology: there, concepts are called ‘traits’, instances ‘witnesses’, and concept witnesses have no specific term. Resolving this difference one way or the other would be beneficial for interop).

To allow concepts to be used with transparency as close to interfaces as possible, we also:

* Add a third pass to type inference, which attempts to fix all concept witnesses by filtering the set of all instances in binder scope at use site to those which implement the required concepts, can have their type parameters recursively inferred (including transitive concept witnesses), and are most specific;
* Allow the user to omit implicit type parameters in both method and class type argument lists, thus triggering the third inference pass separately for those missing parameters only (called part-inference);
* Add new tactics to name lookup to place witness methods implicitly in the local scope (subject to removal, see below); allow use of methods by direct reference to an instance or concept witness (no implementation necessary) or by the concept itself (with the best instance being inferred);
* Add a lowering step to translate calls onto instance types into calls to a default object of that type.

Our implementation of concepts is similar to the Scala implicit objects feature. It is general, allowing for multiple instances per type/concept pair; limited tie breaking in ambiguous situations; explicit naming of instances; undecidable instances; and multi-parameter concepts. This, however, means that the type system is missing some theoretical properties that more restricted systems, such as Haskell 98, deliver.

### Syntax

The concepts feature adds two new type declaration keywords: concept and instance. These are syntactically equivalent to interface and struct respectively. This has added two new type declaration syntax forms to the API, but not modified any existing forms.

The keyword implicit is now allowed to exist after the variance modifier (or lack thereof) in a type parameter declaration. This has modified the API for type parameter declarations. Adding implicit to a type parameter marks it as a concept witness: a type parameter may be constrained by concepts, and is inferred by the third pass, if and only if it is implicit.

Part-inference means that the type argument list of a method or class can contain fewer elements than the corresponding type parameter list, as long as the difference is exactly the number of implicit parameters.

### Implementation

#### Type declarations

The new type declaration forms desugar to existing type declaration forms. The concept form translates to interface; instance to struct. This allows them to be represented in the IL with no runtime changes.

Aside from default accessibility (instance members default to public), the changes to name resolution, type parameter constraints, and inference, the new forms are semantically identical. This is because concepts are literally interfaces over instances, which are just structs with special method invocation semantics.

#### Annotations

In the IL representation, the new type declarations are distinguished from their interface and struct parents by two attributes: [Concept] and [Instance] (in System.Concepts). These carry the same semantics as the presence of the respective new keywords above.

#### Inference and part-inference

The new inference system is self-contained as a new class, ConceptWitnessInferrer. To allow it to be used, we add stubs to MethodTypeInferrer to add it to normal method type inference, and Binder and OverloadResolution (PartInferImplicitTypeParameters) to allow for part-inference and invocation of methods on concepts.

The ConceptWitnessInferrer class can infer witnesses in any situation where a Binder exists (to allow it to discover in-scope instances), regardless of whether they are inside method type arguments, class type arguments (for recursive and part-inference), or standalone with known required concepts (for Form B below).

The high-level scheme of single-type-argument inference for the concepts feature (ignoring changes necessary for associated types) is:

1. Use the binder to find all instances in scope (members of a container in scope, or top-level members of a namespace imported using using);
2. Obtain the set of concepts the argument requires (either given directly to the inferrer, or taken from the set of constrained interfaces on the argument’s type parameter filtered to remove superclasses);
3. Apply all already-inferred type substitutions to the concept set;
4. Filter all instances to those that implement all required concepts, checking using TypeUnification, and apply the resulting unification back to the instances (to fill in some of their type parameters using the types we already know);
5. Filter all instances to those whose type parameters are all fixed. At this stage, if any type parameters are unfixed, and they are concept witness parameters, we recursively begin inferring them. This is how we implement derived instances;
6. Filter all remaining instances using some ad-hoc ‘tie breaker’ rules. This includes rejecting any instances that implement superclasses of the concepts implemented by other instances, and preferring instances with fewer type parameters;
7. If there is exactly one instance remaining, inference has succeeded; else, it has failed (either due to unsatisfiability or ambiguity).

#### Name lookup

We have implemented three different approaches to look-up and invocation of members in witnesses (in order from least to most implicit, and least to most controversial):

1. Allow accesses to Instance.Member by suppressing the usual ERR\_ObjectRequired from MemberGroupFinalValidationAccessibilityChecks when Instance is a concept witness or instance and lowering the resulting type-expression receiver to default(Instance) (see below);
2. Allow accesses to Concept<T>.Member by changing BindMemberAccessWithBoundLeft to perform concept witness inference if the bound left is a concept;
3. Allow accesses to Member directly, when it is a member of a concept witness in scope, by adding a new WithWitnessesBinder into the binder chain for each parent method or type. This binder brings the members of all witnesses that are type parameters of the method or type directly into scope before the members of that method or type.

Form B is equivalent to the lookup implemented in the F# prototype, and would be sufficient for most purposes. Form A is useful when directly calling into an instance when the inference used in the second has failed to disambiguate. Form C is terse and powerful, but complicates the scoping story for C# and cannot resolve to instances not named by witnesses. It should perhaps be changed to a more ‘extension methods’-style design and implementation, or preceded by some syntax for explicitly creating a block with certain witnesses in scope, if retained at all.

#### Lowering

Because concepts are effectively structs, the receivers created in name lookup are not quite correct: they invoke a non-static method on a type.

#### Errors

We add the following new errors:

|  |  |  |
| --- | --- | --- |
| Code | Internal name | Description |
| CS8953 | ERR\_ConceptConstraintOnNonImplicitParam | Type parameter A is constrained to concept C<T> but is not marked implicit. |
| CS8954 | ERR\_NonConceptConstraintOnImplicitParam | Type parameter A is marked implicit but has been constrained to I<T> which is not a concept. |
| CS8955 | ERR\_NoConstraintsOnImplicitParam | Type parameter A is marked implicit but has no type constraints. |
| CS8956 | ERR\_CantInferConceptInstance | Cannot infer a valid witness for concept C<T> at this location. |

### Case study: generic arithmetic

We have implemented a subset of the Haskell 98 prelude, including the numeric tower of concepts:

* Num: addition, subtraction, multiplication, sign, absolute, conversion from integer;
* Fractional: division, conversion from integer fraction;
* Floating: trigonometry, logarithm, exponentiation, hyperbolic functions.

These, in turn, were used in several generic arithmetic examples:

* An implementation of the Quickhull algorithm over Floating;
* An implementation of Conal Elliott’s Beautiful Differentiation automatic differentiation system, turning first and higher-order derivatives into implementations of each step of the numeric tower and allowing them to be manipulated by generic arithmetic;
* A small testbed expression used for checking the disassembly of generic arithmetic code.

## Associated Types

Our third feature set concerns associated types.

### Overview

Associated types are a feature of some concepts systems in which concepts can name, in addition to a set of functions for instances to supply, one or more type aliases for instances to define. These behave like additional implicit type parameters to the concept, but with differences:

* Associated types are never mentioned in explicit type parameter lists, and are only visible in concepts and their instances;
* Associated types are inferred at use site by allowing the choice of instance for whichever concept is naming the associated type to fix the type by

Our implementation of associated types is partially complete. Associated types in Concept C# must be mentioned in the type parameter list, and are denoted using an attribute instead of new type alias syntax. We have implemented the special case for associated types in type inference, but in an ad-hoc manner that has known issues.

### Syntax

There is no new syntax for associated types: they are handled using raw attributes. This is because we have not yet found a suitable syntactic design for them. The fully implicit style of associated types in most languages contradicts the explicit nature of C# type parameters.

Despite this, we have found uses for the partial implementation without syntax.

### Implementation

Associated types constitute an extension to type inference, including part-inference. Any type parameter marked with [AssociatedType]:

* Can be omitted for the purposes of part-inference;
* Can be left unfixed by type inference in the specific case that the associated type is a parameter on a concept that is about to be inferred itself during Form B member lookup (because the latter inference will fix the associated types);
* Can be fixed by type inference by applying the unifications used to fix concept witnesses in the same inference round (in other words, associated types can be inferred from being mentioned in instances successfully inferred in the same set of type arguments).

The latter change is the key one, and is presently implemented in a crude and buggy manner, but works well enough to allow for successful case studies.

### Case study: concept enumerators

With associated types, we can define concepts over enumerators and enumerables such that type inference is possible:

public concept CEnumerator<E, [AssociatedType] S>

{

void Reset(ref S enumerator);

bool MoveNext(ref S enumerator);

E Current(ref S enumerator);

void Dispose(ref S enumerator);

}

public concept CEnumerable<C, [AssociatedType] E, [AssociatedType] S>

{

S GetEnumerator(C container);

}

Unlike the existing interface system, CEnumerable inherits from CEnumerator. This means that both pieces of functionality may be provided by the same instance, as in the below example of a concept enumerator for arrays shows:

public instance CEnumerableArray<E> : CEnumerable<E[], E, (E[], int, E)>

{

(E[], int, E) GetEnumerator(E[] ary) => (ary, -1, default(E));

void Reset(ref (E[], int, E) enumerator)

{

enumerator.Item2 = -1;

enumerator.Item3 = default(E);

}

bool MoveNext(ref (E[], int, E) enumerator)

{

if (++enumerator.Item2 >= (enumerator.Item1.Length)) return false;

enumerator.Item3 = enumerator.Item1[enumerator.Item2];

return true;

}

E Current(ref (E[], int, E) enumerator) => enumerator.Item3;

void Dispose(ref (E[], int, E) enumerator) { }

}

The state of the enumerator can be any type: here, it is a value tuple of the array to be iterated, the current index, and the current element. The enumerator state can also be a scalar value type (such as an infinite enumerator over the natural numbers), a reference type, or generic and constrained by further concepts and interfaces.

We see the following advantages for concept-based enumerators over interface-based enumerators:

* It is easy to build up helpers for constructing enumerators. One example we have is to provide a CEnumerable instance for any type implementing further concepts CIndexable and CLength: any type that can be indexed with an integer and that has a finite length forms an enumerable based on progressively indexing from zero to that length.
* It is easy to derive enumerators from existing enumerators, with a minimal space and time footprint. For example, we have an instance  
    
  public instance CEnumerableZip2<A, [AssociatedType] AE, [AssociatedType] AS,  
   B, [AssociatedType] BE, [AssociatedType] BS,  
   implicit EA, implicit EB>  
   : CEnumerable<(A, B), (AE, BE), (AS, BS)>  
   where EA : CEnumerable<A, AE, AS>  
   where EB : CEnumerable<B, BE, BS>  
    
  that can lift any two enumerators into a zipped enumerator taking a value tuple of containers, returning a value tuple of items, and, crucially, using a value tuple of the enumerators’ state types.
* Given improvements to the tie breaking mechanism in ConceptWitnessInferrer, it should become easy to provide both generic enumerator instances for containers (or even a single enumerator for any class implementing IEnumerable) and specialized enumerator instances for specific constructions of those containers. This would also work well with the above point, to allow types for which there are multiple instances forming a sliding scale of performance to use the best enumerator where possible.

There are some disadvantages:

* Does not work with foreach syntax or yield return;
* Performance compared to BCL classes (arrays, strings) with existing implementations is poor, but it is currently unknown whether this is due to the optimised IEnumerators implemented for those classes or problems with our concepts.

## Default Implementations

We have a proof of concept for the implementation of ‘defaults’ in Concept C#: the ability to specify, as part of a concept, an implementation for a concept function that is used if and when an instance does not explicitly specify its own implementation.

The design of defaults is mostly finalized, but the implementation lacks polish: it breaks error reporting at levels above method compilation, requires invasive and non-robust changes to all levels of Roslyn, and relies on some assumptions that are not checked properly. If these issues are fixed, defaults may be worth including in a final version, but are low priority and can be mocked up with existing features.

### Syntax

By weakening some checks, we extend the syntax of concepts to allow methods to contain method bodies (expression or block oriented). These bodies are not compiled normally, but instead used to populate a hidden ‘default struct‘ containing the default implementations.

### Implementation

### Case study: Prelude

We again use our Haskell 98-style ‘prelude’ concepts as a case study here.

In Haskell 98, the concept **Eq** is used to denote types with a notion of equality. It is defined as follows:

class **Eq** *a* where  
    (==), (/=) :: *a* -> *a* -> Bool  
  
        -- Minimal complete definition:  
        --      (==) or (/=)  
    x /= y     =  not (x == y)  
    x == y     =  not (x /= y)

The definition makes use of defaults: equality is defined as the negation of inequality, and vice versa. This means that an instance need only define one (at the cost of overhead for the negation and call to the opposite method).

The analogous example in Concept C# is:

public concept Eq<A>

{

bool Equals(A x, A y) => !NotEquals(x, y);

bool NotEquals(A x, A y) => !Equals(x, y);

}

## Operator Overloads

C#’s existing operator overloads cannot be used with concepts, as they must be implemented using static methods on the type of one of the operands. Instead, as a proof of concept, we relax the constraints on operator overloads to allow them to be implemented on concepts as if they were a regular concept function.

This is highly efficient, appears to be stable, and has been successfully used in several examples. Operator overloads are key to the IArithmetic example that motivated concepts in the first place. However, it contradicts the existing C# design in confusing and cohesion-breaking ways, and also requires a way of implicitly bringing concept operator definitions into scope (such as Form C above). Therefore, operator overloads need rethinking from a design perspective before proceeding with a final implementation, if any.

### Syntax

The syntax is identical to the existing C# operator overloading syntax, except that operator overloading declarations are now permitted on concepts (with matching definitions on instances).

# Motivation and Comparison

## Existing Languages

We are motivated by the rise of concepts, under various names, in competitor languages. While languages such as Haskell 98 have always had concepts (typeclasses), recently there has been an explosion in languages closer to C#’s paradigm and use pattern supporting similar ideas:

* Rust has traits [1]: concepts with defaults, unnamed witnesses (trait bounds) and instances (impls), and C#-style syntax. It achieves performance by generating specialized code for trait-bound methods where possible (and uses boxed ‘trait objects’, containing type-erased data and vtable pointers, otherwise).
* Swift has protocols [2]: interfaces with support for writing extensions, which behave like instances, to add conformance to existing types, and optional requirements, which are similar to defaults (but put the onus of specifying the default behaviour on the caller). These are not quite concepts in our sense: the only non-instance method permitted is a special init method, and the default mode of implementation is inheritance.
* Scala has implicit parameters [3] [4]: 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. One key difference is that we pass witnesses as types rather than singleton objects, and use a special type declaration for concepts (which are just abstract classes in Scala). Our struct-type-based system gives stronger performance guarantees through JIT specialising.
* C++ currently has theoretical (documentation-level) concepts with no language support beyond the much more general template system. However, this system does not give use-site typechecking or separate compilation. Attempts have been made to create language support for concepts, but these have not been adopted.

Table 2

Table 2 Syntactic comparison of existing languages with concepts.

|  |  |  |  |
| --- | --- | --- | --- |
| Language | Concepts | Instances | Witnesses |
| Rust | Use keyword trait | Use keyword impl… for | Implicit (specialised code and type-erased ‘trait object’ boxes) |
| Swift | Closer to interfaces than concepts; use keyword protocol | Use inheritance syntax for pre-arranged implementations, and keyword extension… for | Interface-style: protocols are types in their own right |
| Scala | Abstract classes | Use keyword implicit object | Singleton objects using implicit object can be passed as special implicit parameters |
| Haskell 98 | Use keyword class | Unnamed, global, and unique per concept and type; use keyword instance | Implicit (additional dictionary value); use constraint syntax |
| Hackathon’16 F# | Use annotation [<Trait>] on interfaces | Explicitly named and scoped; use annotation [<Witness>] on classes | Explicitly named in type parameters: coerce to traits using when syntax, no specific keyword |
| C++14 | No language support besides templates (theoretical only) | No language support | No language support |
| Concept C# | Use keyword concept | Explicitly named and scoped; use keyword instance | Explicitly named in type parameters; use keyword implicitand where syntax |

## Static Interface Methods and IArithmetic

We are also motivated by the static interface methods proposal by Carol Eidt and David Detlefs (MS internal), itself motivated by the generic arithmetic (‘IArithmetic’) request from several customers.

# Language Interoperability

## F#

As a Hackathon 2016 project, a team at Microsoft Research Cambridge including ourselves (as well as Don Syme, James Clarke, and Rupert Horlick) built a prototype fork of Visual F# adding concepts to F#. The implementation level is a subset of the Concepts feature of Concept C#, but gains operator overloading ‘for free’ through F#’s existing operator overloading system.

Both prototype implementations use the same struct-based encoding of typeclasses and largely the same semantics. The main barrier to interoperability between F# and C# at the moment is that both prototypes ascribe semantics to a specific set of attributes, and both sets of concept attributes are disjoint. F#’s attributes lie in the FSharp.Core namespace, whereas C#’s attributes are in a separate assembly for now. To interoperate, one of the prototypes would have to accept the other’s attributes (which unreasonably ties the languages together), or a unified set of attributes would need to be used (which requires forwards planning unreasonable for prototype implementations). We could also weaken the check on attribute equality to name equivalence instead of symbol resolution.

## Visual Basic

We have not focused at all on Visual Basic in this prototype implementation. As it shares its compiler base in Roslyn with C#, it makes sense to open type classes up to Visual Basic, but this would involve a greater amount of syntax decisions to be made.

# Known Issues and Future Work

## Language Interactions

### Concepts vs. interfaces

Concepts are, semantically, just interfaces with access to defaults, operator overloading, and ability to become an implicit parameter. It is unclear as to whether this warrants a new syntactic form when we could just use interfaces. Conversely, there has been much discussion about whether concepts completely subsume interfaces (making them obsolete), which would pose tensions with regards to existing code using interfaces.

#### Solution

We could unify concepts with interfaces, but this would require some design to ensure the result is not unduly confusing (and the semantics of concept witnesses would need to change). Removing interfaces entirely is impossible due to backwards compatibility and interop, and phasing them out of the BCL is unlikely, so if concepts do subsume interfaces the way forwards is unclear.

### Concept witnesses vs. constraints on non-concepts

Concept witnesses cannot be constrained both by a concept and a non-concept syntactically, but there is nothing preventing this in separate compilation, and it is unknown whether this should be permitted or forbidden.

#### Solution

Decide (in conjunction with F#) whether to forbid these as present or accommodate them somehow.

### Defaults vs. member access on a direct instance

Calling into an instance directly which is missing an implementation method doesn't work: it should call into the default instead.

#### Solution

We should change name resolution to do a better job of this.

### Defaults vs. interfaces

Our defaults implementation uses little to no concept-specific properties, and could be generalised to interfaces by changing the witness type parameter into a formal ‘default-using this’ parameter added to the start of each default method.

#### Solution

We should generalise defaults to become interface defaults, with concepts falling out of this for free, and propose it separately as it would be a useful feature in its own right.

### Type inference vs. variance

There is no support for variance in the type inferrer: it will only substitute for direct matches of a type. This would be useful for allowing, for example, concept enumerables to be cast to a more general element type.

#### ***Solution***

We could overhaul the type inferrer to do variance-sensitive substitutions.

### Type inference vs. coercions

Similarly, the type inferrer cannot try witnesses whose types are an implicit cast from the types given, even if the witness type is an interface or superclass of the parameter type.

#### Solution

This would need a weaker unification system that allows coercions, and a 'weighting system' (or tie breaker) whereby the least coerced instance wins.

### Operator overloads vs. existing operator overloads

Operator overloads in Concept-C# are so different from C# operator overloads (instance vs static, different type rules) that this poses a poor interaction with existing C# features.

#### Solution

Unknown at this stage; this would need careful design.

### Scoping rules vs. existing scoping rules

Currently concept methods and operators are injected in front of class scope whenever they are mentioned in a where clause. This jars with the existing C# notion of scope (these new methods completely hide class members, for instance), and is confusing.

#### Solution

We could disable implicit uses of concept methods, but it is unclear what to do for operators. Alternatively, we could move to a more bespoke scoping system outside of binders, perhaps taking cues from extension method resolution.

## Future Work

We see the following future work between now and a formal proposal of the feature for C#:

* Decision on terminology. We have chosen the term concept due to its relationship to C++, but other terms (traits, protocols, implicits, typeclasses) may be more (or less) suitable. As mentioned before, the prototype F# implementation uses ‘trait’: interoperability concerns may mean that if F# proceeds to a formal language feature with this name, C# may want to follow suit. We take the term ‘instance’ from Haskell, but it already has a different meaning in C# that may cause confusion: perhaps ‘witness’, ‘concept map’, ‘model’, or ‘evidence’ would be better;
* A rewrite, or at least refactor, of the prototyped implementation. The implementation in Concept C# today suffers from a lack of knowledge about the idioms and conventions of Roslyn programming, and has many corner-cases, bugs, and implementation quirks that a clean rewrite may fix;
* Decisions on which language features are key, and which can be postponed or rejected;
* Expand defaults to work with interfaces in general;
* Solutions to the interoperability and interaction problems mentioned in this document.

# Bibliography

|  |  |
| --- | --- |
| [1] | Rust developers, “Traits,” [Online]. Available: https://doc.rust-lang.org/book/traits.html. |
| [2] | Apple Corporation, “The Swift Programming Language (Swift 2.2): Protocols,” [Online]. Available: https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift\_Programming\_Language/Protocols.html. |
| [3] | B. C. d. S. Oliveira, A. Moors and M. Odersky, “Type Classes as Objects and Implicits,” in *OOPSLA*, 2010. |
| [4] | Scala Developers, “Implicit Parameters - Scala Documentation,” [Online]. Available: http://docs.scala-lang.org/tutorials/tour/implicit-parameters. |