Concept-C# Design Issues

Claudio Russo Matt Windsor

Sunday 22 October 2017

This document collects all of the known open design issues for Concept-C#.

Core

This section contains known issues with the core of Concept-C#.

Syntax: Terminology

Concept-C# uses the names `concept` (from C++) and `instance` (from Haskell). These choices come with several issues:

- `concept` is often used as a meta-term (see: the amount of times we've mentioned 'the concept of `concepts`' by accident);
- `instance` is already used in C# to refer to an object of a certain class, and the Roslyn codebase is full of uses of this other term.

Each language that uses concepts has a different set of terms for these ideas. Haskell has `classes` (we can't use this for obvious reasons); Rust and Scala have `traits`; Swift uses `protocols`; Shapes is unique and uses `shapes`. A common theme is that concepts represent some *property* of types (their shape, their traits), which gets across the idea of concepts abstracting over more than just how to talk to a type ('interface', 'protocol').

The Shapes proposal made a good case for calling instances `extensions`: not only is this idiomatic given the reading of concepts as extension interfaces, but it matches the Swift terminology. Other terms in use include `impl` (Rust) and `implicit object` (Scala).

Semantics: Unification is incomplete

We decide whether or not we can accept an instance by first trying to unify our required concepts with its provided concepts, and then recursively fixing any remaining type

parameters. This is sound (modulo implementation bugs), but can't handle subtyping, variance, or even use constraints to inform its inference decisions.

Concepts are instances, and method type inference already has an engine for inferring instane parameters, so if we can work out how to connect it to our instance search and recursion/backtracking then we can probably replace unification entirely.

Performance: concept inference searches binder for all instances

Every time we run concept inference, regardless of whether we've seen the same problem before, we search up the binder hierarchy for all instances in scope, and then filter down from them.

This is a performance nightmare for multiple reasons:

- Searching for instances is around $O(mn)$, where m is the number of scopes (including `usings`) and n is the bound on the number of type members and type parameters at each scope (since we have to filter each type member to see if it's an instance and it's accessible from the binder);
- The first reduction we do on instances is unification, which is quite heavyweight (there are probably other things we can easy-out on);
- We do no storage of information about previous inference runs, so we can't re-use previous results (though doing this in a sound way seems non-trivial).

While we haven't profiled the inferrer, anecdotally it begins to slow down VS to a crawl when given sufficiently large jobs. Doing something better to trim down the initial search would be useful.

Architecture: Where to put concept witness inference?

Originally, concept witness inference (CWI) was separate to, and called after, phase 1 and 2 of method type inference. However, methods that combine associated types and delegates often need a feedback loop of concept inference into phase-2 method type inference and back again.

The current solution is to pose the looped problems as if they were fresh method calls with the already-fixed type parameters removed, and do a *full* re-run of method type inference. This is inefficient, semantically suspicious, and suggests we actually want to merge CWI into phase 2.

Things to consider

- Putting CWI into phase 2 would make replacing unification with interface lower-bound inference much easier, and thus help fix problems with variance and sub-

typing.

- CWI does recursion to fix parameters of generic instances, and backtracking when it can't. Any new version of CWI would need to handle this properly.
- CWI can be called from outside a method context, for example when inferring missing type parameters in a generic type.

Semantics: Concept inference with missing information

(Found by Simon Peyton Jones)

Consider the concept:

```
concept CEq<L, R>
{
    bool operator == (L l, R r);
}
```

Suppose we have three instances:

```
instance Eq_Int                     : CEq<int, int> {}

[Overlappable]
instance Eq_Array<A, B, implicit E> : Eq<A[], B[]> where E : CEq<A, B> {}

instance Eq_IntArray                : Eq<int[], int[]> {}
```

Suppose we also have two generic functions:

```
public bool Eq1<A, B, implicit E>(A[] xs, B[] ys) where E : CEq<A, B> =>
    xs == ys;
public bool Eq2<A, B, implicit E>(A[] xs, B[] ys) where E : CEq<A[], B[]> =>
    xs == ys;
```

Also suppose we call them as follows:

```
int[] xs, ys;
Eq1<int, int>(xs, ys);
Eq2<int, int>(xs, ys);
```

The implementation of == will differ from `Eq1` to `Eq2`. Why? This is because Concept-C# will eagerly infer the == in `Eq1` as coming from `CEq`, but the only available instance for `CEq<A[], B[]>` in scope is `Eq_Array<A, B, E>`. For `Eq2`, we know that `A = B = int`, and can pull in `Eq_IntArray`, which overlaps `Eq_Array`.

This is a coherence problem: we move from one successful inference to a different successful inference by moving type parameters around.

Defaults

This section contains issues caused by the defaults system.

Avoiding cyclic defaults

In Concept-C#, default implementations can refer to methods that, themselves, have default implmentations. For example

```
concept CEq<L, R>
{
    bool operator == (L l, R r) => !(l != r);
    bool operator != (L l, R r) => !(l == r);
}
```

Nothing stops someone from writing

```
instance FaultyEq<X, Y> : CEq<L, R> {}
```

and wondering why == generates a stack overflow when called on `X` and `Y`.

Solutions

Do nothing

We could push this problem to the library designer, giving them the responsibility for documenting which defaults must be overridden to prevent cycles. Haskell and Rust both take this approach.

Advantages

- No additional language design or features necessary
- Precedent in other languages (Haskell)

Disadvantages

- Issues appear at run-time, and are difficult to debug
- People who don't read the documentation won't know what to do to prevent cycles

Solution: no defaults

The easiest and most conservative solution would be to take defaults out entirely.

Advantages

- Simplest solution design-wise

Disadvantages

- Lose all of the features and convenience of defaults

Solution: minimum complete definition product-of-sums

We could let all concepts provide a set of maxterms specifying which methods must be overridden.

Possible syntax for this could look like:

```
concept COrd<L, R>
{
    require one
    {
        operator ==(L, R);
        operator !=(L, R);
    }
    // and
    require one
    {
        operator <(L, R);
        operator <=(L, R);
        operator >=(L, R);
        operator >(L, R);
    }

    operator ==(L l, R r) => !(l != r);
    operator !=(L l, R r) => !(l == r);
    operator  <(L l, R r) => l <= r && l != r;
    operator <=(L l, R r) => l <  r || l == r;
    operator >=(L l, R r) => l >  r || l == r;
    operator  >(L l, R r) => l >= r && l != r;
}
```

Advantages

- Complete solution to the problem.
- Compile-time check.

Disadvantages

- Unless combined with static analysis, onus is on concept designer to find all possible cycles.
- Very heavy on new syntax.
- Complex and hard to pick up and use.
- Lots of possible design space in terms of how to expose the idea to users.

Solution 4: defaults can't use other defaulted methods

Only allow default implementations to use methods that, themselves, are not given a default implementation.

This means that, for example.

```
concept CEq<L, R>
{
    bool operator ==(L l, R r) => !(l != r);
    bool operator !=(L l, R r) => !(l == r);
}
```

would be statically forbidden: either == or `!=` would need to lose its default.

Advantages

- Simple solution.
- Can be checked statically.
- No need for library designers to do any cycle detection

Disadvantages

- Forces the library designer to choose which alternative of several possibly equally implementable methods to require an implementation
- Could make instance implementation harder, eg. maybe it's easier to implement `!=` than == for certain types.
- Can't guard against the possibility to calling into a method that then calls into another default.

Variations

- Allow defaults to call other defaulted methods that are defined before it, which would be more flexible but also not very idiomatic for C#.

Associated Types

This section contains issues caused by associated types.

General situation

Associated types in Concept-C# right now are nowhere near a full implementation, either syntactically or semantically. They let us do a lot of things normal associated types would, like encoding complex concepts on enumerators and getting inference, but they need a lot of work before they're go-live.

Also, most of the bugs in the inferrer are due to the inferrer doing weird things in the face of associated types. This mostly stems from them being an ad-hoc extension to the original inferrer.

Propagated associated types aren't implicit

If a generic method or class needs to use an associated type of a concept, we need it to be pulled out into a new type parameter. This should be implicit, with syntactic sugar that allows it to be accessed as if it were a member of the concept witness. Instead, we force the user to give it a name, and also put an annotation on it.

For example, let's take our `CEnumerable` concept:

```
concept CEnumerable<TCollection, [AssociatedType]TEnum> { /* ... */ }
```

Suppose we want to make a derived instance for zipping up two enumerables into a tuple. We want to write something like:[1]

```
concept CEnumerable<TCollection, [AssociatedType]TEnum> { /* ... */ }
instance Zip2<TCollectionA, implicit EA,
              TCollectionB, implicit EB>
    : CEnumerable<(TCollectionA, TCollectionB), TEnum=(EA.TEnum, EB.TEnum)>
    where EA : CEnumerable<TCollectionA>
    where EB : CEnumerable<TCollectionB> { /* ... */ }
```

With even more syntax, resembling that from Järvi et al.'s OOPSLA'05 paper, we could do:

```
concept CEnumerable<TCollection, [AssociatedType]TEnum>
{
    type TEnum;
}
```

[1] If we followed the Shapes approach of giving concepts implicit 'self' type parameters, then it would probably be more natural to dot the associated types off those types, not the witnesses.

```
instance Zip2<TCollectionA, implicit EA,
              TCollectionB, implicit EB>
    : CEnumerable<(TCollectionA, TCollectionB)>
    where EA : CEnumerable<TCollectionA>
    where EB : CEnumerable<TCollectionB>
{
    type TEnum = (EA.TEnum, EB.TEnum)
}
```

Instead, we have to do something like:

```
instance Zip2<TCollectionA, [AssociatedType]TEnumA, implicit EA,
              TCollectionB, [AssociatedType]TEnumB, implicit EB>
    : CEnumerable<(TCollectionA, TCollectionB), (EA.TEnum, EB.TEnum)>
    where EA : CEnumerable<TCollectionA, TEnumA>
    where EB : CEnumerable<TCollectionB, TEnumB> { /* ... */ }
```

This contributes to a lot of the generic hell in our existing associated type examples. On the other hand, implicit type parameters add a lot of magic to the compiler, and hurt the link between C# and CIL.

No distinction between ground and propagated associated type parameters

We use the same annotation, `[AssociatedType]`, for both associated type declarations and extra type parameters that have been propagated from actual type parameters. The idea is that, if we have an `AssociatedType` to fix,

- we can only pick up a fixing from an instance if it was an `AssociatedType` on a concept;
- we can always unify one propagated `AssociatedType` with another `AssociatedType`, ie passing the buck for fixing it down the resolution.

This works in practice most of the time, but conflates two different types of associated type semantics. Worse, it means we can't store dependency information on propagated associated types (eg. which associated type they actually represent!), which will make it hard to improve inference and add syntax later on. We also can't compile-time-check that there is a valid dependency between a propagated associated type and something that is going to fix it.

Instead, we have some slightly ad-hoc logic to enforce the weaker conditions at the top, which stops things like the enumerator parameter of a `CEnumerator` accidentally fixing the enumerator of a `CEnumerable`, but there are invariably more and more bugs we haven't yet shaken out.

Possible solution

Maybe we need a new annotation, such as `AssociatedFrom(int conceptOrdinal, int typeOrdinal)`, that specifies the location of the exact `AssociatedType` or chained `AssociatedFrom` that we need to be fixed. This would be a pair of index into the current type parameter list *and* index into the *concept* type parameter the witness would

Applying this to our running example, we get:

```
concept CEnumerable<TCollection, [AssociatedType]TEnum> { /* ... */ }
instance Zip2<TCollectionA, [AssociatedFrom(2, 1)]TEnumA, implicit EA,
              TCollectionB, [AssociatedFrom(5, 1)]TEnumB, implicit EB>
    : CEnumerable<(TCollectionA, TCollectionB), (TEnumA, TEnumB)
    where EA : CEnumerable<TCollectionA, TEnumA>
    where EB : CEnumerable<TCollectionB, TEnumB>
```

(I've used ordinals here because they won't need any form of name lookup or binding to resolve from IL, but one problem is nested classes etc. pushing type parameters around. Maybe there is a better encoding, eg. ordinal offsets, fully qualified metadata names, etc.?)

One reason this hasn't been done yet is because, without syntax, it would be horrible to hand-encode.

Other thoughts

- Is there a way to encode these propagations without type parameters?
- If not right now, is there a way a CLR change could help us do this?