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;

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;
```

Then, given the calls

```
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
- Easy to enforce
- 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

Associated Types

This section contains issues caused by associated types.