# 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#.

## Defaults

### 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`.

#### Solution 1: 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 takes 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 2: 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 3: 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