Concept-C#: Concept Comparison

Claudio Russo Matt Windsor

Wednesday 25 October 2017

This document compares the Concept-C# system of concept (typeclass) based bounded polymorphism with that of other languages. It also compares Concept-C# to the Shapes proposal for C#.

NOTE: This document is under construction.

Introduction

In this document, we compare Concept-C# to several different concept systems:

- The C# *Shapes proposal*;
- Ordinary C# with hand-encoding[1] of the concepts system used in Concept-C# and Shapes;
- Haskell *type classes*, as implemented by the Glasgow Haskell Compiler;
- Rust *traits*;
- Swift *protocols*;
- Scala *implicit objects*.

We compare using the following criteria:

- The terminology each system uses;
- The system's support for various method conventions;

Terminology

Different implementations use different terms from those used in Concept-C#:

Concept-C#	Shapes	Typeclasses	Traits	Protocols	Implicit Objects
`concept`	`shape`	`class`	`trait`	`protocol`	`n/a`[2]
`instance`	`extension`[3]	`instance`	`impl`	`extension`[4]	`implicit object`

[1]Sometimes called 'the Russo device'.
[2]Scala seems to let concepts be any kind of abstract class-like construct?
[3]Shapes can include inline instances inside classes.
[4]Swift protocols can also include inline instances inside classes.

Method Support

We compare concept systems by the type of methods we can attach to concepts and instances. The categories are as follows:

- *Static*: methods that don't have or need object receivers—useful for abstracting over constructors and type-level properties.
- *Instance*: methods that take an object receiver by value—useful for simulating interfaces with concepts.
- *By-ref*: methods that take an object receiver by reference—useful for when the object is a value type, and by-value semantics would make a copy and thwart direct mutation.
- *Operator*: concepts can overload operators.

Static Methods

All implementations follow Haskell's lead and implement static methods. (Note that Scala instances are objects, so what we refer to as a static concept method in Scala would be an instance member of the implicit object. Though we hide it through syntax, this is how we implement concepts in Concept-C#.)

Examples

Each of these examples shows a concept with two static methods: one, `plus`, is a simple two-argument method, and the other, `zero`, is an abstracted zero-argument constructor.

Concept-C#

```
concept CMonoid<A>
{
    A Plus(A x, A y);
    A Zero();
}
```

Shapes

```
shape SMonoid<A>
{
    static A Plus(A x, A y);
    static A Zero();
}
```

C# with manual encoding

```
interface IMonoid<A>
{
    A Plus(A x, A y);
    A Zero();
}
```

GHC type classes

```
class Monoid a where
    plus :: a -> a -> a
    zero :: a
```

Rust Traits

```
trait Monoid { // implicit Self type parameter
    fn plus(x : Self, y : Self) -> Self;
    fn zero() -> Self;
}
```

Swift Protocols

```
protocol Monoid { // implicit Self type parameter
    static func plus(x: Self, y: Self) -> Self;
    static func zero() -> Self;
}
```

Scala Implicit Objects

```
trait Monoid [A] {
    def plus(x: A, y: A) : A;
    def zero() : A;
}
```

Instance Methods

Neither Haskell nor Scala have instance methods in concepts. Haskell doesn't have a concept of objects or methods—each function is always independent of data. Scala does, but its concept system is simpler than most.

This is another case in which Concept-C# and Shapes differ strongly. Concept-C# concept methods are static by default, so it instead borrows the *extension method* syntax to represent instance methods. This closely resembles the translation down to normal C#. Shapes defines instance methods just as they would be defined for classes, and uses an implicit type parameter to represent the 'this' object.

It is unclear how, in Shapes, we get hold of the 'this' type parameter. We discuss this later.

Examples

This example is from the Shapes proposal, and has an instance method where the receiver's type is only used in the receiver position. While some systems let the type be used elsewhere, this comes with restrictions.

Concept-C#

```
concept CComparable<X, Y>
{
    int CompareTo(this X x, Y y);
}
```

Shapes

```
shape SComparable<Y>
{
    int Plus(Y y); // X and x are implicit
}
```

C# with manual encoding

```
interface IComparable<X, Y>
{
    // Concept-C# puts an attribute here to mark this as a CEM;
    // the Shapes example just introduces the implicit parameters
    int CompareTo(X x, Y y);
}
```

Rust Traits

```
trait Comparable<Y> {
    fn compareTo(self, y : Y) -> Y;
}
```

Swift Protocols

Swift doesn't allow multi-parameter concepts, so we weaken the example to unify `Y` with the receiver type:

```
protocol Comparable {
    func compareTo(y: Self) -> Int;
}
```

By-ref Instance

Only Rust, and Concept-C# (through C# 7.2 `ref this`), implement by-ref instance methods.

Example

The standard Rust `Clone` trait does this to avoid shallow-copying things it's about to deep-copy:

```
pub trait Clone {
    fn clone(&self) -> Self;
}
```

The Concept-C# equivalent would be as follows:

```
public concept Clone<T>
{
    T clone(ref this T self);
}
```

Operator Overloads

Most of the languages we compare support operator overloads (though at present we are unsure whether Scala does).

Rust is unique in that it offers no direct syntax for overloading operators. Instead, it ties operators to certain traits—`std::ops::Add` governs `+`, for example—and defines the operators in terms of methods on those traits (`std::ops::Add.add`).

Examples

Concept-C#

```
concept CSemigroup<A>
{
    A operator +(A x, A y);
}
```

Shapes

```
shape SSemigroup<A>
{
    static A operator +(A x, A y);
}
```

C# with manual encoding

C# doesn't let us use actual `operator` declarations here, so we must desugar to a special method.

```
interface ISemigroup<A>
{
    A op_Addition(A x, A y);
}
```

GHC type classes

```
class Semigroup a where
    (+) :: a -> a -> a
```

Haskell also lets us define new operators, instead of overloading existing ones.

Swift Protocols

```
protocol Semigroup {
    static func +(x: Self, y: Self) -> Self;
}
```

Like Haskell, Swift lets us define new operators.

Summary

	Concept-C#	Shapes	Typeclasses	Traits	Protocols	Implicits
Static	Default	Yes: `static`	Default	Default	Yes: `static`	Default
Instance	Yes: CEMs	Default	N/A	Yes: `self`	Default	No
By-ref	Yes: CEMs	?	N/A	Yes: `&self`	N/A?	N/A?
Operators	Yes	Yes	Yes	Fixed traits	Yes	?

Implicit vs Explicit

Deciding how much of the concept infrastructure to expose to the user is a key trade-off in concept design. Making more things explicit lets the user intervene in ambiguities, gives the user more flexibility, and makes the relationship between concepts and their generated IL more understandable. However, concepts are a complex feature, and overwhelming the user with normally-pointless features and decision points makes them much harder to sell.

Things that are implicit in some systems, but explicit in others, include:

- The names of instances;
- The witness parameters passed into methods and classes depending on concepts;
- The receiver ('this') parameter of concept instance methods;
- The type parameter of the receiver (sometimes called `Self`) in concept instance methods.

Instance name

Some systems make the user explicitly name each instance. Making instances denotable helps with API stability, and lets the user easily manage overlapping instances, explicitly naming the intended instance where possible. In many cases, however, each concept and type parameter combination will have an obvious set of instances, and we can deal with ambiguities through other means (scoping, type wrappers, and so on).

While both Concept-C# and Shapes use explicitly named instances, the only one of our comparison languages that follows suit is Scala. This fits well with its story of instances as implicitly found objects.

Witness

While concept systems usually handle witnesses through some extra parameter or type parameter, few make this implementation detail explicit. The main advantages of this are allowing concept method calls to use the witness as a receiver (which may be more understandable than concepts bringing static methods into scope), and allowing the use multiple witnesses for the same concept and type pairing.

Scala uses explicit witnesses, and Concept-C# follows suit, but it is far more common to keep these implicit. Shapes differs from Concept-C# here: while shapes and concepts are both introduced into signatures by type parameter constraints, shapes constrain their target type and concepts constrain an 'implicit' witness parameter.

While not in our comparison, Coq's type classes system is an interesting example: witnesses always explicitly appear in the parameters to a definition using them, but the definition can call witness functions implicitly, and the parameter can omit a name for the witness.

Receiver Parameter

Haskell and Scala don't support instance-level methods, so they don't use implicit receivers.

Most concept systems that allow instance-level methods keep their receiver parameter implicit, which follows the way interfaces work in Java-style object orientation (right up to exposing it as `this` in methods). The exceptions are Concept-C#, which takes its cues from Scala, and Rust, which uses the presence or absence of a specifically named `self` parameter to distinguish between instance and static methods.

Receiver Type Parameter

Again, Haskell and Scala don't have instance-level methods.

All systems, except Concept-C#, have an implicit type parameter. All, except Shapes, expose it as `Self` for use in the concept or instance body. Shapes, unlike other systems, picks up an implicit receiver type parameter only if there are instance methods in the concept.

Summary

	Concept-C#	Shapes	Typeclasses	Traits	Protocols	Implicits
Instance name	Explicit	Explicit	Implicit	Implicit	Implicit	Explicit
Witness	Explicit	Implicit	Implicit	Implicit	Implicit	Explicit
Recv param	Explicit	`this`	Explicit	Explicit	`this`	N/A

	Concept-C#	Shapes	Typeclasses	Traits	Protocols	Implicits
Recv TP	Explicit	`This`[5]	Explicit	`Self`	`Self`	N/A

Other Features

Inline instances in types

Swift and Shapes both support the inline declaration of an instance inside a class.

Examples

Generic (multi-parameter) concepts

Examples

Generic (derived) instances

Examples

Associated types

Examples

Associated values

TBC: implicits, traits, protocols

Examples

Concept objects

Interfaces are types: when used directly as a type, they box over the actual implementation, using virtual dispatch to resolve members. Since this is a useful abstraction, languages where concepts completely replace interfaces tend to support using certain, well-behaved concepts as boxes.

Both Rust and Swift support this, with certain rules as to which concepts can be boxes (eg. no uses of the implicit type parameter except as the receiver of methods). Haskell does not support this, and neither does Concept-C#. Shapes seems to allow it in the case of using shapes to extend one specific type.

TBC: implicits

[5]Not sure whether this is exposed to the user, or just an implementation detail.

Examples

Summary

	Concept-C#	Shapes	Typeclasses	Traits	Protocols	Implicits
Inline instances	No	Yes	N/A[6]	N/A[7]	Yes	No?
Multi-TP concepts	Yes	Yes	Yes	Yes	No[8]	Explicit
Generic instances	Yes	No?	Yes	Yes	TBC	Yes?
Associated types	Partial[9]	No	Yes	Yes	Yes	Yes
Associated values	Partial[10]	No	N/A[11]	Yes?	No?	TBC
Concept objects	No	Partial[12]	No	Yes	Yes	TBC

[6] Haskell has no classes, so this feature would make no sense.
[7] All Rust methods use `impl` syntax, so this feature wouldn't help.
[8] Any extra type parameters must be associated types.
[9] As decorations on types; no support for implicit propagation.
[10] Properties cover some of the same use cases.
[11] Haskell has no distinction between values and functions, so ordinary typeclass functions are 'associated values'.
[12] Seemingly only when extending a specific type.