Classes for the Masses (Extended Abstract)

Claudio Russo
(Microsoft Research)

Matt Windsor
(University of York)

Don Syme
(Microsoft Research)

Rupert Horlick
(University of Cambridge)

James Clarke
(University of Cambridge)

ABSTRACT

Type classes are an immensely popular and productive feature of Haskell. They have since been adopted in, and adapted to, numerous other languages, including theorem provers. We show that type classes have a natural and efficient representation in .NET that paves the way for the extension of F#, C# and other .NET languages with type classes. Our encoding is type preserving and promises easy and safe cross-language inter-operation. We have extended the open source C# compiler and language service, Roslyn, with pervasive support for type classes and have prototyped a more minimalist design for F#.

1. INTRODUCTION

Haskell's *type classes* [10, 11] are a powerful abstraction mechanism for describing generic algorithms applicable to types that have different representations but common interfaces. A *type class* is a predicate on types that specifies a set of required operations by their type signatures with optional, default code. A type may be declared to be an *instance* of a type class, and must supply an implementation for each of the class' operations. Type classes may be arranged hierarchically, permitting subsumption and inheritance.

Many modern languages have adopted features inspired by type classes, with different implementation techniques. Scala has *implicits*[9], implicit method arguments denoting dictionaries, that are inferred by the compiler but represented, at run-time, as additional heap-allocated arguments to methods (with commensurate overhead). C++ came very close to adopting *concepts*[7], a rather different extension of the template mechanism, directly inspired by Haskell's type classes but enforcing compile-time code specialization for performance. Rust has *traits*[3]. Swift has *protocols*[4].

Contribution We describe a simple encoding that allows us to add type classes to any .NET language, allowing interoperable definitions of type classes. Our encoding relies on the CLR's distinctive approach to representing and compiling generic code[8, 12]. Unlike, for example, the JVM, the CLR byte-code format is fully generic (all source level type information, including class and method type parameters, are represented in the metadata and virtual instruction set). Parameterized code is JIT-compiled to type passing code, with type parameters having run-time representations as (second-order) values. The JIT compiler uses the reified types to generate specialized memory representations (for instantiated generic types) and specialized (and thus more efficient) code for generic methods. For example, scalar types and compounds of scalars called structs have natural unboxed representations familiar to C(++) programmers; generic array manipulating code will manipulate array elements without boxing when instantiated at scalar types. This run-time specialization allows the JIT to avoid the uniform (i.e. lowest-common-denominator) representations adopted by many implementations of ML, Haskell, the JVM and most dynamic languages.

Haskell compilers typically compile type classes using the so-called *dictionary translation*. The translation, guided by source types, inserts evidence terms that justify type class constraints. The evidence terms are dictionaries (i.e. records) of functions that provide implementations (and thus proofs) for all of the constraint's methods. Although similar to object-oriented virtual method tables, dictionaries are not attached to objects, but passed separately as function arguments. Because type classes are resolved statically, aggressive in-lining can remove most, but not all, indirection through dictionary parameters. This leads to efficient code with fewer indirect calls and leaner representations of values than full-blown objects. Objects, in contrast, must lug their method-tables wherever they go.

Given the obvious similarity between type passing and dictionary passing, it is perhaps not surprising that type passing forms an excellent implementation technique for Haskell's dictionary passing. This talk will give on overview of the technique that we are applying to provide efficient, interoperable type class implementations to both C# and F#.

2. THE REPRESENTATION

This section sketches our representation of the Haskell'98 type classes on .NET by example. For each example, we give the Haskell code, underlying .NET code in vanilla C#, and proposed F# syntax. We use vanilla C# as a more readable proxy for .NET intermediate bytecode and metadata.

Type Classes A Haskell type class, for example:

```
class Eq a where
  (==) :: a -> a -> Bool
```

is naturally represented in C# as:

```
interface Eq<A> { bool Equal(A a, A b); }
```

For F#, we use a `Trait`-attributed interface declaration:

```
[<Trait>]
type Eq<'A> = (* an interface *)
  abstract equal: 'A -> 'A -> bool
```

Haskell Overloads Haskell's declaration of class `Eq a` implicitly declares its members as overloaded operations:

```
(==) :: (Eq a) => a -> a -> Bool
```

Observe that the overloaded operation has a more general constrained type `(Eq a) =>`....

This generic operation is captured in C# by the method:

```
static bool Equal<A,EqA>(A a, A b) EqA: struct, Eq<A>
            =>  default(EqA).Equal(a, b);
```

This method has not one, but *two*, type parameters. The first, `A`, is just the type parameter from the declaration. The second, EqA, is a type parameter that is constrained to be a struct and is evidence for the constraint that A supports interface Eq<A>.

The use of the struct constraint on EqA is significant and subtle. Structs are stack-allocated so essentially free to create, especially when they contain no fields. Moreover, every struct type, including a type parameter T of kind struct, has a default (all-zero) value denoted by expression default(T). Invoking a method on a default value of reference type would simply raise a null-reference exception because the receiver is null. However, methods on structs (including interface methods) can always be properly invoked by calling the method on the struct's default value.

Thus an operation over some class can be represented as a static generic method, parameterized by an additional dictionary type parameter (here EqA). Derived operations with type class constraints can be represented by generic methods with suitably constrained type parameters. Finally, Haskell dictionary *values* correspond to C# dictionary *types*.

For F#, we do not overload a top-level Equal binding but, instead, allow qualifed access to trait members (e.g. Eq.equal), as in:

```
let equal = Eq.equal (* defines overloaded equal *)
```

Instances A Haskell instance declaration is represented by the declaration of an empty (field-less) .NET struct that implements the associated type class (itself an interface). This gives us a cheap representation of Haskell instances.

For example, the Haskell instance declaration:

```
instance Eq Integer where
  x == y =  x ‘integerEq‘ y
```

can be represented by the C# structure:

```
struct EqInt : Eq<int>
{  public bool Equal(int a, int b) => a == b; }
```

For F#, we use a Witness-attributed struct declaration:

```
[<Witness>]
type EqInt = (* a struct *)
  interface Eq<int> with member equal a b = a = b
```

Note that the F# syntax, unlike Haskell, names the instance as in the C# representation. In Haskell, instances are anonymous but names are useful for explicit disambiguation and interoperation with languages that cannot always rely on type argument inference (such as C#).

Derived Instances This Haskell code defines a family of derived instances: given an equality type a, it defines equality over lists of a.

```
instance (Eq a) => Eq ([a]) where
    nil == nil  = true
   a:as == b:bs = (a == b) && (as == bs)
      _ == _    = false
```

We can represent such a Haskell parameterized instance as a *generic* struct:

```
struct EqList<A, EqA> : Eq<List<A>>
  where EqA : struct, Eq<A> {
    public bool Equal(List<A> a, List<A> b) =>
          (a.IsNull && b.IsNull)
      || (a.IsCons && b.IsCons
           && default(EqA).Equal(a.Head,b.Head)
           && Equal(a.Tail,b.Tail)); }
```

This struct implements the interface Eq<List<A>>, but only when instantiated with a suitable type argument and evidence for constraint Eq<A>. Notice that EqList has, once again, an additional evidence type parameter EqA for constraint Eq<A>. Instantiations of the generic struct EqList<->, in turn, construct evidence for Eq<List<A>>.

For F# we use a parameterized Witness-declaration:

```
[<Witness>]
type EqList<'A,'EqA when 'EqA :> Eq<'A>> = (* a struct *)
  interface Eq<'A list> with
    member equal a b = match a,b with
      | a::l,b::m -> Eq.equal a b && Eq.equal l m
      | [],[] -> true | _,_ -> false
```

Other features We do not have space to describe the representations of other features but suffice to say that we can encode [5]: type class operations that themselves have constrained types in their signatures (using interface methods that are generic); type class hierarchies using interface inheritance; default operations using shared static methods; instances requiring polymorphic recursion; instances as data (to constrained term constructors) and multi-parameter type classes. Moreover, choosing to provide named rather than anonymous instances would allow us to selectively support explicit as well as implicit evidence when preferable. We cannot support higher-kinded type classes (like Monad), because .NET lacks higher-kinded abstraction. First-order associated types are in reach. For C#, evidence inference is a mild generalization of type argument inference, with instantiations derived from the pervasive and locally assumed concept hierarchy. For F#, we have adapted Haskell's more elaborate techniques for propagating inferred type class constraints, by extension of F#'s existing constraint system.

Implementations We prioritized our efforts on designing and implementing type classes for C# in a fork[1] of Microsoft's open source Roslyn compiler[2], adopting a dedicated syntax loosely inspired by C++ concepts[7]. The F# design and implementation [6] was the result of a 3-day hackathon aiming for a minimal viable product (with suboptimal syntax). Performance results are promising - we anticipated .NET's code specialization to turn virtual calls to dictionary members into direct calls, but the JIT exceeded expectations and aggressively inlined those calls. The JIT failed to eliminate dictionary arguments that became dead after inlining; hoist dictionary allocations out of loops or do CSE on dictionary values. Fortunately, the latter two are suitable compiler optimizations.

[1] *Roslyn concepts fork, https://github.com/CaptainHayashi/roslyn.*
[2] *Roslyn https://github.com/dotnet/roslyn.*
[3] *Rust traits https://doc.rust-lang.org/book/traits.html.*
[4] *Swift https://swift.org.*
[5] *https://github.com/CaptainHayashi/roslyn/blob/master/concepts/docs/concepts.md.*
[6] *https://github.com/CaptainHayashi/visualfsharp/tree/hackathon-vs.*
[7] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in c++. OOPSLA '06, pages 291–310.
[8] A. Kennedy and D. Syme. Design and Implementation of Generics for the .NET Common Language Runtime. PLDI '01, pages 1–12.
[9] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. OOPSLA '10, pages 341–360, 2010.
[10] S. Peyton Jones. *Haskell 98 language and libraries : the revised report.* Cambridge University Press, May 2003.
[11] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. POPL '89, pages 60–76.
[12] D. Yu, A. Kennedy, and D. Syme. Formalization of Generics for the .NET Common Language Runtime. POPL '04, pages 39–51.