Classes for the Masses (Extended Abstract)

Claudio Russo (Microsoft Research) Matt Windsor (University of York)

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. This talk will sketch that type classes have a natural and efficient representation in .NET. This paves the way for the extension of F# and other .NET languages with Haskell style type classes. The representation is type preserving and promises easy and safe cross-language inter-operation. We are currently, and rapidly, extending the open source C# compiler and language service, Roslyn, with support for type classes but intend to do the same for F# once that work has been completed.

1. INTRODUCTION

Haskell's *type classes* [9, 10] 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. 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. A default method may also be associated with an operation allowing its implementations to be omitted.

Many modern language have adopted features inspired by type classes, with different implementation techniques. Scala has *implicits*[8], 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*[6], 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*[4] and Swift has *protocols*[5].

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[7, 11]. 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 cla sses 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.

2.1 Haskell 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 Equals(A a, A b);
}
```

For new F# syntax we adopt the keyword `concept` (`class` is already taken):

```
concept Eq 'a where
    (==): 'a -> 'a -> bool
```

2.2 Haskell Overloads

The Haskell declaration of class `Eq a` implicitly declares the overloaded operations induced by class' members.

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

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

This generic operation can be represent in C# by the method:

```
public static bool Equals<A,EqA>(A a, A b)
   where EqA : struct, Eq<A> {
     return default(EqA).Equals(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`). Similarly, derived operations with type class constraints can be represented by generic methods with suitably constrained type parameters.

So a Haskell dictionary *value* corresponds to a C# dictionary *type*.

2.3 Haskell 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 Equals(int a, int b) { return a == b; }
}
```

For F#, we introduce the shorter `instance` declaration:

```
instance Eq int where
  Equals (a, b) = a = b
```

Note that the F# syntax, like Haskell, elides the name of the instance in the more explicit C# representation. In Haskell, instances are anonymous.

2.4 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 Equals(List<A> a, List<A> b) {
    return     (a.IsNull && b.IsNull)
             || (a.IsCons && b.IsCons
                  && default(EqA).Equals(a.Head,b.Head)
                  && Equals(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 adopt the more concise, nameless declaration:

```
instance Eq 'a => Eq (List 'a) where
  Equals(a,b) = match a,b with
    | [],[] -> true
    | (a::l),(b::m) -> Equals(a,b) && Equals(l,m)
    | _,_ -> false
```

2.5 Other features

We do not have space to describe the representations of other features but suffice to say that we can encode [1]: 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-kinds. First-order associated types are in reach.

Much of the concision of Haskell comes not only from the declaration of type class hierarchies but from the implicit solution of constraints during type inference. For C#, we envision evidence inference to be a mild generalization of type argument inference, with instantiations derived from the pervasive and locally assumed concept hierarchy. For F#, we hope to adapt Haskell's more elaborate techniques for not only solving but also implicitly propagating inferred type class constraints.

3. IMPLEMENTATION

We are currently prioritizing our efforts on designing and implementing concepts for C# in a fork[2] of Microsoft's open source Roslyn compiler[3], adopting a syntax loosely inspired by C++ concepts[6]. However, we hope to turn our attention to F# over the second half of the summer.

4. REFERENCES

[1] *A natural representation for type classes in .NET https://github.com/CaptainHayashi/roslyn/blob/master/concepts/docs/concepts.md*.

[2] *Roslyn concepts fork, https://github.com/dotnet/roslyn*.

[3] *Roslyn https://github.com/CaptainHayashi/roslyn*.

[4] *Rust traits https://doc.rust-lang.org/book/traits.html*.

[5] *Swift protocols https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Protocols.html*.

[6] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in c++. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 291–310, New York, NY, USA, 2006. ACM.

[7] A. Kennedy and D. Syme. Design and implementation of generics for the .net common language runtime. *SIGPLAN Not.*, 36(5):1–12, May 2001.

[8] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, New York, NY, USA, 2010. ACM.

[9] S. Peyton Jones. *Haskell 98 language and libraries : the revised report*. Cambridge University Press, May 2003.

[10] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.

[11] D. Yu, A. Kennedy, and D. Syme. Formalization of generics for the .net common language runtime. *SIGPLAN Not.*, 39(1):39–51, Jan. 2004.