Modules and Type Classes

Dave MacQueen
WG2.8, Park City, June 2008

S P-J: WG 2.8 meetings occasionally include "inflammatory and inconclusive discussions".

PL design is like architecture

scientific and engineering principles have a central role

also "esthetics"

- elegance and economy of design
  - orthogonality
  - design equivalent of Occam's razor (don't multiply solutions)
    - => avoid overlaps of functionality
      - n alternative ways of doing the same thing

- style
  - superficial syntax style
  - deeper architectural structure (selection and composition)

Modules vs Type Classes

An either-or choice?
Haskell users will immediately choose classes.
ML users will immediately choose modules/functors.
So what's the point?

Thought experiment: What if you were defining a brand new language, trying to merge the legacy from ML and Haskell.

- Common inheritance:
  HM type inference and polymorphism
  algebraic data types

- Would you include a typed module system?

- Would you include both functors and type classes?
  If so, how would they be related, how would they interact?

(S)ML	**Haskell**
HM type inference	ditto
parametric polymorphism	ditto
algebraic datatypes	ditto
strict evaluation*	lazy evaluation*
impure	pure
real state mutable data; imperative IO	IO monad
general exceptions	exceptions for IO monad
typed modules	untyped, name-space modules

(S)ML	**Haskell**
ad hoc overloading (fixed)	type classes
equality types & equality polymorphism	type classes
functors	type classes
(functors)	second-order polymorphism
(modules)	first-class polymorphism
—	existential types

The common problem addressed by functors and type classes

"Pure" parametric polymorphism (parametricity) is not enough.

Need to parameterized over types with an *interpretation*
interpretation : set of functions and values for creating or manipulating values of the type (i.e. "dictionaries")

E.g. a type with ordering

Note: an interpreted type is not necessarily an abstract type!
A type does not need to be opaque to benefit from an interpretation.

Primitive types normally come with a pervasively available interpretation via primitive functions, constants.

Both modules and type classes provide ways to provide a interpretation with a type (or types).

My original criticisms of type classes (from modules perspective)

1. only simple types (nullary type constructors) can have an interpretation as an instance of a type class

2. only single constructors can have interpretations

3. only one interpretation per constructor (e.g. only one Ord instance for Int)
   implicit global instance environment

4. can't parameterize with respect to a class/type if the type variable doesn't appear in the result (related to ambiguity)

5. because of use of type inference to determine type inference, flow of information is implicit:
  - implicit type abstraction
  - implicit data flow of dictionaries (interpretations)
  - implicit composition of classes to construct interpretation at an overloaded variable occurrence

Additional issues:

* Names of components of interpretations cannot be overloaded! (An overloaded variable can be a member of only one class.)

* Only atomic constructors can have interpretations

* Classes are orthogonal to type abstraction.

* Instances cannot introduce new types, they can only associate interpretations with existing types.

Classes win for equality (& show?)

Generic and polymorphic equality in SML is not what you want.
Equality needs to be specialized by type.

Equality as a type class allows more control, allowing special implementations for atomic type constructors (datatypes).

Deriving makes it convenient to provide default structural equality for new datatypes.

Evolution of classes

Features were added to classes (e.g. in GHC extensions)

(1) constructor classes (n-ary type constructors)

(2) multiple constructor classes

(3) associated types

(1) and (2) address corresponding problems with original class system.

These seem to narrow the functionality gap with functors.

Resolving class overloading

Suppose we have an occurrence of a component f of a class C (which must be unique). How do we resolve the meaning of f?

1. type checking determines the occurrence type ty of f.

2. ty is matched against the class type of f and the resulting instantiation is interpreted to construct an instance I of C.

3. the meanng of f is the f component of I (I.f).

Step 2 depends on an environment mapping atomic tycons to instances or instance operators.

Dreyer, Harper, Chakravarty: Classes ==> Modules

Modeling type classes and instances with modules

A natural translation:

```
class Eq a where
    == :: a -> a -> bool

signature Eq = sig
  type a
  val == : a -> a -> bool
end
```

Dreyer, Harper, Chakravarty: Classes ==> Modules

Modeling type classes and instances with modules

A natural translation:

```
class Eq a where
    == :: a -> a -> bool

signature Eq = sig
  type a
  val == : a -> a -> bool
end

[Another possibility -- parameterized signature:

signature Eq(type a) = sig
  val == : a -> a -> bool
end
]
```

Instances as Modules

```
instance Eq Int where
   == = <primitive == for Int>

structure EqInt : Eq = struct
  type a = int
  val == = Int.=
end

instance a :: Eq => Eq List a where
  [] == [] = true
  x:xs == y : ys = x == y && xs == ys

functor EqList (X : Eq) : Eq = struct
  type a = X.a list
  fun == ([],[]) = true
      l == (x::xs, y::ys) = X.==(x,y) andalso ==(xs,ys)
end
```

Is mapping into modules the best choice?

core language / module language

core computes values, described by types

modules compute both types and values

Are type classes part of the core, or part of the module language?

HS models core constructs using module constructs internally
E.g. datatypes, classes & instances
Not the only way, nor necessarily the best way.

Classes and Modules

In SML, everything lives in a module, and can be spec'd in a signature

Hence both classes and instances would be members of modules
and have specifications in signatures.

=> instances as well as classes must be named
(and can be referenced by paths)

What are the consequences?

* classes, a sort of signature, can be module components
what are their signature specs?

* if instances are modules, nothing to do (but module system
must be higher order (like SML/NJ)!)

* instance bindings are scoped as usual (?)

Instance scopes

If instances are named, exported by modules and scoped, how is the implicit tycon to instance environment managed?

Dreyer et all: using declarations

    using EqInt

Should these behave like other declarations.

    => local "bindings" of instances

Discussion

Adding some form of type classes and instances to an ML-like language seems plausible.

Probably should be a stripped down version, do minimize functional overlap with functors.

Many questions remaining:

- Module level or core level constructs?

- Scope of instances?

Text