Generics vs. Overloading [again]

- Parametric polymorphism:
  - Single algorithm may be given many types
  - Type variable may be replaced by any type
  - If \( f :: t \to t \) then \( f :: Int \to Int, f :: Bool \to Bool, ... \)

- Overloading
  - Single symbol may refer to more than one algorithm
  - Each algorithm may have different type
  - Choice of algorithm determined by type context
  - + has types \( Int \to Int \to Int \) and \( Float \to Float \to Float \), but not \( t \to t \to t \) for arbitrary \( t \)

Why overloading?

Many useful functions are not parametric

- Can \( \text{member} \) work for any list type?
  \( \text{member} :: [a] \to a \to \text{Bool} \)

  No! Only for types \( a \) for that support equality

- Can \( \text{sort} \) work for any list type?
  \( \text{sort} :: [a] \to [a] \)

  No! Only for types \( a \) that support ordering

- Can \( \text{serialize} \) work for any type?
  \( \text{serialize} :: a \to \text{String} \)

  No! Only for types \( a \) that support ordering

How you do this in OCaml/SML

The general always-works approach is have callers pass function(s) to perform the operations:

\[
\begin{align*}
\text{member} & :: (a \to a \to \text{Bool}) \to [a] \to a \to \text{Bool} \\
\text{member} \_ \_ = \text{False} \\
\text{member} \text{eqFun} (x:xs) v = \text{eqFun} x v \\
| | \text{member} \text{eqFun} xs v
\end{align*}
\]

Works fine but:
- A pain to thread the function(s) everywhere
- End up wanting a record of functions, a "dictionary"
- Now have to thread right dictionaries to right places
- Types get a little messier?

Acknowledgments

- Slide content liberally appropriated with permission from Kathleen Fisher, Tufts University
- She in turn acknowledges Simon Peyton Jones, Microsoft Research, Cambridge "for some of these slides"
- And then I probably introduced errors and weaknesses as I changed them [and added the material on the Monad type-class and wrote the accompanying code file]…
- Also note: This lecture relies heavily on lec9.hs
- Then onto OOP as a separate topic (acks not applicable)
Enter Type Classes

Type-classes are built-in support for implicit dictionary-passing

- Concise types to describe [records of] overloaded functions
- Sophisticated standard library of type classes for [all the] common purposes
- But nothing “privileged” in the library/language: Users can declare their own type classes (nothing special about ==, +, etc.)
- Interacts well enough with type inference [won’t study the “magic”]

And/but:
- Ends up “taking over the language and standard library’
- Lots of fancy features that are super-useful, but we’ll have time for just a quick exposure beyond the basics

Type Class Design Overview

- [Step 0: Do not try to compare these things to OOP classes and such; they are different. Will study OOP next.]
- Step 1: Type class declarations
  - Define a set of [typed] operations and give the set a name
  - Example: The \( \texttt{Eq a} \) type-class has operations \( == \) and \( /= \) both of type \( a \rightarrow a \rightarrow \text{Bool} \)
- Step 2: Instance declarations
  - Specify the implementations for a particular type
  - Examples: for \( \texttt{Int} \), \( == \) is integer equality, for \( \texttt{String} \), \( == \) is string equality (but could have decided case-insensitive)
- Step 3: Qualified types
  - Use qualified types to express that a polymorphic type must be an instance of your type class
  - Example: \( \texttt{member’} :: \textbf{Eq a} \Rightarrow \texttt{[a]} \rightarrow \texttt{a} \rightarrow \text{Bool} \)

Qualified types

\[
\text{member’} :: \textbf{Eq a} \Rightarrow \texttt{[a]} \rightarrow \texttt{a} \rightarrow \text{Bool}
\]

- Very roughly like a bound on the type variable
  - Caller must instantiate type variable with a type that is known to be an instance of the class
  - Callee may assume the type is an instance of the class (so can use the operations)
  - So “fewer” callers type-check and “more” callees type-check
- At run-time, the right dictionary will be implicitly passed and used
  - Call-site “knows which dictionary”
  - Calls in callee “use the dictionary”

More Examples

\[
\begin{align*}
\text{sort} & :: \textbf{Ord a} \Rightarrow \texttt{[a]} \rightarrow \texttt{[a]} \\
\text{reverse} & :: \texttt{[a]} \rightarrow \texttt{[a]} \\
\text{square} & :: \textbf{Num a} \Rightarrow \texttt{a} \rightarrow \texttt{a} \\
\text{squarePair} & :: (\textbf{Num a}, \textbf{Num b}) \Rightarrow (\texttt{a}, \texttt{b}) \rightarrow (\texttt{a}, \texttt{b}) \\
\text{stringOfMin} & :: (\textbf{Ord a}, \textbf{Show a}) \Rightarrow \texttt{[a]} \rightarrow \texttt{String}
\end{align*}
\]

Our own classes and instances

- The class declaration gives names and types to operations
- An instance declaration provides the operations’ implementations

\[
\begin{align*}
\textbf{class MyNum a where} \\
\text{plus’} :: a \rightarrow a \rightarrow a \\
\text{times’} :: a \rightarrow a \rightarrow a \\
\text{neg’} :: a \rightarrow a \\
\text{zero’} :: a
\end{align*}
\]

\[
\begin{align*}
\textbf{instance MyNum Int where} \\
\text{plus’} = (+) \\
\text{times’} = (*) \\
\text{neg’} = \lambda x \rightarrow -1 \cdot x \\
\text{zero’} = 0
\end{align*}
\]

\[
\begin{align*}
\textbf{instance MyNum Float where} \\
\text{plus’} = (+) \\
\text{times’} = (*) \\
\text{neg’} = \lambda x \rightarrow -1.0 \cdot x \\
\text{zero’} = 0.0
\end{align*}
\]

Then use them

- Use qualified types to write algorithms over overloaded operations

\[
\begin{align*}
\text{member’} :: \textbf{Eq a} \Rightarrow \texttt{[a]} \rightarrow \texttt{a} \rightarrow \text{Bool} \\
\text{member’} [] & = \text{False} \\
\text{member’} (x:xs) & = (==) x v || \text{member’} xs v \\
\text{double’} :: \textbf{MyNum a} \Rightarrow \texttt{a} \rightarrow \texttt{a} \\
\text{double’} v & = (\text{plus’} (\text{plus’} v v)) v \\
\text{sumOfSquares’} :: \textbf{MyNum a} \Rightarrow \texttt{a} \rightarrow a \\
\text{sumOfSquares’} [] & = \text{zero’} \\
\text{sumOfSquares’} (x:xs) & = \text{plus’} (\text{times’} x x) (\text{sumOfSquares’} xs)
\end{align*}
\]

\[
\begin{align*}
\text{i8} & = \text{double’} 4 \\
\text{f8} & = \text{double’} 4.0 \\
\text{yes} & = \text{member’} [3, 4, 5] 4 \\
\text{no} & = \text{member’} ["hi", "bye"] "foo"
\end{align*}
\]
Compositionality of functions

- Overloaded functions can be defined using other overloaded functions

```haskell
square :: Num a => a -> a
square x = x * x

quadAndFour :: Num a => a -> (a,Int)
quadrAndFour x = (square x * square x, square 2)
```

- `quadAndFour` “doesn’t know” what dictionary it was passed, but it knows which dictionary to pass to each of its calls to `square`

Compositionality of Instances

- Can use qualified instances to build compound instances in terms of simpler ones

```haskell
class Eq a where
  (==) :: a -> a -> Bool
instance Eq Int where
  (==) = intEq -- intEq primitive equality
instance (Eq a, Eq b) => Eq (a,b) where
  (==) (u,v) (x,y) = (u == x) && (v == y)
instance Eq a => Eq [a] where
  (==) []     []     = True
  (==) (x:xs) (y:ys) = x==y && xs == ys
  (==) _      _      = False
```

- A little more complicated example: see lec9.hs for

```haskell
instance MyNum a => MyNum (Complex a) ...
```

Subclasses

- Can specify “any instance of class `Foo` must also be an instance of class `Bar`”
  - Example: `Ord` a subclass of `Eq`
  - Example: `Fractional` a subclass of `Num`
    * `(Fractional` supports real division and reciprocals)*
  - Easy to define:

```haskell
class Eq a => Ord a where -- defines Ord a ...
```

- An instance must provide everything in the superclass (too)
- Makes a qualified type “provide more”

- This still isn’t OOP classes [we are defining and passing dictionaries separately and with static type resolution]

Default methods

- A class declaration can provide default implementations
  - Including in terms of other implementations
  - Instances can override the default or not
  - Example: not-equal as not of equal
  - Example: `>=` as `>` or `==`
  - Example: arbitrary result like 42

```haskell
-- Minimal complete definition: (==) or (/=)
class Eq a where
  (==) :: a -> a -> Bool
  x == y = not (x /= y)
  (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

- This still isn’t OOP classes [we are defining and passing dictionaries separately and with static type resolution]

No, really, it’s not OOP

- Dictionaries and method suites (vtables) are similar

But...

- As we have said:
  - Dictionaries “travel” separately from values
  - Method resolution is static in Haskell, based on types

- Also:
  - Constrains polymorphism, does not introduce subtyping
  - Can add instance declarations for types “retroactively”
  - Dictionary selection can depend on result types:

```haskell
fromInteger :: Num a => Integer -> a
```

Topics to skip

- Very useful for practical programming but a bit off our trajectory:
  - `deriving` to get automatic instances from data definitions
    - Example: Show a tree
  - Support for numeric literals using the `fromInteger` operation that lets you use `0, 3, 79,` etc. in any instance of `Num`
  - Interaction with type inference
    - Mostly “works fine”
    - Various details, including do not reuse operation names across classes in same scope
Now constructor classes

• Recall:
  - Int, [Int], Complex Int, Bool, Int -> Int, etc. are types
  - [-], Tree, etc. are type constructors (given a type, produce a type)

• We can define type classes for type constructors
  - Nothing really "new" here
  - Harder to read at first, but "arity" of the constructor is inferred from use in class declaration

• See Part 3 of lec9.hs for instances and uses of this example:

  class HasMap g where
  map' :: (a -> b) -> g a -> g b

Now back to monad

• Monad is a constructor class just like HasMap (!)
  - "Required" operations are >>= and return
  - Default operations for things like >>
  - IO is "just" one "special" instance of monad
  - There are many useful instances of monad
  - Any instance of monad can use do-notation since it's just sugar for calls to >>=

  • See Parts 4, 5, and 6 of lec9.hs to blow your mind 😄

Summary of all that (!) 😊

• "Part 4"
  – Monad is a constructor typeclass
  – Instance Monad Maybe’ gives intuitive definitions to >>= and return
  – do-notation for "maybe" can be much less painful than life without it

• "Part 5"
  – Naturally, can write code generic over "which monad instance"
  – So can reuse combinators like
    sequence :: Monad m => [m a] -> m [a]

• "Part 6"
  – State monad definition is purely functional but looks-and-feels like imperative programming when using it

Other cheats

• So type classes seem to work pretty well
  – Haskell has, over time, committed to them ever-more fully

• Without them, you can:
  – Do explicit dictionary passing
  – "Cheat" in various ways:
    • SML: special support for Eq and nothing else
      – Oh also +, *, etc. for int and float
    • OCaml: cheat on key functions like hash and = being allegedly fully polymorphic but can fail at runtime and/or violate abstractions
  – C++: OOP or code duplication, neither of which is the same??