Lecture 11: Overloading & Type Classes

CSC 131
Spring, 2019

Kim Bruce

Restrictions on ML/Haskell Polymorphism

- Type (a → b) →([a] → [b]) stands for:
  - ∀a. ∀b. (a → b) →([a] → [b])
- Haskell functions may not take polymorphic arguments. E.g., no type:
  - ∀b. ((∀a.(a → a)) →(b → b))
  - define: foo f (x,y) = (f x, f y)
  - id z = z
  - foo id (7, True) -- gives type error!
  - Type of foo is only (t -> s) -> (t, t) -> (s, s)

Restrictions on Implicit Polymorphism

Polymorphic types can be defined at top level or in let clauses, but can't be used as arguments of functions

```
id x = x
   in (id "ab", id 17)
```

OK, but can't write

```
test g = (g "ab", g 17)
```

Can't find type of test w/unification.
More general type inference is undecidable.

Explicit Polymorphism

Easy to type w/ explicit polymorphism:

```
test (g: forall t.t -> t) = (g "ab", g 17)
   in test (\t => \(x:t) -> x)
```

Languages w/explicit polymorphism:
Clu, Ada, C++, Eiffel, Java 5, C#, Scala, Grace

Explicit Polymorphism

- Clu, Ada, C++, Java
- C++ macro expanded at link time rather than compile time.
- Java compiles away polymorphism, but checks it statically.
- Better implementations keep track of type parameters.

Summary

- Modern tendency: strengthen typing & avoid implicit holes, but leave explicit escapes
- Push errors closer to compile time by:
  - Require over-specification of types
  - Distinguishing between different uses of same type
  - Mandate constructs that eliminate type holes
  - Minimizing or eliminating explicit pointers
- Holy grail: Provide type safety, increase flexibility

Polymorphism vs Overloading

- Parametric polymorphism
  - Single algorithm may be given many types
  - Type variable may be replaced by any type
    - Examples: hd, tail ::[t]->t , map::(a->b)->[a]->[b]
- Overloading
  - A 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 → Int → Int and Float → Float → Float, but not t→t→t for arbitrary t.

Why Overloading?

- Many useful functions not parametric
  - List membership requires equality
    - member: [w] -> w -> Bool (for "good" w)
  - Sorting requires ordering
    - sort: [w] -> [w] (for w supporting <,>,...)
- What are problems in supporting it in a PL?
  - Static type inference makes it hard!
  - Why are Haskell type classes a solution?

Overloading Arithmetic

- First try: allow fcns w/overloaded ops to define multiple functions
  - square x = x * x
    - versions for Int -> Int and Float -> Float
  - But then
    - squares (x,y,z) = (square x, square y, square z)
    - ... has 8 different versions!!
  - Too complex to support!

ML & Overloading

- Functions like +, * can be overloaded, but not functions defined from them!
  - 3 * 3 -- legal
  - 3.14 * 3.14 -- legal
  - square x = x * x -- Int -> Int
  - square 3 -- legal
  - square 3.14 -- illegal
  - To get other functions, must include type:
    - squaref (x:float) = x * x -- float -> float

Equality

- Equality worse!
  - Only defined for types not containing functions, files, or abstract types -- *why?*
  - Again restrict functions using ==
- ML ended up defining eq types, with special mark on type variables.
  - member: "a -> ["a] -> Bool
  - Can't apply to list of functions

Type Classes

- Proposed for Haskell in 1989.
- Provide concise types to describe overloaded functions -- avoiding exponential blow-up
- Allow users to define functions using overloaded operations: +, *, <, etc.
- Allow users to declare new overloaded functions.
  - Generalize ML's eqtypes
  - Fit within type inference framework

Recall ...

- Definition of quicksort & partition:

```
partition lThan (pivot, []) = ([],[])
partition lThan (pivot, first : others) =
  let
    (smalls, bigs) = partition lThan (pivot, others)
  in
    if (lThan first pivot)
      then (first:smalls, bigs)
      else (smalls, first:bigs)
```

- Allowed partition to be parametric
  - Steal this idea to pass overloaded functions!
  - Implicitly pass argument with any overloaded functions needed!!

Example

- Recall

```
class Order a where
  (<) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  ...
```

- Implement w/dictionary:

```
data OrdDict a = MkOrdDict (a -> a -> Bool)
                           (a -> a -> Bool) ...

getLT (MkOrdDict lt gt ...) = lt

getGT (MkOrdDict lt gt) = gt
...
```

Using Dictionaries

```
partition dict (pivot, []) = ([],[])
partition dict (pivot, first : others) =
  let
    (smalls, bigs) = partition dict (pivot, others)
  in
    if ((getLT dict) first pivot)
      then (first:smalls, bigs)
      else (smalls, first:bigs)

partition:: OrdDict a -> [a] -> ([a].[a])
```

Compiler adds dictionary parameter to all calls of partition.

Reports type of partition (w/out lThan parameter) as
(Ord a) => [a] -> ([a].[a])

Instances

- Declaration
  - instance Show TrafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"
  - Creates dictionary for “show” w/def. above

Implementation Summary

- Compiler translates each function using an overloaded symbol into function with extra parameter: the *dictionary*.
- References to overloaded symbols are rewritten by the compiler to lookup the symbol in the *dictionary*.
- The compiler converts each type class declaration into a *dictionary* type declaration and a set of *selector* functions.
- The compiler converts each instance declaration into a *dictionary* of the appropriate type.
- The compiler rewrites calls to overloaded functions to pass a *dictionary*. It uses the static, qualified type of the function to select the dictionary.

Multiple Dictionaries

- Example:
  - squares :: (Num a, Num b, Num c) => (a, b, c) -> (a, b, c)
  - squares(x,y,z) = (square x, square y, square z)
- goes to:
  - squares (da,db,dc) (x, y, z) =
    (square da x, square db y, square dc z)

Compositionality

- Build compounds from simpler
  - class Eq a where
    (==) :: a -> a -> Bool
  - instance Eq Int where
    (==) = intEq -- *where 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

Subclasses

- Example:
  - class (Eq a) => Num a where
    (+) :: a -> a -> a
    ... *other arith ops*
    fromInteger :: Integer -> a
    - instance of Num a must be Eq a
    - dictionary for Eq is part of that for Num

What about Literals?

- fromInteger in Num class makes it possible
  - data Complex a = MkCmpx a a *deriving Eq*
  - instance Show a => Show (Complex a) where
    show (MkCmpx rv iv) = (show rv)++" + "++ (show iv)++"i"
  - instance Num a => Num (Complex a) where
    (MkCmpx r1 i1) + (MkCmpx r2 i2) =
    MkCmpx (r1+r2) (i1+i2)
    ...
    fromInteger n = MkCmpx (fromInteger n) 0
  - fromInteger will be called implicitly when needed

Using Literals

- Example:
  - c1 = 1 :: Complex Int
  - c2 = 2 :: Complex Int
  - c3 = MkCmpx 1 3
  - c4 = c1 + c3
  - c5 = c1 * c2
  - c6 = c3 + 47

Type Inference

- Type inference infers a qualified type Q => T
  - T is a Hindley Milner type, inferred as usual
  - Q is set of type class predicates, called a constraint
- Consider the example function:
  - example z xs = case xs of
    [] -> False
    (y:ys) -> y > z || (y==z && ys == [z])
  - Type T is a -> [a] -> Bool
  - Constraint Q is { Ord a, Eq a, Eq [a]}

Simplifying Constraints

- Constraint sets Q can be simplified:
  - Eliminate duplicates
    - {Eq a, Eq a} simplifies to {Eq a}
  - Use an instance declaration
    - If we have instance Eq a => Eq [a], then {Eq a, Eq [a]} simplifies to {Eq a}
  - Use a class declaration
    - If we have class Eq a => Ord a where ..., then {Ord a, Eq a} simplifies to {Ord a}
- Thus, {Ord a, Eq a, Eq[a]} simplifies to {Ord a}

Inference

- As a result:
  - example z xs = case xs of
    [] -> False
    (y:ys) -> y > z || (y==z && ys == [z])
  - Type T is a -> [a] -> Bool
  - Constraint Q is { Ord a, Eq a, Eq [a]}, which simplifies to {Ord a}
  - So, example :: (Ord a) => a -> [a] -> Bool

Reporting Errors

```
*Main> 'a' + 1
<interactive>:1:0:
    No instance for (Num Char)
      arising from a use of `+' at <interactive>:1:0-6
    Possible fix: add an instance declaration for (Num Char)
    In the expression: 'a' + 1
    In the definition of `it': it = 'a' + 1
```

- *Why this error message?*

Type Classes w/Constructors

- Recall Functor class:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor ([]) where
  fmap = map
```

  - [] here means operator that takes a type and makes it into a list type
  - f is a type function, not a type! Monads similar!

Trees are functors too!

```
data Tree a = Niltree | Maketree (a, Tree a, Tree a)
                deriving Show

instance Functor Tree where
  fmap f Niltree = Niltree
  fmap f (Maketree (root, left, right)) =
        Maketree (f root, fmap f left, fmap f right)
```

Type Classes ≠ OOP Classes

- Dictionaries and method suites are similar
  - In OOP, a value carries a method suite.
  - With type classes, the dictionary travels separately
- Method resolution is static for type classes, dynamic for objects.
- Dictionary selection can depend on result type
  - fromInteger :: Num a => Integer -> a
- Based on “ad hoc” polymorphism
  - like use of interfaces in Java.

Type Inference Oddity

- Type inference algorithms have difficulties with polymorphism
  - Already seen can’t have polymorphic params
  - Worse when use type classes

Weird Example

```
sqr x = x * x
h = sqr
*Main> :t sqr
sqr :: (Num a) => a -> a
*Main> :t h
h :: Integer -> Integer
*Main> h 1.4
        Crash!!!!
```

```
s x = sqr x
*Main> :t s
s :: (Num a) => a -> a
*Main> s 1.4
        1.96
```

η-expansion often solves problem
Alternatively, declare h with full type.

h defined in interactive mode is fine!

Ack!!

- The monomorphism restriction is probably the most annoying and controversial feature of Haskell's type system. All seem to agree that it is evil - it is commonly called "The Dreaded Monomorphism Restriction" - but whether or not it is considered a necessary evil depends on who you ask.
  - wiki.haskell.org/Monomorphism_restriction

See the article for a better discussion

The monomorphism restriction

Rule 1.
We say that a given declaration group is unrestricted if and only if:
(a):
every variable in the group is bound by a function binding or a simple pattern binding (Section 4.4.3.2), and
(b):
an explicit type signature is given for every variable in the group that is bound by simple pattern binding.
The usual Hindley-Milner restriction on polymorphism is that only type variables that do not occur free in the environment may be generalized. In addition, the constrained type variables of a restricted declaration group may not be generalized in the generalization step for that group. (Recall that a type variable is constrained if it must belong to some type class; see Section 4.5.2.)

Rule 2.
Any monomorphic type variables that remain when type inference for an entire module is complete, are considered ambiguous, and are resolved to particular types using the defaulting rules (Section 4.3.4).

Often prevent problems by writing type explicitly!

Scope -- a bit out of order ...

Scope

- Range of instructions where identifier is known
- Static: Scope associated with static text of program.
- Dynamic: Scope associated with execution path of program.

Hole in Scope (Static)

```
program ...
    var M: integer;
    ....
    procedure A ...
        var M: array [1..10] of real;
        begin
            ...
        end;
begin
    ...
end.
```

Static vs Dynamic Scope

```
program ...
    var A : integer;

    procedure Y(B: integer);
        begin
            ...;
            B := A + B;
            ...
        end; {Y}
```

which A?

```
    procedure Z(...);
         var A: integer;
         begin
             ...;
             Y(...);
             ...
         end; {Z}
    begin {main}
         ...;
         Z(...);
         ...
    end.
```

Symbol Table: Built compile-time or run-time?