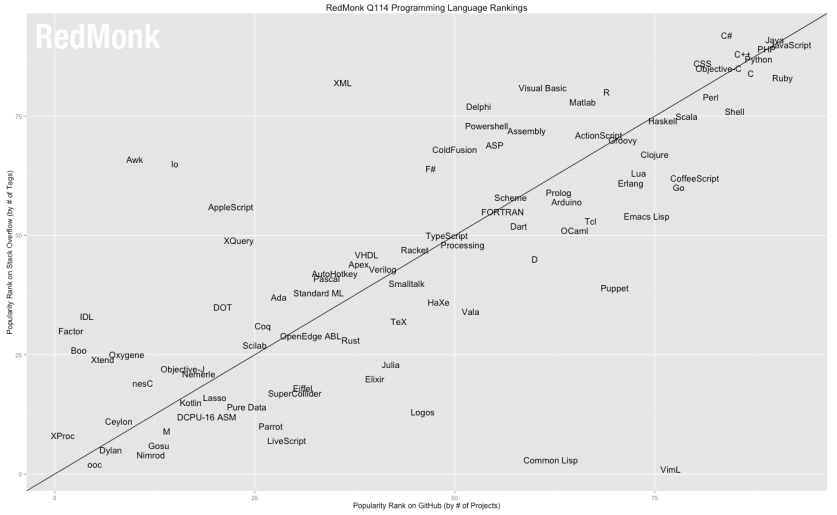
Lecture 15: Type Inference, polymorphism & Type Classes

CSC 131
Fall, 2014

Kim Bruce

Programming Language Rankings



Top Combined

1. JavaScript (+1)
2. Java (–1)
3. PHP
4. C# (+2)
5. Python (–1)
6. C++ (+1)
7. Ruby (–2)
8. C
9. Objective-C
10. CSS (new)
11. Perl
12. Shell (–2)
13. Scala (–1)
14. Haskell
15. R (1)
16. Matlab (+3)
17. Clojure (+5)
18. CoffeeScript (–1)
19. Visual Basic (+1)
20. Groovy (–2)

Tiobe Index

Sep 2014	Sep 2013	Change	Programming Language	Ratings	Change
1	1		C	16.721%	-0.25%
2	2		Java	14.140%	-2.01%
3	4	⌃	Objective-C	9.935%	+1.37%
4	3	⌄	C++	4.674%	-3.99%
5	6	⌃	C#	4.352%	-1.21%
6	7	⌃	Basic	3.547%	-1.29%
7	5	⌄	PHP	3.121%	-3.31%
8	8		Python	2.782%	-0.39%
9	9		JavaScript	2.448%	+0.43%
10	10		Transact-SQL	1.675%	-0.32%
11	11		Visual Basic .NET	1.532%	-0.31%
12	12		Perl	1.369%	-0.32%
13	13		Ruby	1.281%	-0.10%
14	-	⌃⌃	Visual Basic	1.272%	+1.27%
15	14	⌄	Delphi/Object Pascal	1.157%	+0.26%
16	26	⌃⌃	F#	0.990%	+0.49%
17	15	⌄	Pascal	0.893%	+0.01%
18	-	⌃⌃	Swift	0.852%	+0.85%
19	19		MATLAB	0.818%	+0.18%
20	17	⌄	PL/SQL	0.809%	+0.13%

Scala #41
Haskell #44
ML #25
Go #38
Dart #42
Scheme #48

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”

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