Overloading, Type Classes, and Algebraic Datatypes

Delivered by Michael Pellauer

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

September 28, 2006

Last Time...

- Type Inference Rules

(App)
$$\frac{TE \vdash e_1 : \tau \rightarrow \tau' \qquad TE \vdash e_2 : \tau}{TE \vdash (e_1\ e_2) : \tau'}$$

(Abs)
$$\frac{TE + \{x : \tau\} \vdash e : \tau'}{TE \vdash \lambda x.e : \tau \rightarrow \tau'}$$

...

- Type Inference Algorithm

Def W(TE, e) = *Case* e *of* ...
$\lambda x.e$ = *let* (S_1, τ_1) = W(TE + { x : u }, e);
in $(S_1, S_1(u) \rightarrow \tau_1)$
$(e_1\ e_2)$ = *let* (S_1, τ_1) = W(TE, e_1);
(S_2, τ_2) = W(S_1(TE), e_2);
S_3 = Unify($S_2(\tau_1)$, $\tau_2 \rightarrow$ u);
in $(S_3\, S_2\, S_1,\ S_3(u))$...

Last Time...

- Hindley-Milner Type System
  - Allows ForAll Generalization and Instantiation

- What's the type of:

```
fst x y = x
snd x y = y

complicated x = fst (snd x)
```


Aside: Type Annotations

- When programming in Haskell, we could let the compiler infer all the types
- However this can be difficult for a human to read. Consider:

```
c f g = \x -> g (f x)
```

vs

```
c :: (a -> b) -> (b -> c) -> (a -> c)
c f g = \x -> g (f x)
```

- Type signatures serve as documentation and are as important as a good function name

Inference Algorithm with Annotations

- Question 1: Are annotations an assertion, or a hint?
  - In Haskell, they are an assertion
- Question 2: How should the inference algorithm change?
  - First approach: Add annotations as a constraint
  - Second approach: Attempt to unify inferred type with given type
- Question 3: What happens if the user's annotation is too general? Too specific?
  - Too general: an error
  - Too specific: The user-given type becomes the actual type of the function

Conclusion: Type Inference

- Now you understand the Haskell type system!
  - ...Almost
- The Hindley-Milner system allows for *parametric* polymorphism
  - Similar to Java Generics
- Haskell also features a second type of polymorphism: Overloading
  - Somewhat similar to Java virtual functions

Overloading *ad hoc polymorphism*

A symbol can represent multiple values each with a different type. For example:

\+ represents

```
plusInt   :: Int -> Int -> Int
plusFloat :: Float -> Float -> Float
```

The *context* determines which value is denoted.

The overloading of an identifier is *resolved* when the unique value associated with the symbol in that context can be determined.

Compiler tries to resolve overloading but sometimes can't. The user must declare the type explicitly in such cases.

Overloading vs. Polymorphism

Both allow a single identifier to be used for multiple types.

However, the two concepts are very different:

1. A polymorphic function represents a *single function* that works for many types.

   Overloading uses the same name for several different functions.

2. All specific types of a polymorphic identifier are instances of a *most general type*

The Most General Type

The most general type of “twice f = \x -> f (f x)” is

```
∀t.(t -> t) -> (t -> t)
```

Any type can be substituted for **t** to get an instance of **twice**:

```
(Int -> Int)       -> (Int -> Int)
(String -> String) -> (String -> String)
```

Overloaded + does not have a most general type:

```
plusInt   :: Int -> Int -> Int
plusFloat :: Float -> Float -> Float
```

Has + type ∀**t. t -> t -> t ?**

No! + makes sense for some types t, but not for all!

Handling Overloading

- Not a problem in explicitly typed languages: the compiler has enough context information to resolve the overloading.
- Not a problem in OO languages (e.g., Java) where objects carry their type at runtime, and dynamic dispatch is possible.
- Hard to integrate in languages that use type inference
  - ML: ad-hoc support for limited cases (==)
  - Haskell: real solution – type classes
    Allows overloading of user-defined symbols

Type Classes

Type classes group together related functions (e.g., +, -) that are overloaded over the same types (e.g., Int, Float):

```
class Num a where
   (==), (/=)          ::   a -> a -> Bool
   (+), (-), (*)       ::   a -> a -> a
   negate              ::   a -> a
   ...

instance Num Int where
    x == y             = integer_eq x y
    x + y              = integer_add x y
      …
instance Num Float where ...
```


Overloaded Constants

(Num t) is read as a predicate
"**t** is an instance of class **Num**"

```
sqr  ::  (Num a) => a -> a
sqr x = x * x
```

What about constants? Consider

```
plus1 x = x + 1
```

If **1** is treated as an integer then **plus1** cannot be overloaded. In pH numeric literals are overloaded and considered a short hand for
(fromInteger *the_integer_1_value***)**
where
fromInteger :: (Num a) => Integer -> a

The Equality Operator

- Equality is an overloaded function, not a polymorphic one

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  a /= b     = not (a == b)
```

  - Equality needs to be defined for each type of interest.
  - Default definition for /=
  - Smart compilers can derive the code for structural equality

Type Class Hierarchy

```
class (Eq a) => Ord a where
  (<),(<=),(>=),(>) :: a -> a -> Bool
        max, min    :: a -> a -> a
```

- Eq is a superclass of Ord:
  - If type a is an instance of Ord, a is also an instance of Eq
- Ord inherits the specification of (==), (/=) from Eq

Read and Show Functions

The raw input from a keyboard or output to the screen or file is usually a string. However, different programs interpret the string differently depending upon their type signature.

A program to calculate monthly mortgage payments may assign the following signatures:

```
read :: String -> Int     - principal, duration
read :: String -> Float   - rate
show :: Float  -> String  - monthly payments
```

what is the type of **`read`** and **`show`** ?

```
read :: String -> a
show :: a      -> String
```

Polymorphic ?

Overloaded Read and Show

Haskell has a type class **Read** of "readable" types and a type class **Show** of "showable" types

```
read :: Read a => String -> a
show :: Show a => a      -> String
```

Ambiguous Overloading

```
identity   :: String -> String
identity x = show (read x)
```

What is the type of **`(read x)`** ?

Cannot be resolved ! Many different types would do.

Compiler requires type declarations in such cases.

```
identity   :: String -> String
identity x = show ((read x) :: Int)
```


Implementation

How does sqr find the correct function for * ?

```
sqr   ::  (Num a) => a -> a
sqr x = x * x
```

An overloaded function is compiled assuming an extra “dictionary” argument.

```
sqr’ = \class_inst x ->
                 (class_inst.(*)) x x
```

Then **`(sqr 23)`** will be compiled as

```
sqr’ IntClassInstance 23
```

Most dictionaries can be eliminated at compile time by function specialization.

Haskell Type Classes vs. Java Classes

- Similarities
  - Group together common sets of operations
  - Class hierarchy: super/sub-classes, inheritance
  - Dictionaries ≈ virtual method tables (vtables)

- Differences
  - The instance of a type class is a type, while the instance of a class is an object; types ≠ objects
  - No notion of mutable state in Haskell
  - In Java, objects carry "dictionaries" (vtables); in Haskell, dictionaries are separate from values (connected by the type system)

Aside: Static vs Dynamic Typing

- What is the type of x?
  - let x = 5
  - let x = False
  - let x = if False then 5 else False
  - let x = if readBoolFromUser() then 5 else False
- Haskell is a Statically typed language
  - Types must be determinable at compilation time
- Scheme, Lisp are dynamically typed
  - Values are tagged with types at runtime and dynamically checked
- In Haskell, one represents dynamic choice between different types with *algebraic datatypes*

Algebraic datatypes

- Algebraic types are *tagged unions of products*
- Example

```
data Shape = Line      Pnt  Pnt
           | Triangle  Pnt  Pnt  Pnt
           | Quad      Pnt  Pnt  Pnt  Pnt
```

keyword

"union"

new type

"products" (fields)

- new "constructors" (a.k.a. "tags", "disjuncts", "summands")
- a k-ary constructor is applied to k type expressions

Examples of Algebraic datatypes

```
data Bool = False | True
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
data Maybe a  = Nothing | Just a
data List a = Nil | Cons a (List a)
data Tree a = Leaf a | Node (Tree a) (Tree a)
data Tree' a b = Leaf' a
               | Nonleaf' b (Tree' a b) (Tree' a b)

data Course  = Course String Int String (List Course)
```

name number description pre-reqs

Constructors are functions

- Constructors can be used as functions to *create* values of the type

```
let
    l1 :: Shape
    l1 = Line   e1   e2

    t1 :: Shape = Triangle  e3  e4  e5
    q1 :: Shape = Quad   e6  e7  e8  e9
in
    ...
```

where each "eJ" is an expression of type "Pnt"

Pattern-matching on algebraic types

- *Pattern-matching* is used to examine values of an algebraic type

```
anchorPnt :: Shape -> Pnt
anchorPnt s = case s of
                Line     p1 p2       -> p1
                Triangle p3 p4 p5    -> p3
                Quad     p6 p7 p8 p9 -> p6
```

- A pattern-match has two roles:
  - A test: "does the given value match this pattern?"
  - Binding ("if the given value matches the pattern, *bind* the variables in the pattern to the corresponding parts of the value")

Pattern-matching *scope & don't cares*

- Each clause starts a new *scope*: can re-use bound variables
- Can use "don't cares" for bound variables

```
anchorPnt :: Shape -> Pnt
anchorPnt s = case s of
                Line     p1  _       -> p1
                Triangle p1  _  _    -> p1
                Quad     p1  _  _  _ -> p1
```


Pattern-matching *more syntax*

- Functions can be defined directly using pattern-matching

```
anchorPnt :: Shape -> Pnt
anchorPnt (Line     p1 _)     = p1
anchorPnt (Triangle p1 _ _)   = p1
anchorPnt (Quad     p1 _ _ _) = p1
```

- Pattern-matching can be used in list comprehensions (*later*)

```
(Line p1 p2) <- shapes
```

Pattern-matching *Type safety*

- Given a "Line" object, it is impossible to read "the field corresponding to the third point in a Triangle object“ because:
  - all unions are *tagged* unions
  - fields of an algebraic type can only be examined *via* pattern-matching

Special syntax

- Function type constructor

  `Int -> Bool`

  Conceptually:

  `Function Int Bool`

  i.e., the arrow is an "infix" type constructor

- Tuple type constructor

  `(Int, Bool)`

  Conceptually:

  `Tuple2 Int Bool`

  Similarly for Tuple3, ...

Type Synonyms

```
data Point = Point Int Int
```

a new data type

versus

```
type Point = (Int,Int)
```

a type synonym

Type Synonyms do not create new types. It is just a convenience to improve readability.

```
move :: Point -> (Int,Int) -> Point
move (Point x y) (sx,sy) =
              Point (x + sx) (y + sy)
```

versus

```
move (x,y) (sx,sy) = (x + sx, y + sy)
```


Abstract Types

A rational number is a pair of integers but suppose we want to express it in the reduced form only. Such a restriction cannot be enforced using an algebraic type.

```
module Rationalpackage
      (Rational,rational,rationalParts) where
   data Rational = RatCons Int Int

   rational :: Int -> Int -> Rational
   rational x y = let
                      d = gcd x y
                   in RatCons (x/d) (y/d)

   rationalParts :: Rational -> (Int,Int)
   rationalParts (RatCons x y)= (x,y)
```

No pattern matching on abstract data types

List: A Recursive Data Type

```
data  List t = Nil | Cons t (List t)
```

A list data type can be constructed in two different ways:

an empty list — `Nil`
or a non-empty list — `Cons x xs`

`x`: the first element
`xs`: the rest of the elements

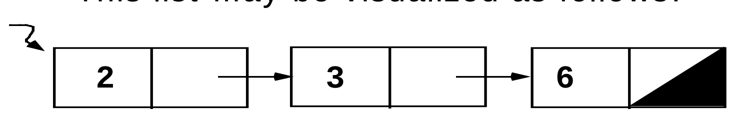
- All elements of a list have *the same type*
- The list type is *recursive* and *polymorphic*

Infix notation

```
Cons x  xs ≡ x:xs
```

```
2:3:6:Nil ≡ 2:(3:(6:Nil)) ≡ [2,3,6]
List Int ≡ [Int]
```

This list may be visualized as follows:



Example: Split a list

```
data Token = Word String | Number Int
```

Split a list of tokens into two lists - a list words and a list of numbers.

```
split :: [Token]-> ([String],[Int])

split []     = ([],[])
split (t:ts) =                        ?
```

```
let
      (ws,ns) = split ts
in
      case t of
            Word w       -> ((w:ws),ns)
            Number n     -> (ws,(n:ns))
```

Next time --- list comprehensions