Type Classes

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Slides by Arvind
September 20, 2011
Algebraic types are *tagged unions of products*

Example

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

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

```haskell
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

```haskell
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

```haskell
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")

- Clauses are examined top-to-bottom and left-to-right for pattern matching
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
Pattern-matching \textit{scope} & \textit{don't cares}

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

\begin{verbatim}
anchorPnt :: Shape -> Pnt
anchorPnt s = case s of
    Line    p1  _       -> p1
    Triangle p1  _  _    -> p1
    Quad     p1  _  _  _ -> p1
\end{verbatim}
Pattern-matching more syntax

- Functions can be defined directly using pattern-matching

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

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

```haskell
(Line p1 p2) <- shapes
```
Type Constructors: special syntax

- Function type constructor
  \[ \text{Int} \to \text{Bool} \]
  Conceptually:
  \[ \text{Function} \ \text{Int} \ \text{Bool} \]
  i.e., the arrow is an "infix" type constructor

- Tuple type constructor
  \[ (\text{Int}, \text{Bool}) \]
  Conceptually:
  \[ \text{Tuple2} \ \text{Int} \ \text{Bool} \]
  Similarly for Tuple3, ...
Type Synonyms

\textit{data} \textit{Point} = \textit{Point} \textit{Int} \textit{Int} \quad \text{a new data type}

\text{versus}

\textit{type} \textit{Point} = (\textit{Int},\textit{Int}) \quad \text{a type synonym}

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

\text{move} :: \textit{Point} \rightarrow (\textit{Int},\textit{Int}) \rightarrow \textit{Point}
\text{move (Point} \textit{x y) (sx, sy) =}
\quad \textit{Point} (\textit{x} + \textit{sx}) (\textit{y} + \textit{sy})

\text{versus}

\text{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.

```haskell
module Rational
package (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

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

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

- an empty list
- or a non-empty list

The first element

the rest of

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

Cons $x$ $xs \equiv x:xs$

$2:3:6:Nil \equiv 2:(3:(6:Nil)) \equiv [2,3,6]$

This list may be visualized as follows:
Example: Split a list

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

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

split :: (List Token) -> ((List String), (List Int))
split [] = ([], [])
split (t:ts) = let
                        (ws, ns) = split ts
                        in
                        case t of
                            Word w -> ((w:ws), ns)
                            Number n -> (ws, (n:ns))
```

Recursive functions which return data structures whose “size” is not known in advance would be impossible to write without recursive types
Overloading and Type Classes
Overloading *ad hoc polymorphism*

A symbol can represent multiple values each with a different type. For example, `+` represents:

```hs
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
\[ \forall t. (t \to t) \to (t \to t) \]

Any type can be substituted for \( t \) to get an instance of \texttt{twice}:

\[
\begin{align*}
\text{(Int -> Int)} & \to \text{(Int -> Int)} \\
\text{(String -> String)} & \to \text{(String -> String)}
\end{align*}
\]

Overloaded \( + \) does not have a most general type:

\[
\begin{align*}
\text{plusInt} & :: \text{Int -> Int -> Int} \\
\text{plusFloat} & :: \text{Float -> Float -> Float}
\end{align*}
\]

Can \( + \) be assigned the type \( \forall t. t \to t \to t \)?

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.
- Trickier 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

\( (\text{Num} \ t) \) is read as a predicate

“\( t \) is an instance of class \textbf{Num}”

\[
\begin{align*}
\text{sqr} & \quad :: \quad (\text{Num} \ a) \Rightarrow a \rightarrow a \\
\text{sqr} \ x &= x \ast x
\end{align*}
\]

What about constants?

\[
\text{plus1} \ x = x + 1
\]

If 1 is treated as an integer then \texttt{plus1} cannot be overloaded.

In Haskell numeric literals are overloaded and considered a short hand for

\[
(\text{fromInteger} \ \texttt{the_integer_1_value})
\]

where

\[
\text{fromInteger} \quad :: \quad (\text{Num} \ a) \Rightarrow \text{Integer} \rightarrow a
\]
The Equality Operator

- Equality is an overloaded function, not a polymorphic one

```haskell
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:

\[
\begin{align*}
\text{read} &: \text{String} \rightarrow \text{Int} \quad \text{- principal, duration} \\
\text{read} &: \text{String} \rightarrow \text{Float} \quad \text{- rate} \\
\text{show} &: \text{Float} \rightarrow \text{String} \quad \text{- monthly payments}
\end{align*}
\]

What is the type of \textit{read} and \textit{show}?

\[
\begin{align*}
\text{read} &: \text{String} \rightarrow a \\
\text{show} &: a \rightarrow \text{String}
\end{align*}
\]

Polymorphic?
Overloaded Read and Show

Haskell has a type class `Read` of “readable” types and a type class `Show` of “showable” types

```haskell
read :: Read a => String -> a
show :: Show a => a -> String
```
Ambiguous Overloading

\[
\begin{align*}
\text{identity} & \quad :: \quad \text{String} \to \text{String} \\
\text{identity} \ x & \quad = \quad \text{show} \ (\text{read} \ x)
\end{align*}
\]

What is the type of \((\text{read} \ x)\)?

Cannot be resolved! Many different types would do.

Compiler requires type declarations in such cases:

\[
\begin{align*}
\text{identity} & \quad :: \quad \text{String} \to \text{String} \\
\text{identity} \ x & \quad = \quad \text{show} \ ((\text{read} \ x) :: \text{Int})
\end{align*}
\]
Implementation

How does `sqr` find the correct function for `*`?

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

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

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

Then `(sqr 23)` will be compiled as

```haskell
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)