CS 115
Functional Programming

Lecture 6: April 13, 2016

Algebraic Datatypes

Today

- Defining new datatypes in Haskell
- Enumeration-style datatypes
- Datatypes with arguments
- Constructor functions
- Record syntax
- Recursive and polymorphic datatypes
- Type synonyms
- `newtype`

`Bool`

- Consider the humble boolean type `Bool`
- It has two possible values: `True` and `False`
- The datatype `Bool` is thus a finite enumeration of these two values
- In fact, the `Bool` type is not "hard-wired" into Haskell; it is defined in the Haskell libraries as a new type definition

`data`

- New datatypes are defined in Haskell with the `data` declaration
- Definition of `Bool`:

```
data Bool = True | False
```

- The vertical bar `|` is used to separate alternative values of the datatype
- The name of the datatype (`Bool`) must begin with a capital letter

`data`

```
data Bool = True | False
```

- The two values `True` and `False` are *constructors* of the data
- Constructors also have to have names beginning with a capital letter
- Now `Bool` is a type just like `Integer` or `Char`, and `True` and `False` are values just like `0` or `'a'`
- `True` and `False` are the *only* valid `Bool` values

ghci

- Let's define `Bool` ourselves:

```
Prelude> import Prelude hiding (Bool(..))
Prelude> True
[error message]
Prelude> data Bool = True | False deriving
(Show)
Prelude> True
True
```

ghci

```
Prelude> :info Bool
data Bool = True | False
instance Show Bool
Prelude> :type True
True :: Bool
Prelude> :type False
False :: Bool
```

(explained in later lecture)

Pattern matching

- Defining a new datatype with a data declaration not only defines a new type and new values of that type
- Also allows you to pattern-match against those values
- Example: `not` function:

```
not :: Bool -> Bool
not True  = False
not False = True
```

- `True`/`False` on LHS are patterns (trivial patterns)

Aside: strictness

- A Haskell function can be *lazy* or *strict* in any of its arguments
- If it's *strict* in an argument, then evaluating a non-terminating expression in that argument position will force the entire function call to not terminate
- We represent non-termination by the `_|_` (bottom) value (not Haskell syntax)
- So a strict argument in function `f` has `f _|_ = _|_`

Example: || function

- Logical *or* function uses the `||` operator
- One definition:

```
(||) :: Bool -> Bool -> Bool
True || True = True
True || False = True
False || True = True
False || False = False
```

- Is this definition lazy or strict in either argument?

Example: || function

```
(||) :: Bool -> Bool -> Bool
True || True = True
True || False = True
False || True = True
False || False = False
```

- To pattern-match a boolean expression against either **`True`** or **`False`**, it must be evaluated completely
- Therefore, this definition is strict in *both* arguments

Example: || function

- Alternative definition of `||`:

```
(||) :: Bool -> Bool -> Bool
True || _ = True
False || x = x
```

- Is this definition lazy or strict in either argument?
- Clearly strict in first argument (must evaluate to `True` or `False` to do pattern matching)
- Lazy in second argument (1st equation: don't even need to evaluate 2nd argument)

Example: || function

- Moral: Just because Haskell is a "lazy language" doesn't mean that *all* functions are lazy in *all* arguments
- Structure of function may dictate that certain arguments will always have to be evaluated all the way, others not

Sum types

- Many datatypes in Haskell are a series of alternatives
- Each constructor (like `True` or `False`) represents one possible kind of data value of that type
- These are sometimes called "sum types" (the type as a whole is the "sum" of several disjoint constructors)
- Sum types where the constructors have no arguments are simply enumerations

More `data` examples

- Easy to define enumeration-style datatypes:

```
data Color  = Red | Green | Blue | Yellow
data Day    = Mon | Tue | Wed | Thurs | Fri | Sat | Sun
data Beatle = John | Paul | George | Ringo
data Major  = CS | Whatever
```

etc.

- All such definitions also define pattern-matching on the corresponding datatypes

More `data` examples

- Example with `Color`:

```
data Color  = Red | Green | Blue | Yellow

opposite :: Color -> Color
opposite Red    = Green
opposite Green  = Red
opposite Blue   = Yellow
opposite Yellow = Blue
```

Beyond enumerations

- Enumeration types are trivial, but still more pleasant and well-typed than *e.g.* C-style `enum` values (aliases for integers)
- However, `data` declarations can do much more:
  - constructors can have one or more arguments (even of same type as the type being defined!)
  - different constructors don't have to have the same number of arguments

Product types

- Some datatypes consist of a single constructor which has one or more arguments
- This corresponds to what is usually called a "record" or "struct" in other languages
  - also similar to a Haskell tuple, but with a label
- Technically, these are called "product types" because the type is the Cartesian product of the types of the arguments to the constructor (where types conceptually represent sets of values)

Product types

- Examples:

```
data Point  = Pt Double Double    -- x and y coords
data Person = Per String String   -- first and last names
data Course = C String Int        -- field, number
```

- Some programmers reuse the type name as the constructor name (legal in Haskell):

```
data Point  = Point Double Double
data Person = Person String String
data Course = Course String Int
```

Product types

- Again, product types define pattern-matching over their constructors:

```
distance :: Point -> Point -> Double
distance (Pt x1 y1) (Pt x2 y2) =
  sqrt ((x1 - x2)^2 + (y1 - y2)^2)

pointX :: Point -> Double
pointX (Pt x _) = x

pointY :: Point -> Double
pointY (Pt _ y) = y
```

Record syntax

- Simple product types are common, but having to explicitly define accessors is a pain:

```
pointX :: Point -> Double
pointX (Pt x _) = x

pointY :: Point -> Double
pointY (Pt _ y) = y
```

Record syntax

- Haskell provides a shortcut where both the datatype and the accessors can be defined at the same time:

```
data Point = Pt { pointX :: Double, pointY :: Double }

Prelude> :t pointX
Point -> Double
Prelude> :t pointY
Point -> Double
```

Record syntax

- Can pattern match using record syntax too:

```
distance :: Point -> Point -> Double
distance (Pt {pointX=x1, pointY=y1})
         (Pt {pointX=x2, pointY=y2}) =
  sqrt ((x1 - x2)^2 + (y1 - y2)^2)
```

- In pattern, can put fields of constructor in any order if name labels are included

Record syntax

- You might wish that we could do this:

```
distance :: Point -> Point -> Double
distance p1 p2 =
  sqrt ((p1.x - p2.x)^2 + (p1.y - p2.y)^2)
```

- Alas, this is not legal Haskell syntax
- One of the most asked-for syntax extensions
- One proposal called *"Type Directed Name Resolution"* (see Haskell web pages)

Record syntax

- Similarly, can't have two different data definitions which use same field names:

```
data Point2 = P2 { x :: Double, y :: Double }
data Point3 = P3 { x :: Double,  -- illegal
                   y :: Double,  -- illegal
                   z :: Double }
```

- Records are thus somewhat clumsy to use in Haskell

So far

- We've seen
  - simple sum types (enumerations)
  - simple product types (records)
- More generally, many types have both sum and product components
  - different constructors, each with different number of arguments
- We refer to these as "algebraic datatypes"

Natural numbers

- Simple example: natural numbers
- A natural number is either
  - zero
  - the successor of a natural number
- Write this in Haskell as:

```
data Nat =
    Zero
  | Succ Nat
```

Natural numbers

```
data Nat =
     Zero
   | Succ Nat
```

- This defines two constructors: `Zero` and `Succ`
- `Zero` is a value
- `Succ` is a "constructor function" with type `Nat -> Nat`
- Constructors like `Succ` that have type arguments can be used as regular functions (though they have capitalized names)

Natural numbers

```haskell
data Nat =
      Zero
    | Succ Nat
```

- Note that this type is "recursive"
  - Defining `Nat`, but one of the constructors assumes that `Nat` has been defined
  - Haskell has no problem with this

Natural numbers

```
Prelude> :t Zero
Zero :: Nat
Prelude> :t Succ
Succ :: Nat -> Nat
Prelude> :t Succ Zero
Succ Zero :: Nat
```

Natural numbers

- **Nat** definition also defines pattern-matching on **Nat**s:

```
addNat :: Nat -> Nat -> Nat
addNat Zero n = n
addNat (Succ m) n = Succ (addNat m n)

mulNat :: Nat -> Nat -> Nat
mulNat Zero _ = Zero
mulNat (Succ m) n = addNat n (mulNat m n)
```

Natural numbers

- More functions on **`Nat`**s

```
natToInteger :: Nat -> Integer
natToInteger Zero = 0
natToInteger (Succ n) = 1 + natToInteger n
```

- Note: Structure of a **`Nat`**:

```
Succ (Succ (Succ (Succ Zero)))
```

- Want to convert to:

```
1 + (1 + (1 + (1 + 0)))
```

- What would be a more elegant way to define **`natToInteger?`**

Natural numbers

`Succ (Succ (Succ (Succ Zero)))`

- Want to convert to:

`1 + (1 + (1 + (1 + 0)))`

- Seems like we should be able to do something like `foldr` here…
- `foldr` works only on lists
- Let's define `foldn` to work on `Nat`s
- It will specify:
  - a special value to be used in place of `Zero`
  - a special unary function to be used in place of `Succ`

Natural numbers

```
foldn :: (a -> a) -> a -> Nat -> a
foldn _ init Zero = init
foldn f init (Succ n) = f (foldn f init n)
```

- Now we can define:

```
natToInteger :: Nat -> Integer
natToInteger = foldn (1+) 0
```

Polymorphic datatypes

- Algebraic datatypes can also depend on type variables
  - like polymorphic functions
- Recall the built-in list type
  - not specified to any particular list element type
- Want to be able to do this with user-defined types too
- Let's re-create the list type at the user level

Polymorphic datatypes

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

- This defines a family of types called **`List a`**
- "List of elements of some particular type **`a`**"
- Isomorphic to normal Haskell list type, which could be written as:

```
data [a] =
    []
  | a : [a]
```

Polymorphic datatypes

- Could define `foldr` version to work on this `List` type:

```
foldr2 :: (a -> b -> b) -> b -> List a -> b
foldr2 _ init Nil = init
foldr2 f init (Cons h t) = f h (foldr2 f init t)
```

- Moral: built-in list type is not special, except for syntax

Kinds

- Note that `List` is not a type
  - `List Integer` is a type
  - `List Float` is a type
  - `List Char` is a type
  - but `List` by itself is not a type!
- `List` is a "type constructor"
  - like a "function on types"
  - Give it a type (like `Integer`) and it will return a type (`List Integer`)

Kinds

- Type constructors do not have "types", they have "kinds"
- A "kind" is a "type of types"
- Simple types (non-type constructors) have the kind `*`
- Type constructors have the kind (`* -> *`), (`* -> * -> *`) *etc.* depending on how many type variables they have
- `ghci` will tell you what the kind of a type constructor is

Kinds

```
Prelude> :info List
data List a = Nil | Cons a (List a)
Prelude> :kind List
List :: * -> *
Prelude> :kind List Integer
List Integer :: *
Prelude> :kind Integer
Integer :: *
```

- Can abbreviate `:kind` as just `:k`

Maybe

- Another useful polymorphic type constructor is **`Maybe`**
- Used to represent values that "may or may not exist"

```
data Maybe a =
    Nothing
  | Just a
```

- Mostly useful as a function return argument
- Functions of type **`a -> Maybe b`** represent computations that may fail
- **`Maybe`** is also a monad (as we'll see later)

Maybe

- Let's ask **ghci** about **Maybe**

```
Prelude> :i Maybe
data Maybe = Nothing | Just a
Prelude> :t Nothing
Nothing :: Maybe a
Prelude> :k Maybe
Maybe :: * -> *
```

`Either`

- Polymorphic datatypes can depend on more than one type variable
- Simplest example: `Either` type constructor

```
data Either a b =
    Left a
  | Right b
```

- `Either` allows us to define a type which can be either of two arbitrary types
- For instance, `Either Int String` is either an `Int` or a `String`

Either

- Again, `Either` is often used as a return type from a function
- Functions with the type `a -> Either String b` can represent functions that either
  - succeed with a value of type `b`
  - fail with an error message (`String` value)
- This also constitutes a monad (as we'll see later)

```
Prelude> :k Either
Either :: * -> * -> *
```

Trees

- Polymorphic datatypes very often used to represent generic data structures
- For instance, binary trees of some type `a`

```
data Tree a =
    Leaf
  | Node a (Tree a) (Tree a)
```

- For now, we won't worry about balancing or ordering

Trees

- Function to collect **`Tree`** values into a list in order:

```
treeToList :: Tree a -> [a]
treeToList Leaf = []
treeToList (Node x left right) =
  treeToList left ++ [x] ++ treeToList right
```

- Again, note pattern matching on **`Tree`** constructors

Type aliases

- `data` declarations are the normal way to create new Haskell datatypes
- Sometimes, we have an existing type with an unpleasant name (too long, not descriptive enough) but don't want/need to define a completely new type
- We can define a type alias using the `type` keyword:

```
type String = [Char]
```

- This defines `String` as another name for `[Char]`

Type aliases

- Type aliases mean that **ghci** can't always choose the name for a type you might prefer:

```
Prelude> :t "foobar"
"foobar" :: [Char]    -- not String
```

- Type aliases also may mean that error messages involving types may refer to the aliased name or the unaliased name
- Therefore, type aliases are mainly a convenience for the person writing the code

`newtype`

- Type aliases only provide a new name for an old type, not a new type
- Sometimes, we want to define a new type for a previously-existing type in such a way that the new type is identifiably distinct from values of the old type but the contents are the same
- Can do this with a `data` declaration e.g.:

```
data Label = Lbl String
```

- Now Haskell considers `Label` and `String` to be distinct types

`newtype`

```
data Label = Lbl String
```

- Difference between a value of type `Label` and a value of type `String`:
  - `Label` value is "wrapped" in the constructor `Lbl`, `String` value is not
  - This "wrapping" means that `Label` values take up more space than `String`s, and must be unwrapped to get the contents (space and time costs)
  - What if we wanted to keep the `Label` and `String` types distinct, but not pay this cost?

`newtype`

- Define **`Label`** as a **`newtype:`**

```
newtype Label = Lbl String
```

- Differences between **`data`** and **`newtype`**:
  - **`newtype`** only allowed for datatypes with *one* constructor which has exactly *one* type argument
  - **`newtype`** defined datatypes are just as efficient as the type they wrap (no wrapping/unwrapping penalty), but are distinct to the type checker
  - (and a few other subtle issues you're unlikely to run into for a long time)

Next time

- Type classes!