CS 320: Concepts of Programming Languages

Wayne Snyder
Computer Science Department
Boston University

Lecture 03: Bare-Bones Haskell Continued:

- Function Application = Rewriting by Pattern Matching
- Haskell Types and Polymorphism

Function Application by Matching and Rewriting

Recall: Rewriting involves matching the left-hand side of a function definition with a subexpression, where variables are instantiated to subexpressions. Function definitions are tried in order from the top.

```
data Bool = False | True

data Nat = Zero | Succ Nat deriving Show

not True = False
not False = True

pred Zero     = Zero
pred (Succ x) = x

cond True x y  = x
cond False x y = y
```

Data (constructors) in green, Variables (including function names) in black.

I'll use => to indicate "rewrites to" and the "redex" = term being rewritten, will be underlined.

```
   cond (not True) (Succ Zero) (pred (Succ Zero))
=> cond False (Succ Zero) (pred (Succ Zero))
=> (pred (Succ Zero))
=> Zero
```

Function Application by Matching and Rewriting

Three important things to remember about defining functions by pattern matching:

(1) The left-side of a function definition must consist of a function name followed by expressions consisting only of constructors and variables, and variables can occur at most once:

```
data Bool = False | True

data Nat = Zero | Succ Nat deriving Show

not True = False
not False = True

pred Zero     = Zero
pred (Succ x) = x

cond True x y  = x
cond False x y = y
```

Not allowed:

```
cond  True x y       = x
cond  (not True) x y = y

xor x x = False
xor x y = True
```

Function Application by Matching and Rewriting

Three important details on matching in Haskell:

(1) Continued...

Note that constructor expressions can be as complicated as you want!

```
data Nat = Zero | Succ Nat deriving Show

data Expr = Val Nat
          | Plus Expr Expr
          | Times Expr Expr deriving Show

rightAssoc (Plus (Plus x y) z) = Plus x (Plus y z)
```

rightAssoc (Plus (Plus (Val Zero) (Val Zero)) (Val Zero))

=> Plus (Val Zero) (Plus (Val Zero) (Val Zero))

Functional Application by Matching and Rewriting

Three important details on matching in Haskell:

(2) The patterns (LHSs) have to account for all possible expressions, that is, the range of the patterns has to be exhaustive. Haskell can check this for you!

```
incr Zero   =   (Succ Zero)
```

What about (Succ Zero)??

Better:

```
incr Zero       = (Succ Zero)
incr (Succ x) = (Succ (Succ x) )
```

Best:

```
incr x       = (Succ x)
```

Functional Application by Matching and Rewriting

Three important details on matching in Haskell:

(3) You can use "wildcard" variables, that match anything and don't create a binding:

```
isZero Zero = True
incr _      = False
```

If you put such a rule LAST, it can account for anything other expressions have not matched yet.

Haskell Type System

Reading: Hutton Ch. 3

Type declarations are given by the syntax:

```
expression :: type-name
```

Examples:

```
 False :: Bool

(not (not False)) :: Bool
```

Function types have the form:

```
argument-type -> result-type
```

Example:

```
not :: Bool -> Bool
```

Haskell Type System

Reading: Hutton Ch. 3

You can find the type of an expression in the repl using `:type` or `:t`

```
data Bool = False | True
data Nat = Zero | Succ Nat deriving Show

not True  = False
not False = True

isZero Zero  = True
isZero _     = False

odd Zero             = False
odd (Succ Zero)      = True
odd (Succ(Succ x)) = odd x
```

```
Main> :type (not (not True))
(not (not True)) :: Bool

Main> :t (isZero Zero)
(isZero Zero) :: Bool

Main> :t not
not :: Bool -> Bool

Main> :t isZero
isZero :: Nat -> Bool

Main> :t odd
odd :: Nat -> Bool
```

Haskell Type System

Reading: Hutton Ch. 3

You should specify a type as part of the definition of a function:

```
data Bool = False | True
data Nat = Zero | Succ Nat deriving Show

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

isZero :: Nat -> Bool
isZero Zero   = True
isZero _      = False

odd :: Nat -> Bool
odd Zero            = False
odd (Succ Zero)     = True
odd (Succ(Succ x)) = odd x
```

In general, this is good practice, and expected as part of good Haskell programming style. It provides **documentation** about how the function works and in some cases, is necessary to be specific about what you want the function to do.

Haskell Type System

Reading: Hutton Ch. 3

If you don't specify a type, Haskell can infer the types from the expressions:

```
data Bool = True | False

data Nat = Zero | Succ Nat

even Zero        = True
even (Succ x)    = odd x
```

Must be Nat -> Bool!

Haskell uses the following rule to infer the types of expressions:

$$\frac{\texttt{f :: A -> B} \qquad \texttt{e :: A}}{\texttt{(f e) :: B}}$$

premises

conclusion

Therefore, `(even Zero)` must have the type `Bool`:

$$\frac{\texttt{even :: Nat -> Bool} \qquad \texttt{Zero :: Nat}}{\texttt{(even Zero) :: Bool}}$$

Haskell Type System

Reading: Hutton Ch. 3

The type system also applies to the data types, and constructors have types just like function types, except the constructors don't do anything except structure the data.

```
data Bool = False | True
data Nat = Zero | Succ Nat deriving Show

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

isZero :: Nat -> Bool
isZero Zero  = True
isZero _     = False

odd :: Nat -> Bool
odd Zero              = False
odd (Succ Zero)       = True
odd (Succ(Succ x))    = odd x

data Expr = Val Nat
          | Plus Expr Expr
          | Times Expr Expr deriving Show

rightAssoc (Plus (Plus x y) z) = Plus x (Plus y z)
```

```
Main> :t Succ
Succ :: Nat -> Nat

Main> :t Val
Succ :: Nat -> Expr

Main> :t Plus
Plus :: Expr -> Expr -> Expr
```

Haskell Type System

Reading: Hutton Ch. 3

Functions and constructors of more than one argument have types with multiple "arrows"; the last type is the result type and the others are the argument types:

```
data Bool = False | True deriving Show

data Nat = Zero | Succ Nat deriving Show

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

isZero :: Nat -> Bool
isZero Zero  = True
isZero _     = False

odd :: Nat -> Bool
odd Zero              = False
odd (Succ Zero)       = True
odd (Succ(Succ x))    = odd x

cond :: Bool -> Nat -> Nat -> Nat
cond True x y  = x
cond False x y = y
```

```
Main> :t cond
cond :: Bool -> Nat -> Nat -> Nat
```

Polymorphic Types

Reading: Hutton Ch. 3.7

Recall: Many functions (and data types) do not need to know everything about the types of the arguments and results.

Let's start with data types. Why should we have to define a list type for every possible kind of data in the list?

```
data ListBool = NilBool | ConsBool Bool ListBool

data ListNat = NilNat | ConsNat Nat ListNat
```

Instead, we can define polymorphic types using type variables:

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

(List Nat) is isomorphic to ListNat

a is a type variable, and just like any other variable, it can stand for anything (in this case, any type).

Compare Java Generics:

```
class List< T > {

        T element;
        .....
}
```

Polymorphic Types

Reading: Hutton Ch. 3.7

```
data Bool = False | True deriving Show

data Nat = Zero | Succ Nat deriving Show

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

```
Main> :t (ConsNat Zero NilNat)
(ConsNat Zero NilNat) :: ListNat

Main> :t (Cons Zero Nil)
(Cons Zero Nil) :: List Nat

Main> :t (Cons True (Cons False Nil))
(Cons True (Cons False Nil)) :: List Bool

Main> :t (Cons (Cons True Nil) Nil)
```

What's the type?

Polymorphic Types

Reading: Hutton Ch. 3.7

```
data Bool = False | True deriving Show

data Nat = Zero | Succ Nat deriving Show

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

```
Main> :t (Cons (Cons True Nil) Nil)
(Cons (Cons True Nil) Nil) :: List (List Bool)
```

Haskell can also infer polymorphic types:

```
Main> :t Nil
Nil :: List a

Main> :t Cons
Cons :: a -> List a -> List a
```

```
Main> identity x = x

Main> :t identity
identity :: a -> a

Main> test x y = x

Main> :t test
test :: a -> b -> a
```

Polymorphic Types

Reading: Hutton Ch. 3.7

Functions also can have polymorphic types when they don't need to know exactly what type of data they manipulate.

Most of these functions involve restructuring or selecting out pieces of data, for example in lists:

```
data Bool = False | True deri

data Nat = Zero | Succ Nat de

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


head :: List a -> a
head (Cons x _) = x

tail :: List a -> List a
tail (Cons _ xs) = xs

second (Cons _ (Cons x _)) = x
```

```
Main> :t second
second :: List a -> a

Main> a = Cons True (Cons False Nil)

Main> :t a
a :: List Bool

Main> head a
True

Main> tail a
Cons False Nil
```