CS 320: Concepts of Programming Languages

Wayne Snyder
Computer Science Department
Boston University

Lecture 04: Basic Haskell Continued

- Polymorphic Types
- Type Inference with Polymorphism
- Standard Types: Bools, Integers
- Function definitions in more detail:
  if-then-else, guards, where

Reading: Hutton Chapter 3, 4.1 – 4.4

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.

Many data types and most list-processing functions are of this kind:

```
data List a = Nil | Cons a (List a)
data Pair a b = P a b
data Triple a b c = T a b c

append :: List a -> List a -> List a

reverse :: List a -> List a

head (Cons x _ ) = x

tail (Cons x xs) = xs
```

Check: What is the type of

```
P ?

head ?

tail ?
```

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.

Many data types and most list-processing functions are of this kind:

```
data List a = Nil | Cons a (List a)
data Pair a b = P a b
data Triple a b c = T a b c

append :: List a -> List a -> List a

reverse :: List a -> List a

head (Cons x _ ) = x

tail (Cons x xs) = xs
```

Check: What is the type of

```
P :: a -> b -> Pair a b

head :: List a -> a

tail :: List a -> List a
```

Polymorphic Types

Reading: Hutton Ch. 3.7

Polymorphic type inference, based on unifying type expressions, determines the types of all expressions by looking at all the places where types must be the same:

```
pred :: Nat -> Nat

-- return the first n elements of a list

take    Zero   xs            = xs
take    n      Nil           = Nil
take    n      (Cons x xs)   = Cons x (take (pred n) xs)
```

- Same variable in a rule must be same type.
- Arguments each each position and result types must be the same.
- Inputs to function and type of arguments must be same.

Type of function must be:

```
take :: Nat  -> List a      ->    List a
```

Polymorphic Types

Reading: Hutton Ch. 3.7

Polymorphic type inference, based on unifying type expressions, determines the types of all expressions by looking at all the places where types must be the same:

```
pred :: Nat -> Nat

-- return the first n elements of a list

take :: Nat -> List a      -> List a
take    Zero   xs          = xs
take    n      Nil         = Nil
take    n     (Cons x xs)  = Cons x (take (pred n) xs)
```

But ALL expressions must have appropriate types, using rule:

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

Polymorphic Types

Reading: Hutton Ch. 3.7

Polymorphic type inference determines the most general type that a function can have. This involves accounting for all the type constraints implied when you examine two type expressions that must apply to a single context (say an argument to a function):

```
data Bool = False | True
data Nat = Zero | Succ Nat
data Triple a b c = T a b c          -- ex: (Zero,Zero,True)

T :: a -> b -> c -> Triple a b c
```

Example 1:

Let `s = (T Zero x     y)`   `-- x,y,z,w` can have any types
Let `t = (T z    False w)`

If `s` and `t` have to have the same type, what would that type be?

```
(Triple Nat Bool a)
```

and furthermore, we must have `z :: Nat`, `x :: Bool` but `y,w` can be anything as long as they are the same type `a` !

Polymorphic Types

Reading: Hutton Ch. 3.7

```
data Bool = False | True
data Nat = Zero | Succ Nat
data Triple a b c = T a b c        -- ex: (A,C,B)
```

Example 2:

Let `s = (T True x     False)`   `--` `x,y,z,w` have unknown types
Let `t = (T z    False x     )`

If `s` and `t` have to have the same type, what would that type be?

For s we have `(Triple Bool a    Bool)`
For t we have `(Triple b    Bool c    )`

For the types to be the same we would have to have `a = b = c = Bool:`

```
(Triple Bool Bool Bool)
```

This process is like "two-sided matching" and is called **Unification**:

`(Triple Bool b Bool))` ⟶ `(Triple Bool Bool Bool)` ⟵ `(Triple a Bool c)`

Polymorphic Types

Reading: Hutton Ch. 3.7

```
data Bool = False | True
data Nat = Zero | Succ Nat
data Pair a b = P a b                -- ex: (A,B)
data Triple a b c = T a b c          -- ex: (A,C,B)
```

Example 3:

```
Let s = (T True x     x    )    -- x,y can have any types
Let t = (T y    False Zero )
```

If `s` and `t` have to have the same type, what would the type of `T` be?

Answer: No type exists, as x would have to simultaneously be Bool and Nat, so it is contradictory and is a type error! The type expressions

`(Triple Bool a a)` and `(Triple b Bool Nat)`

can NOT be unified!

Polymorphic Types

Reading: Hutton Ch. 3.7

```
data Bool = False | True
data Nat = Zero | Succ Nat
data Pair a b = P a b
data Triple a b c = T a b c
```

Example:

```
f :: (Pair a b) -> (Triple a b b)
f    (P x y) =  (T x y y)

k :: (Pair Bool a) -> (Pair a Bool)
k    (P x    y)  = (P y x   )

test x = (f (k x))
```

unify these two

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

$$\frac{\texttt{k :: (Pair Bool a) -> (Pair a Bool)} \qquad \texttt{x :: c}}{\texttt{(k x) :: (Pair a Bool)}}$$

```
c = (Pair Bool a)
```

Polymorphic Types

Reading: Hutton Ch. 3.7

```
data Bool = False | True
data Nat = Zero | Succ Nat
data Pair a b = P a b
data Triple a b c = T a b c
```

Unification determines what type a function must have:

```
f :: (Pair a b) -> (Triple a b b)
f    (P    x y) =  (T        x y y)

k :: (Pair Bool a) -> (Pair a Bool)
k    (P    x    y) =  (P    y x    )

test x = (f (k x))
```

unify these two

```
f :: A -> B       e :: A'
-------------------------
      (f e) :: B'
```

```
         k :: (Pair Bool a) -> (Pair a Bool)      x :: c
         -------------------------------------------------
                                (k x) :: (Pair a Bool)
 f :: (Pair a' b') -> (Triple a' b' b')
-----------------------------------------------------------
                 (f (k x)) :: ??
```

Unify:

```
(Pair a' b'   )
(Pair a  Bool)
```

```
c = (Pair Bool a)     a = a'      b' = Bool
```

Note: names can include prime marks:

```
a  a'  a''
```

Polymorphic Types

Reading: Hutton Ch. 3.7

```
data Bool = False | True
data Nat = Zero | Succ Nat
data Pair a b = P a b
data Triple a b c = T a b c
```

Unification determines what type a function must have:

```
f :: (Pair a b) -> (Triple a b b)
f    (P    x y) =  (T      x y y)

k :: (Pair Bool a) -> (Pair a Bool)
k    (P    x    y)  = (P    y x    )

test x = (f (k x))
```

unify these two

```
f :: A -> B        e :: A'
---------------------------
      (f e) :: B'
```

```
         k :: (Pair Bool a) -> (Pair a Bool)        x :: c
         -------------------------------------------------
                               (k x) :: (Pair a Bool)
 f :: (Pair a' b') -> (Triple a' b' b')
 ---------------------------------------------------------
           (f (k x)) :: (Triple a' Bool Bool)
```

```
c = (Pair Bool a)      a = a'      b' = Bool
```

test :: ??

Polymorphic Types

Reading: Hutton Ch. 3.7

```
data Bool = False | True
data Nat = Zero | Succ Nat
data Pair a b = P a b
data Triple a b c = T a b c
```

Unification determines what type a function must have:

```
f :: (Pair a b) -> (Triple a b b)
f    (P     x y) =  (T       x y y)

k :: (Pair Bool a) -> (Pair a Bool)
k    (P     x    y)  = (P    y x   )

test x = (f (k x))
```

unify these two

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

$$\frac{\texttt{f :: (Pair a' b') -> (Triple a' b' b')} \qquad \dfrac{\texttt{k :: (Pair Bool a) -> (Pair a Bool)} \qquad \texttt{x :: c}}{\texttt{(k x) :: (Pair a Bool)}}}{\texttt{(f (k x)) :: (Triple a' Bool Bool)}}$$

```
c = (Pair Bool a)      a = a'      b' = Bool
```

```
test :: P Bool a' -> T a' Bool Bool
```

Another example at end of the slides.....

Adding Numbers to Bare Bones Haskell: Built-in Numeric Types

Int -- fixed-precision integers
Integer – arbitrary-precision integers
Float – 32-bit floating-point
Double – 64-bit float-point
Rational

Operators `+`, `-`, `*`, `==` are the same in Haskell as in Python, Java, &&C except:

exponentiation: `x^3` (only for integer exponents)
`x**3.1415` (only for floating-point exponents)
unary minus: `(-9)` (must use parentheses)
not equals: `/=`
Integer division: `(div 10 7) => 1`
Floating-point division `( 3.4 / 4.9) => 0.693877551020408`
modulus: `(mod 10 7)  => 3`

We'll explore types in detail next week..... for now we will only use Integers.

Built-in Numeric Types: Infix vs Prefix Functions

We have been using prefix notation up to this point and two of the new functions we have for Integers are given in this form:

Integer division: `(div 10 7) => 1`

modulus: `(mod 10 7) => 3`

But most (binary) arithmetic operators are infix:

```
(4 * 3)  =>  12

(2 - 3)  => (-1)
```

Remember:
=> means "evaluates to"

There are also postfix (unary) functions in mathematics:

```
5! => 120
```

as well as mixfix for functions of more than 2 arguments:

```
( 3 < 4 ? 2 : 5 ) => 2

(if 6 < 4 then 2 else 5) => 5
```

The term **operator** generally refers to a function which is used with infix notation: **+ * ^** etc.
We'll just call them **functions**.

Built-in Numeric Types: Infix vs Prefix Functions

Haskell is completely flexible about prefix and infix notation for binary (two argument) functions:

To use a function defined in **prefix form as infix** surround it by **backquotes**:

```
(div 10 7) => 1                  (10 `div` 7) => 1
(mod 10 7) => 3                  (10 `mod` 7) => 3
```

To use a function defined in **infix form as prefix** surround it by **parentheses**:

```
(10 + 7) => 17                   ((+) 10 7) => 17
(10 ^ 3) => 1000                 ((^) 10 3) => 1000
```

To define an **infix** function it must consist of special symbols (no letters) and the type declaration must use prefix (with parentheses):

```
(!!) :: List a -> Integer -> a    -- select the nth element
(!!) (Cons x _ ) 0 = x
(!!) (Cons x xs) n = xs !! (n-1)
```

Prelude's Boolean Type

So Haskell defines the Bool type in the Prelude as follows (Hutton p. 281):

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

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

```
otherwise :: Bool
otherwise = True
```

```
(&&) :: Bool -> Bool -> Bool
False && _ = False
True  && b = b
```

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

So you can just use the Bool defined in Prelude from now on...

Functions Definitions

Reading: Hutton Ch. 4

Now we will look at ways to extend BB Haskell to make it easier to use!

An important predefined function: Conditional expressions

Real Haskell	Bare-Bones Haskell
`(if x then y else z)`	`(cond x y z)`

Bool   Must be same type because expressions can only return one type:

```
(if 5 < 8  then  6  else  8) * 3       =>   18
```

So, the type is: **`Bool -> a -> a -> a`**

Functions Definitions: Where Expressions

It is very common to need "helper functions" to define a function:

```
remDup :: List Integer -> List Integer
remDup  Nil           = Nil
remDup (Cons x Nil) = (Cons x Nil)
remDup (Cons x xs)  = remDup' x (reDup xs)

remDup' :: List Integer -> List Integer
remDup' x xs = if x == head xs then xs else (Cons x xs)
```

But why should `remDup'` be visible anywhere but the definition of `remDup`?

What if you just want to call it `f` or `helper`? Then you can't use these names anywhere again in this file!

Would be nice to have a "local definition" of the helper functions....

Functions Definitions: Where Expressions

Is is a good idea to indent your helper functions using the keyword where:

```
remDup :: List Integer -> List Integer
remDup  Nil            = Nil
remDup (Cons x Nil) = (Cons x Nil)
remDup (Cons x xs)  = remDup' x (reDup xs)
   where remDup' :: List Integer -> List Integer
         remDup' x xs = if x == head xs then xs else (Cons x xs)
```

scope of where

```
len x y = sqrt (sq x + sq y)
      where sq a = a * a

doStuff :: Int -> String
doStuff x  | x < 3 = report "less than three"
           | otherwise = report "normal"
           where report y = "the input is " ++ y
```

Haskell Types

Reading: Hutton Ch. 4

Guarded Equations

Consider the following functions to find minimum and maximum of two Integers

```
min :: Integer -> Integer -> Integer
min x y = if x <= y then x else y

max :: Integer -> Integer -> Integer
max x y = if x >= y then x else y
```

This is a fairly common pattern, where we test some Boolean condition on the parameters. In Haskell, this can be equivalently done using "Guarded Matching": (Hutton p.280)

```
min :: Integer -> Integer -> Integer
min x y | x <= y   = x      match succeeds only if guard true
min x y            = y

max :: Integer -> Integer -> Integer
max x y | x >= y    = x          match succeeds only if guard true
        | otherwise = y          otherwise is always True
```

Haskell Types

Reading: Hutton Ch. 4

There are usually many different ways of defining a function, and no one way (helper functions, if-then-else, guards) is automatically better. These are available if you want to use them...

Using where:

```
remDup :: List Integer -> List Integer
remDup  Nil           = Nil
remDup (Cons x Nil) = (Cons x Nil)
remDup (Cons x xs)  = remDup’ x (reDup xs)
   where remDup’ :: List Integer -> List Integer
         remDup’ x xs = if x == head xs then xs else (Cons x xs)
```

Or you can do it with an if-then-else in the main function:

```
remDup :: List Integer -> List Integer
remDup  Nil                   = Nil
remDup (Cons x Nil)           = (Cons x Nil)
remDup (Cons x (Cons y ys)) = if (x == y)
                                 then (remDup (Cons y ys))
                                 else (Cons x (remDup (Cons y ys))
```

Haskell Types

Reading: Hutton Ch. 4

There are usually many different ways of defining a function, and no one way (helper functions, if-then-else, guards) is automatically better. These are available if you want to use them...

Or you can do it with an if-then-else in the main function:

```
remDup :: List Integer -> List Integer
remDup  Nil                  = Nil
remDup (Cons x Nil)          = (Cons x Nil)
remDup (Cons x (Cons y ys)) = if (x == y)
                                 then (remDup (Cons y ys))
                                 else (Cons x (remDup (Cons y ys))
```

Or you can do it with a guard:

```
remDup :: List Integer -> List Integer
remDup  Nil                          = Nil
remDup (Cons x Nil)                  = (Cons x Nil)
remDup (Cons x (Cons y ys)) | x == y = (remDup (Cons y ys))
remDup (Cons x (Cons y ys)) = (Cons x (remDup (Cons y ys)))
```

Polymorphic Types (Extra Practice!)

```
data Bool = False | True
data Nat = Zero | Succ Nat
data Pair a b = P a b
data Triple a b c = T a b c
```

Such a process determines what type a function must have:

```
g :: (Triple Bool a b) -> (Pair  (Pair Nat  a) b)
g    (T      True y z) =  (P     (P    Zero y) z)

h :: (Pair a (Pair b Nat )) -> (Triple a b Bool)
h    (P    x (P    y Zero)) =  (T      x y Bool)

comp x = (h (g x))
```

$$\frac{\texttt{g ::(Triple Bool a b) -> (Pair (Pair Nat a) b} \qquad \texttt{x::c}}{\texttt{(g x) :: (Pair (Pair Nat a) b)}}$$

`c =  (Triple Bool a b)`

unify these two

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

Polymorphic Types (Extra Practice!)

```
data Bool = False | True
data Nat = Zero | Succ Nat
data Pair a b = P a b
data Triple a b c = T a b c
```

Such a process determines what type a function must have:

```
g :: (Triple Bool a b) -> (Pair   (Pair Nat   a) b)
g    (T      True y z) =   (P     (P    Zero y) z)

h :: (Pair a (Pair b Nat )) -> (Triple a b Bool)
h    (P    x (P    y Zero)) =   (T      x y Bool)

comp x = (h (g x))
```

$$\dfrac{\dfrac{\texttt{g ::(Triple Bool a b) -> (Pair (Pair Nat a) b} \qquad \texttt{x::c}}{\texttt{(g x)::(Pair (Pair Nat a) b)}} \qquad \texttt{h ::(Pair a'(Pair b' Nat))->(Triple a' b' Bool)}}{\texttt{(h (g x)) :: ??}}$$

Unify:

```
(Pair  a'              (Pair b' Nat))        a' =  (Pair Nat a)
(Pair  (Pair Nat a) b              )         b  =  (Pair b' Nat)
```

Polymorphic Types (Extra Practice!)

```
data Bool = False | True
data Nat = Zero | Succ Nat
data Pair a b = P a b
data Triple a b c = T a b c
```

Such a process determines what type a function must have:

```
g :: (Triple Bool a b) -> (Pair  (Pair Nat  a) b)
g    (T      True y z) =  (P     (P    Zero y) z)
h :: (Pair a (Pair b Nat )) -> (Triple a b Bool)
h    (P    x (P    y Zero)) =  (T      x y Bool)

comp x = (h (g x))
```

```
                g ::(Triple Bool a b) -> (Pair (Pair Nat a) b      x::c
              ----------------------------------------------------------
                                              (g x)::(Pair (Pair Nat a) b)
 h ::(Pair a'(Pair b' Nat))->(Triple a' b' Bool)
-------------------------------------------------------------------------
              (h (g x)) :: (Triple  (Pair Nat a) b' Bool)
```

```
a' = (Pair Nat a)
b  = (Pair b' Nat)
c  = (Triple Bool a b)
```

comp :: ??

Polymorphic Types (Extra Practice!)

```
data Bool = False | True
data Nat = Zero | Succ Nat
data Pair a b = P a b
data Triple a b c = T a b c
```

Such a process determines what type a function must have:

```
g :: (Triple Bool a b) -> (Pair  (Pair Nat  a) b)
g    (T      True y z) =  (P     (P    Zero y) z)
h :: (Pair a (Pair b Nat )) -> (Triple a b Bool)
h    (P    x (P    y Zero)) =  (T      x y Bool)

comp x = (h (g x))
```

```
           g ::(Triple Bool a b) -> (Pair (Pair Nat a) b      x::c
           -----------------------------------------------------------
                                              (g x)::(Pair (Pair Nat a) b)
 h ::(Pair a'(Pair b' Nat))->(Triple a' b' Bool)
 ----------------------------------------------------------------------
             (h (g x)) :: (Triple  (Pair Nat a) b' Bool)
```

```
a' = (Pair Nat a)
b  = (Pair b' Nat)
c  = (Triple Bool a b)
```

`comp ::(Triple Bool a (Pair b' Nat)) -> (Triple (Pair Nat a) b' Bool)`