Haskell Boot Camp

CS4430 Spring 2017

Bill Harrison

February 1, 2017

Haskell Basics

- Modern (pure) lazy functional language
- Statically typed, supports type inference
- Compilers and interpreters:
  - http://www.haskell.org/implementations.html
  - Hugs interpreter
  - GHC Compiler Haskell Platform: `https://www.haskell.org/platform/`
- A peculiar language feature: indentation matters
- Also: capitalization matters

Reading

Read the following:

- Learn You a Haskell: Chapters 1 & 2
- Gentle Introduction to Haskell (http://www.haskell.org/tutorial/): Sections 1 & 2

Data + Algorithms = Programs

- Any program is a combination of data structures and code that manipulates that data

Data + Algorithms = Programs

- Any program is a combination of data structures and code that manipulates that data
- Ex: simple arithmetic interpreter
  - data structure:
    `data Exp = Const Int | Neg Exp | Add Exp Exp`
  - code:

```
interp :: Exp -> Int
interp (Const i)   = i
interp (Neg e)     = - (interp e)
interp (Add e1 e2) = interp e1 + interp e2
```

Data + Algorithms = Programs

- Any program is a combination of data structures and code that manipulates that data
- Ex: simple arithmetic interpreter
  - data structure:
    `data Exp = Const Int | Neg Exp | Add Exp Exp`
  - code:

```
interp :: Exp -> Int
interp (Const i)    = i
interp (Neg e)      = - (interp e)
interp (Add e1 e2) = interp e1 + interp e2
```

- Manipulation: How do Haskell programs use data?
  - Patterns break data apart to access:
    "`interp (Neg e) =...`"
  - Functions recombine into new data:
    "`interp e1 + interp e2`"

Type Declarations

In Haskell, a new name for an existing type can be defined using a type declaration.

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

`String` is a synonym for the type `[Char]`.

Type declarations can be used to make other types easier to read. For example, given

```
type Pos = (Int,Int)
```

we can define

```
origin     :: Pos
origin      = (0,0)

left       :: Pos -> Pos
left (x,y) = (x-1,y)
```

Like function definitions, type declarations can also have parameters. For example, given

```
type Pair a = (a,a)
```

we can define

```
mult        :: Pair Int -> Int
mult (m,n)  = m*n

copy        :: a -> Pair a
copy x       = (x,x)
```

Type declarations can be nested:

```
type Pos   = (Int,Int)      -- GOOD

type Trans = Pos -> Pos    -- GOOD
```

However, they cannot be recursive:

```
type Tree = (Int,[Tree])   -- BAD
```

A completely new type can be defined by specifying its values using a data declaration.

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

`Bool` is a new type, with two new values `False` and `True`.

Note:

- The two values False and True are called the constructors for the type Bool.
- Type and constructor names must begin with an upper-case letter.
- Data declarations are similar to context free grammars. The former specifies the values of a type, the latter the sentences of a language.

Values of new types can be used in the same ways as those of built in types. For example, given

```
data Answer = Yes | No | Unknown
```

we can define:

```
answers      :: [Answer]
answers       = [Yes,No,Unknown]

flip         :: Answer -> Answer
flip Yes      = No
flip No       = Yes
flip Unknown = Unknown
```

The constructors in a data declaration can also have parameters. For example, given

```
data Shape = Circle Float
           | Rect Float Float
```

we can define:

```
square         :: Float -> Shape
square n        = Rect n n
area           :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Note:

- Shape has values of the form Circle r where r is a float, and Rect x y where x and y are floats.
- Circle and Rect can be viewed as functions that construct values of type Shape:

```
-- Not a definition
Circle :: Float -> Shape
Rect   :: Float -> Float -> Shape
```

Not surprisingly, data declarations themselves can also have parameters. For example, given

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

we can define:

```
safediv    :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead   :: [a] -> Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

Recursive Types

In Haskell, new types can be declared in terms of themselves. That is, types can be recursive.

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

Nat is a new type, with constructors `Zero :: Nat` and `Succ :: Nat -> Nat`.

Note:

- A value of type `Nat` is either `Zero`, or of the form `Succ n` where `n` :: Nat. That is, `Nat` contains the following infinite sequence of values:

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

Note:

- We can think of values of type Nat as natural numbers, where `Zero` represents `0`, and `Succ` represents the successor function `1+`.
- For example, the value

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

represents the natural number

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

Using recursion, it is easy to define functions that convert between values of type Nat and Int:

```
nat2int          :: Nat -> Int
nat2int Zero      = 0
nat2int (Succ n) = 1 + nat2int n

int2nat          :: Int -> Nat
int2nat 0         = Zero
int2nat n         = Succ (int2nat (n - 1))
```

Two naturals can be added by converting them to integers, adding, and then converting back:

```
add     :: Nat -> Nat -> Nat
add m n = int2nat (nat2int m + nat2int n)
```

However, using recursion the function add can be defined without the need for conversions:

```
add Zero     n = n
add (Succ m) n = Succ (add m n)
```

The recursive definition for add corresponds to the laws

$$0 + n = n$$

and

$$(1 + m) + n = 1 + (m + n)$$

Using recursion, an expression tree can be defined using:

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

One example of such a tree written in Haskell is

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

Using recursion, it is now easy to define functions that process expressions. For example:

```
size              :: Expr -> Int
size (Val n)       = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval              :: Expr -> Int
eval (Val n)       = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Note:

- The three constructors have types:

```
-- Not a definition
Val :: Int -> Expr
Add :: Expr -> Expr -> Expr
Mul :: Expr -> Expr -> Expr
```

Using recursion, a binary tree can be defined using:

```
data Tree = Leaf Int
          | Node Tree Int Tree
```

One example of such a tree written in Haskell is

```
Node (Node (Leaf 1) 3 (Leaf 4))
     5
     (Node (Leaf 6) 7 (Leaf 9))
```

We can now define a function that decides if a given integer occurs in a binary tree:

```
occurs                   :: Int -> Tree -> Bool
occurs m (Leaf n)         = m==n
occurs m (Node l n r) = m==n
                           || occurs m l
                           || occurs m r

-- N.b., || is ``or''
```

In the worst case, when the integer does not occur, this function traverses the entire tree.

Search trees have the important property that when trying to find a value in a tree we can always decide which of the two sub-trees it may occur in:

```
occurs                    :: Int -> Tree -> Bool
occurs m (Leaf n)                 = m==n
occurs m (Node l n r) | m==n = True
                      | m<n  = occurs m l
                      | m>n  = occurs m r
```

This new definition is more efficient, because it only traverses one path down the tree.
What are we assuming at each `Node`?

Finally consider the function `flatten` that returns the list of all the integers contained in a tree:

```
flatten                :: Tree -> [Int]
flatten (Leaf n)        = [n]
flatten (Node l n r) = flatten l
                       ++ [n]
                       ++ flatten r
```

A tree is a search tree if it flattens to a list that is ordered.

Type inference

- `x = 1 + 2`

  `1` has type `Integer`, `2` has type `Integer`, adding two Integers results in another `Integer`, therefore `x :: Integer`. [1]

[1] Actually, member of `Num` type class is inferred; but, `Integer` $\in$ `Num`.

Type inference

- `x = 1 + 2`

  `1` has type `Integer`, `2` has type `Integer`, adding two Integers results in another `Integer`, therefore `x :: Integer`. [1]

- `inc x = x + 1` With similar reasoning,
  `inc :: Integer -> Integer`

[1] Actually, member of `Num` type class is inferred; but, `Integer` $\in$ `Num`.

Type inference

- `x = 1 + 2`

  `1` has type `Integer`, `2` has type `Integer`, adding two Integers results in another `Integer`, therefore `x :: Integer`. [1]
- `inc x = x + 1` With similar reasoning,
  `inc :: Integer -> Integer`
- Explicit type annotations are possible:

```
inc :: Integer -> Integer
inc x = x + 1
```

[1] Actually, member of `Num` type class is inferred; but, `Integer` $\in$ `Num`.

Lists in Haskell

- The data-structure for almost everything is List
- Constructing lists:

```
[]          -- empty list
[1]         -- list with one element
[1, 2, 3]   -- a longer list
```

Lists in Haskell

- The data-structure for almost everything is List
- Constructing lists:

```
[]          -- empty list
[1]         -- list with one element
[1, 2, 3]   -- a longer list
```

- List patterns:
  - `x:xs` matches to any list with one or more elements
  - `x:y:z:xs` matches to any list with three or more elements
  - `[x]` matches to any list with one element
  - `[]` matches to empty list

Lists in Haskell

- The data-structure for almost everything is List
- Constructing lists:

```
[]          -- empty list
[1]         -- list with one element
[1, 2, 3]   -- a longer list
```

- List patterns:
  - `x:xs` matches to any list with one or more elements
  - `x:y:z:xs` matches to any list with three or more elements
  - `[x]` matches to any list with one element
  - `[]` matches to empty list

```
let x:xs = [1, 2, 3]
 -- x is 1
 -- xs is [2, 3]
```

Defining Functions

- Defined as equations (with pattern matching)

```
len1::[a] -> Integer
len1 [] = 0
len1 (x:xs) = 1 + len1 xs
```

Defining Functions

- Defined as equations (with pattern matching)

```
len1::[a] -> Integer
len1 [] = 0
len1 (x:xs) = 1 + len1 xs
```

- With lambda abstraction

```
len2::[a] -> Integer
len2 = \ x -> if (null x) then 0 else 1 + (len2 (tail x))
```

Defining Functions

- Defined as equations (with pattern matching)

```
len1::[a] -> Integer
len1 [] = 0
len1 (x:xs) = 1 + len1 xs
```

- With lambda abstraction

```
len2::[a] -> Integer
len2 = \ x -> if (null x) then 0 else 1 + (len2 (tail x))
```

- Note the function invocation syntax:

```
(len1 [1, 2, 3])
```

Haskell functions can be *curried*

```haskell
add :: Int -> Int -> Int
add x y = x + y

add3 :: Int -> Int
add3 = add 3

z :: Int
z = add3 4
```

Remark (Currying relies on the following isomorphism:)

$$A \rightarrow B \rightarrow C \cong (A \times B) \rightarrow C$$

Haskell is *pure*

- I.e., no side effects (e.g. assignments, etc.). For example, in

  `x = add 1 2`

  - a fresh variable `x` is bound to the value of `add 1 2`,
  - the value of `add 1 2` is not computed until the value of `x` is required (*lazy evaluation*),
  - `x` stays bound to `add 1 2` within the scope of definition.

Haskell is *pure*

- I.e., no side effects (e.g. assignments, etc.). For example, in

  `x = add 1 2`

  - a fresh variable `x` is bound to the value of `add 1 2`,
  - the value of `add 1 2` is not computed until the value of `x` is required (*lazy evaluation*),
  - `x` stays bound to `add 1 2` within the scope of definition.
- $\therefore$ Haskell functions are pure "mathematical" functions
  - Makes reasoning about programs feasible N.b., side-effects are necessary for realistic programming (for IO, efficiency, ...).
  - Haskell type system encapsulates all effects inside *monads*

Haskell is *lazy*

- Lazy evaluation (a.k.a., call-by-need): Never evaluate an expression, unless its value is needed
- Example: The following program is not erroneous.

```
omit x = 0
v      = omit (1/0)
main   = putStr (show v)
```

Parametric Polymorphism

- Examples:

```
id :: a -> a
id x = x

length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs

tail :: [a] -> [a]
tail []     = []
tail (x:xs) = xs

eval::(a -> b) -> a -> b
eval f x = f x
```

- Note syntax for type parameters

- Consider now a non-parameterically polymorphic function.

```
not_equal:: a -> a -> Bool ???
not_equal x y = if (x == y) then False else True
```

Type Classes

- Consider now a non-parameterically polymorphic function.

```
not_equal:: a -> a -> Bool ???
not_equal x y = if (x == y) then False else True
```

- There are requirements for `a`; Not all `a`'s will be acceptable.

Type Classes

- Consider now a non-parameterically polymorphic function.

```
not_equal:: a -> a -> Bool ???
not_equal x y = if (x == y) then False else True
```

- There are requirements for `a`; Not all `a`'s will be acceptable.
- The type bound to `a` must be *equality comparable*

Type Classes

- Consider now a non-parameterically polymorphic function.

```
not_equal:: a -> a -> Bool ???
not_equal x y = if (x == y) then False else True
```

- There are requirements for `a`; Not all `a`'s will be acceptable.
- The type bound to `a` must be *equality comparable*
- `a` must be an instance of the type class `Eq`

```
not_equal:: Eq a => a -> a -> Bool
not_equal x y = if (x == y) then False else True
```

Motivating Type Classes

- Primary motivation: Function overloading mechanism for Haskell (ad-hoc polymorphism)[2]
  - Overloading with type classes is akin to OO overloading
- Two different kinds of polymorphism in Haskell
  - Parametric polymorphism: one implementation covers all types
  - Ad-hoc polymorphism: same syntax for different implementations

[2]Wadler, Blott: "How to Make Ad-Hoc Polymorphism Less Ad Hoc", 1988

Type Classes (cont'd)

- Type classes represent a set of requirements
- Requirements are expressed as function signatures
- Default implementations for each signature can be provided
- Example:

```
class Eq a where
   (==), (/=) :: a -> a -> Bool
```

- The class definition can be read as: *A class of types that conforms to the specified interface*
- Note how the declaration of conformance is separate from the definition of a type (unlike, say, `implements` in Java)

Instances of Type Classes

- Members of type classes are called *instances*. A type is not an instance of a type class unless explicitly defined as such:

```
instance Eq Bool where
   True == True   = True
   False == False = True
   _ == _         = False
```

- This would be painful without parameterized instance declarations, referred to as "conditional instance declarations". Example:

```
instance Eq a => Eq [a] where
    [] == []          = True
    (x:xs) == (y:ys)  = x==y && xs==ys
     _ == _           = False
```

- `Eq a =>` is the context (constraint).

Constraining polymorphic functions

- If a function is not explicitly annotated with its type, constraints will be deduced with type inference

```
not_equal x y = if (x == y) then False else True
```

- From `x == y` it can be inferred that the types of `x` and `y` must be instances of `Eq`, and they must be of the same type.
- The type of `not_equal` is thus deduced to:

```
not_equal :: Eq a => a -> a -> Bool
```

- Type inference determines the least constrained function type (a.k.a., principal type).
- Type annotations are an important form of documentation
  - annotations are (usually) not essential
  - sometimes must to help the type inference process (polymorphic recursion)
- Consequence of type inference: a particular function name, such as `==` can only be required by one type class.

Inheritance in type classes

- ... is comparable to extending interfaces in Java
- Accomplished with conditional class denitions.
- The same syntax Eq a for expressing the context is used.

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

- To be an instance of `Ord`, type must meet the signature requirements listed in `Ord` and those of `Eq`.
- An instance declaration that makes a type an instance of `Ord` does not establish that the type is an instance of `Eq`!

Multiple type class constraints

- A single type parameter can be constrained with several type classes.
- E.g. a function that needs to compare values, and also show them as strings:

```
class Show a where
    show     :: a -> String
    show_min :: (Ord a, Show a) => a -> a -> String
    show_min x y = show (min x y)
```

Modules, Data Types, Libraries

- `data` vs. `newtype` vs. `type`
- records, tuples, lists
- `import`