CS4450/7450: Haskell in a Hurry

Bill Harrison

August 24, 2016

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
- A peculiar language feature: indentation matters
- Also: capitalization matters

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` ∈ `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` ∈ `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` ∈ `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)

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

- With lambda abstraction

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

- Note the function invocation syntax:

```haskell
(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:)

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

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.
- ∴ 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

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
```

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`