More Haskell and Equational Reasoning

Dr. Isaac Griffith
Idaho State University

Outcomes

After today's lecture you will:

- Learn more about Haskell
  - Pattern matching in function definitions
  - Higher order functions
  - Conditional Expressions
  - Let Expressions
  - Type Variables
  - Common List Functions
  - Type Definitions and Type Classes
- Learn and be able to use Equational Reasoning

Functions

CS 1187

Pattern Matching

- The standard means of defining a function requires a name which will take on a value during function application.
- However there is another approach called **Pattern Matching**
  - allows us to setup a series of cases, which against which the input is checked.
  - it is wise to start with the most specific and ending with the most general
- What can we match against
  - Constant values such as `3` or `"abc"`
  - Empty lists or empty tuples: `[]` or `()`
  - A placeholder for which we don't care about: `_`
- Example:

```haskell
is_three :: Int -> Bool
is_three 3 = True
is_three _ = False
```

Pattern Matching Tuples

- We can pattern match on tuples to have direct access to its contents:

```
fst :: (a, b) -> a  -- argument is a pair
fst (x, y) = x

snd :: (a, b) -> b
snd (x, y) -> y
```

- If we need access to the original tuple we can use the following notation:
  - `pair@(x, y)`
  - Here `pair` is the name storing the original argument, and `x` and `y` the contents

Pattern Matching Lists

Idaho State University | Computer Science

- Because we can construct lists with the cons `(:)` operator, we can use this to match on a list

```haskell
isEmpty :: [a] -> Bool
isEmpty [] = True       -- matches on the empty list (most specific)
isEmpty (x:xs) = False -- matches a list with at least one item, x, and a following list
```

- Again, we can access the original containing list as follows:
  - `list@(x:xs)`
  - Where `list` is the name storing the original list, `x` the first item of the list, and `xs` the remainder of the list (or the empty list).
- Additionally:
  - `(x:y:xs)` - allows access to the first two items of the list and will only match a list with 2 or more items
    - `x` is the first item, `y` is the second item, and `xs` the rest of the list, or the empty list

Higher Order Functions

- Haskell considers functions to be *first class objects*
  - Functions can be stored in data structures
  - Functions can be passed as arguments to other functions
  - Functions can be used to create new functions
- **First Order Function** - any function whose arguments and results are ordinary data values
- **Higher Order Function** - any function that takes a function as an argument, or that returns a function as a result
  - These lead to extremely power programming techniques
- **Full Application** - an expression providing all arguments to a function
- **Partial Application** - an expression providing less than the required arguments, which results in a new function

Example: `twice`

Definition

```haskell
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Equational Reasoning

Given an application: `twice sqrt 81`, we can work out its application as follows:

$$
\begin{aligned}
\mathit{twice} \quad & \mathit{sqrt} \quad 81 \\
= \quad & \mathit{sqrt} \quad (\mathit{sqrt}\ 81) \\
= \quad & \mathit{sqrt} \quad 9 \\
= \quad & 3
\end{aligned}
$$

Higher Order and Partial Application

Example: prod

Definition:

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)

prod :: Integer -> Integer -> Integer
prod x y = x * y
```

Partial Application:

```
g = prod 4      -- partial app
p = g 6         -- full app of g
q = twice g 3
```

Results:

```
p => 24
q => 48
```


ROAR

Conditional Expressions

- **Conditional Expression** - an expression using a `Bool` condition to make a choice

- Syntax:
  `if` *Boolean_expression* `then` *expr1* `else` *expr2*
  - All parts must be present
  - If the *Boolean_expression* is true, then *expr1* is executed
  - Otherwise, *expr2* is executed
  - Type of *expr1* and *expr2* must be the same

- Example:

```haskell
if 2 < 3 then "bird" else "fish"
```

- Example:

```haskell
abs :: Integer -> Integer
abs x = if x < 0 then -x else x
```

Bad Conditional Expressions

Idaho State University | Computer Science

- The following are examples of poorly constructed conditional expression (which won't compile)

```haskell
if 2 < 3 then 10              -- missing else expression
if 2 + 2 then 1 else 2        -- must be Bool after if
if True then "bird" else 7  -- different types
```

Local Variables: `let` Expressions

Idaho State University | Computer Science

- `let` expressions set up an explicit local scope to define a set of variables for use in an expression

- General form:

```
let equation
    equation
    ...
    equation
in expression
```

- Components of this are:
  - The *equations* define the variables local to the let scope
  - The `in` *expression* is the value of the entire `let` expression

- `let` expressions may be used anywhere an expression may be used

`let` Expression Examples

Idaho State University | Computer Science

```haskell
quadratic :: Double -> Double -> Double -> (Double, Double)
quadratic a b c
  = let d = sqrt (b^2 - 4 * a * c)
        x1 = (-b + d) / (2 * a)
        x2 = (-b - d) / (2 * a)
      in (x1, x2)
```

```haskell
2 + let x = sqrt 9 in (x + 1) * (x - 1)
  => 10.0
```

Type Variables

- Often we want to define functions that accept *any* type in their arguments
- To do this we use **type variables**
  - These must begin with a lower case letter (convention is to use `a`, `b`, and so on)
- Examples:

```haskell
fst :: (a, b) -> a
snd :: (a, b) -> b
```

- Functions using type variables are said to be **polymorphic**
  - Additionally, they enhance reusability

Common List Functions

CS 1187

`length` and `(!!)`

Idaho State University | Computer Science

The ‘length‘ Function

- Returns the number of elements in a list

```
length :: [a] -> Int

length [2,8,1] => 3
length [] => 0
length "hello" => 5
length [1..n] => n
length [1..] => infinite loop
```

The ‘!¡ (index) Operator

- Accesses a list element by index (starting at 0)

```
(!!) :: [a] -> Int -> a

[1,2,3] !! 0 => 1
"abcde" !! 2 => 'c'
```

take and drop

The 'take' Function

- extracts the first `n` elements from a list

```
take :: Int -> [a] -> [a]

take 2 [1,2,3] => [1,2]
take 0 [1,2,3] => []
take 4 [1,2,3] => [1,2,3]
```

The 'drop' (index) Operator

- removes the first `n` elements form a list

```
drop :: Int -> [a] -> [a]

drop 2 [1,2,3] => [3]
drop 0 [1,2,3] => [1,2,3]
drop 4 [1,2,3] => []
```

(++) and map

The '++' (append) Operator

- joins two lists (of the same type) together

```
(++) :: [a] -> [a] -> [a]

[1,2] ++ [3,4,5] => [1,2,3,4,5]
[] ++ "abc" => "abc"
```

The 'map' Function

- Applies a given function (first arg) to each element of a list (second arg)

```
map :: (a -> b) -> [a] -> [b]

map toUpper "the cat" => "THE CAT"
map (* 10) [1,2,3] => [10,20,30]
```

- Effectively this is a replacement for a **for** loop

`zip` and `zipWith`

The ‘zip‘ Function

- pairs up the elements of two lists

```
zip :: [a] -> [b] -> [(a, b)]

zip [1,2,3] "abc" => [(1, 'a'), (2, 'b'), (3, 'c')]
zip [1,2,3] "ab" => [(1, 'a'), (2, 'b')]
zip [1,2] "abc" => [(1, 'a'), (2, 'b')]
```

The ‘zipWith‘ Function

- applies a function to each pair of items from two lists

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

zipWith (+) [2,4,..10] [1,3,..10] => [3,7,11,14,19]
zipWith (*) [1,2,3] [1,2,3] => [1,4,9]
```

`foldl` and `foldr`

- **fold** - iteration across a list executing a function to reduce the list to an accumulated value

The 'foldl' Function

- A fold starting from the left

```
foldl (+) a [p,q,r,s]
  = (((a + p) + q) +  r) + s
```

  - `(+)` - function to be applied
  - `a` - starting value
  - `[p,q,r,s]` input list

The 'foldr' Function

- A fold starting from the right

```
foldr :: (a -> b -> b) -> b -> [a] -> b

foldr (+) a [p,q,r,s]
  = p + (q + (r + (s + a)))
```

  - `(+)` - function to be applied
  - `a` - starting value
  - `[p,q,r,s]` input list

`foldl` and `foldr`

Idaho State University | Computer Science

The ‘foldl‘ Function

```
foldl max 0 [1,2,3]
  => max (max (max 0 1) 2) 3
  => max (max 1 2) 3
  => max 2 3
  => 3

foldl (-) 0 [1,2,3]
  => (-) ((-) ((-) 0 1) 2) 3
  => (-) ((-) -1 2) 3
  => (-) -3 3
  => -6
```

The ‘foldr‘ Function

```
foldr (-) 0 [1,2,3]
  => (-) 1 ((-) 2 ((-) 3 0))
  => (-) 1 ((-) 2 3)
  => (-) 1 -1
  => 2

foldr (:) [3,4,5] [1,2]
  => (:) 1 ((:) 2 [3,4,5])
  => [1,2,3,4,5]

foldr (||) False [True, False, True] => True
```

Composition

The '.' (composition) Operator

- Allows us to create a pipeline of function applications, each of which is awaiting an argument

```
(toUpper . toLower) 'A' => 'A'

((:) . toUpper) 'a' "bc" => "Abc"
```


ROAR

Data Type Definitions

CS 1187

Data Type Definitions

- Often we need to define data types that are better suited to our needs than lists or tuples

- **Algebraic Data Types** - a flexible form of user-defined data structure is there to help.
  - Additionally, these support pattern matching (just as Lists and Tuples did)

- Example: Colors

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

Constructors and Fields

- Each of the color names (i.e., `Red`, `Orange`, etc.) are **constructors** for the type `Color`
  - Constructors always start with a *capital letter}}
    - Consider the definition of `Bool`:

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

  - Thus a list like: `[Red, Orange, Yellow]` has a type of `[Color]`

- Defining types like this is great, but often we want values that contain fields
  - This allows us to associate information with each of the values
  - Example:

```
data Animal = Cat String | Dog String | Rat
```

Type Variables

- If we want to associate arbitrary information, we can use **type variables**, for example

```haskell
data Animal a b
  = Cat a | Dog b | Rat

data BreedOfCat = Siamese | Persian | Moggie

Cat Siamese  :: Animal BreedOfCat b
Cat Persian  :: Animal BreedOfCat b
Cat "moggie" :: Animal String b
Dog 15       :: Animal a Integer
Rat          :: Animal a b
```

`show` and `Show`

Idaho State University | Computer Science

- Now if we were to use any of these types in GHCi, we would run into errors anytime it attempts to print one of the values.
  - This is because, to print the values it uses the function `show`
  - `show` takes a type derived from `Show` and prints a `String` representation of the value to the console
  - We can adjust the `Animal` and `Color` types to accommodate this as follows:

```
data Color = Red | Orange | Yellow
           | Green | Blue | Violet
           deriving Show

data Animal a b
  = Cat a | Dog b | Rat
  deriving Show
```


ROAR

Maybe and Just

- Often we write a function that may or may not succeed in computing its result
  - If it succeeds, it returns the result, otherwise it will cause an error and the program will crash
  - To address this we have the `Maybe` type

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

  - So in the case the computation succeed we return a `Just a` and otherwise we return `Nothing`

Maybe and Just

- Examples:

```
phone_lookup :: [(String, Integer)] -> String -> Maybe Integer
...

phone_messsage :: Maybe Integer -> String
phone_message Nothing = "Telephone number not found"
phone_message (Just x) = "The number is " ++ show x
```

Type Classes and Overloading

CS 1187

Class Constraints

- There are many operations in Haskell, that can be used on some but not all types
  - Example: `(+)` which adds two numbers, but nothing else

```
(+) :: Num a => a -> a -> a
```

  - The “`Num`” is the name of a *type class* which includes `Int`, `Integer`, `Float`, `Double`
  - `Num a =>` is called a *class constraint* or *context*

ROAR

Type Classes

- **Type Classes** are sets of types sharing a common property
  - Most important type classes are `Eq`, `Show`, `Num`
    - `Eq` - denotes something that can be compared for equality
    - `Num` - denotes something that acts numerically
    - `Show` - denotes something that can be printed to the console

Implied Context

Idaho State University | Computer Science

- Additionally, when we define functions, we must be aware of what operators or function we use imply
  - For example, if we include `(==)` within our function definition
    - This implies that the involved operands derive from `Eq`
  - Example:

```
fun a b c = if a then b == c else False -- will not compile

fun :: Eq b => a -> b -> b -> Bool
fun a b c = if a then b == c else False -- will compile
```

- **Rule:** if your function uses an overloaded operator (one with a type that has a context), then *its* type must contain that context as well.

Equational Reasoning

CS 1187

ROAR

Equational Reasoning

- **Equational Reasoning** is both one of the most powerful and simplest forms of formal methods.
  - Requires only a basic understanding of simple algebra
  - Forms the basis of the most advanced formal methods

Equations and Substitutions

- An equation such as $x = y$ states the following:
  - **Substitution** - We can replace any instance of *x* with *y*, and vice versa
  - At least within the context of this definition
- Using substitution, definitions, prior axioms, prior theorems, and laws we can begin to reason about mathematics and programs.
  - For each step of reasoning, we provide the justification in curly braces (i.e., *x*)

```
-- definitions
x = 8
y = 4

-- proof
2*x + x/y
  = 2*8 + 8/y       { x }
  = 2*8 + 8/4       { y }
  = 16 = 2          { arithmetic }
  = 18              { arithmetic }
```

- Justifications may be

Hand-Execution

- **Hand-Execution** - technique or capability of a programmer to think through the execution of their program.
  - In the context of *imperative languages* such as Python, Java, or C++
    - We simulate the operation of the computer based on the commands in the program
  - In the context of *functional languages* such as Haskell
    - We instead use Equational Reasoning

A Haskell Script

```
f :: Integer -> Integer -> Integer
f x y = (2 + x) * g y

g :: Integer -> Integer
g z = 8 - z
```

Equational Reasoning

```
f 3 4
  = (2+3) * g 4     { f }
  = (2+3) * (8-4)   { g }
  = 20              { arithmetic }
```

Equational Reasoning Considerations

- We must be careful during hand-execution when considering the following:
  - Use of parentheses
  - Variable names and scope
  - Multipledefinitions of a function
    - Use either the number `{ f.1 }` or pattern for justifications

Conditionals

- A conditional satisfies the following equations:

```
if True  then e2 else e3 = e2   { if True }
if False then e2 else e3 = e3   { if False }
```

- Example:

Script

```haskell
f :: Double -> Double
f x =
  if x >= 0
    then sqrt x
    else 0
```

Reasoning

```
f (-3)
  = if (-3) >= 0 then sqrt (-3) else 0   { f }
  = if False then sqrt (-3) else 0       { arithmetic }
  = 0                                    { if False }
```

Equational Reasoning with Lists

Idaho State University | Computer Science

- Proving the following theorem:
  - `length (map f (xs ++ ys)) = length xs + length ys`

- We require the following theorems:
  1. `length (++)`: `length (xs ++ ys) = length xs + length ys`
  2. `length map`: `length (map f xs) = length xs`
  3. `map (++)`: `map f (xs ++ ys) = map f xs + map f ys`

Equational Reasoning with Lists

- Proof:

```
length (map f (xs ++ ys))
  = length (map f xs ++ map f ys)         { map (++) }
  = length (map f xs) + length (map f ys) { length (++) }
  = length xs + length ys                 { length map }
```

Language's Role

- Due to the mathematical nature of Haskell, equations are actual equations
- This fact leads to Haskell's property of **referential transparency**
  - Allowing for substitution
- This is unlike imperative languages which feature *assignment* rather than equations

Rigor and Formality

- **Rigorous Proof** - A proof which is thought through, and does not contain shortcuts, but possibly omits trivial details.
  - includes only the essential details

- **Formal Proof** - A proof consisting of solid reasoning based on a clearly specified set of axioms
  - No details omitted
  - No sloppiness allowed
  - Can be checked using software

For Next Time

- Review Chapter DMUC 1.6 - 1.10 and 2
- Review this Lecture
- Come To Class

Are there any questions?

ROAR