From Ruby to Haskell: An Introduction

Curt J. Sampson
<cjs@cynic.net>

Tokyo Rubyist Meetup
2013-01-22

Introduction

- This presentation is an introduction to some of the cool features of Haskell in a way that should be comfortable for Ruby programmers.
- It won't really teach you to program in Haskell, but I hope it will show you why you should learn to do so.
- We're focused on showing what's available and why it's cool, rather than delving too deeply into the details of any particular feature.

What We Cover

- We will cover:
  - A fair amount of Haskell syntax.
  - Some things we do in similar ways in both Haskell and Ruby.
  - Some neat and powerful things Haskell does that Ruby doesn't.
- These slides and the example code files are all available in the github repo:

  `https://github.com/cjs-cynic-net/ruby2haskell-tutorial`

- If you know Ruby reasonably well (or can pick it up quickly!), most of the the Haskell code presented here should be easy to read and understand.
- If something seems too weird, well, remember that before you knew Ruby, this was weird, too:

```
a.collect { |x| ... }
```

The Interpreter

- Standard Matz Ruby comes with a REPL you can use to evaluate Ruby expressions:

```
$ irb
irb(main):001:0> 2 * 3
=> 6
irb(main):002:0>
```

- GHC, the Glasgow Haskell Compiler, also includes an interpreter:

```
$ ghci -v0              # verbosity level zero
Prelude> 2 * 3
6
Prelude>
```

Some Simple Syntax

- Ruby uses # for comments; Haskell uses --, as below.
- (Haskell also offers nesting open/close comment delimiters that may span multiple lines, `{- like this -}`.)
- Basic constant expressions in Haskell are awfully similar to those in Ruby:

```
2               -- integer
3.1415          -- floating point number
[1,2,3]         -- list of integers
'x'             -- character (a type Ruby doesn't have)
"Hello."        -- list of characters, or string
```

- Expressions with binary operators are pretty much the same, though the particular operators are sometimes different:

```
2 + 3                       => 5
2 * (3 + 4)                 => 14
[1,2] ++ [3,4]              => [1,2,3,4]
"Hello, " ++ "world."       => "Hello, world."
```

Calling Functions

- In Ruby we can call functions with or without parens around the arguments, which are separated by commas.

```
irb> printf("%.3f\n", 3.1415/2)
1.571
=> nil
irb> printf "%.3f\n", 3.1415/2
1.571
=> nil
```

- In Haskell, we never use parens for functions calls, and we use just space to separate the arguments:

```
Prelude> import Text.Printf
Prelude Text.Printf> printf "%.3f\n" (3.14159/2)
1.571
```

- Note the parens we need around the `3.14159/2` expresssion to make it a single argument, due to the precedence rules.

Infix Functions

- Two-arugment functions whose names are punctuation are *infix* functions, and are used between their arguments:

```
Prelude> 3.14159 / 2
1.570795
```

- But these are actually just normal functions, and can be applied in prefix form as well:

```
Prelude> (/) 3.14159 2
1.570795
```

- We can also apply non-punctuation functions that take two arguments in infix form:

```
Prelude> 15 `mod` 4
3
```

Defining Variables

- `name = value`; how simple can it get? Only, if we're using the GHC interpreter, we need to prefix the definition with `let` for reasons that will remain mysterious for the course of this presentation.

```
irb> x = 3                  Prelude> let x = 3
=> 3
irb> x + 1                  Prelude> x + 1
=> 4                        4
```

- Variables in Haskell are "single-assignment"; once defined in a scope, you cannot define it a second time in the same scope.
- `x = x + 1` is nonsensical mathematically and evaluates as an infinite recursion Haskell.

```
x = 1 + x
  = 1 + (1 + x)          -- Substituting the
  = 1 + (1 + (1 + x))    -- definition of 'x'
  = ...
```

Defining Functions

- Functions are defined as we do in math, but, as with function calls, without the parens.

```
irb> def add x, y
irb>     x + y
irb> end
=> nil
irb> add 2, 3
=> 5
```

```
Prelude> let add x y = x + y

Prelude> add 2 3
5
```

- The syntax seems simpler than Ruby, doesn't it?

Function or Variable?

- But wait, function definitions and variable definitions look awfully similar:

```
f x = 7          -- The 'x' argument is ignored.
g   = 7
```

Function or Variable?

- But wait, function definitions and variable definitions look awfully similar:

```
f x = 7          -- The 'x' argument is ignored.
g   = 7
```

- Is `g` a variable or a function with no arguments?

Function or Variable?

- But wait, function definitions and variable definitions look awfully similar:

```
f x = 7          -- The 'x' argument is ignored.
g   = 7
```

- Is `g` a variable or a function with no arguments?
- Like Ruby, it mostly doesn't matter; often functions look like variables and you can use them as such.

Function or Variable?

- But wait, function definitions and variable definitions look awfully similar:

```
f x = 7          -- The 'x' argument is ignored.
g   = 7
```

- Is `g` a variable or a function with no arguments?
- Like Ruby, it mostly doesn't matter; often functions look like variables and you can use them as such.
- Unlike Ruby, it *really* doesn't matter: there is no difference!

Function or Variable?

- But wait, function definitions and variable definitions look awfully similar:

```
f x = 7          -- The 'x' argument is ignored.
g   = 7
```

- Is `g` a variable or a function with no arguments?
- Like Ruby, it mostly doesn't matter; often functions look like variables and you can use them as such.
- Unlike Ruby, it *really* doesn't matter: there is no difference!
- You can think of "variables" as functions with no arguments, if you like.
- But really, all functions are stored in variables, just like any other objects.

More Complex Expressions

- In Ruby (think `collect` instead of `map` if you're Smalltalk-oriented):

```
(0..7).map { |x| x*3+1 } .select { |x| x.even? }
=> [4, 10, 16, 22]
```

- In Haskell we prefix things, so the "flow" runs the other way around:

```
filter (\x -> even x) (map (\x -> x*3+1) [0..7])
[4,10,16,22]
```

- Haskell's `\x -> ...` is lambda syntax, similar to Ruby:

```
f = lambda { |x| x + 1 }      # Ruby
f = \x -> x + 1               -- Haskell
```

Using (or Not Using) Lambdas I

- But hang on! In the last slide we used a lambda for the parameter to `filter` in Haskell, but not in Ruby:

```
(0..7).select { |x| x.even? }
filter (\x -> even x) [0..7]
```

- Well, we could use a lambda in Ruby, but that's even more verbose.

```
(0..7).select(&lambda { |x| x.even? })
```

- And we don't like verbosity and extra punctuation, do we?

Using (or Not Using) Lambdas II

- We don't like verbosity and extra punctuation. So in Haskell we don't use lambdas so often.
- Instead, we get even simpler than Ruby, and just pass the function itself:

```
filter even [0..7]
```

- This uses no syntactic sugar! (Because variables hold functions.)
- Ruby's prefix notation and syntactic sugar for blocks end up adding complexity and extra characters to simple things.

```
(0..7).select(even?)            # Doesn't work in Ruby.
(0..7).select { |x| x.even? }   # Sigh.
```

First Class Values

- Every language has "first class" values, which are, roughly, the ones that variables can hold.
- In languages like C, variables hold simple values but not complex ones:

```
int i = 7;
double x = 3.1459;

char s[] = "Hello.";      // s is a pointer to a string
int (*f)(int, int) = ... // f is a pointer to a function
```

- In C, strings and functions are not first-class values.

First Class Values in Ruby

- In Ruby the following are all treated the same:

```
3               # FixNum object
"foo"           # String object
[1,2,3]         # Array object
Array           # Class object
o               # object of arbitrary user-created class
```

- All of these are easily used in the same simple way:

```
def f x; ...; end
f 3
f "foo"
f [1,2,3]
f Array
f o
```

Functions as First Class Values

- A functional programming key point: treat functions as ordinary (or "first class") objects or values, no differently from integers, strings, lists or programmer-defined types.

Functions as First Class Values in Ruby

- Functions are not consistently first-class in Ruby. “Ordinary” functions are not stored in variables:

```
def inc_f x
    x + 1
end
inc_f 3                          # => 4

[1, 2].map inc_f                 # These don't work!
f2 = inc_f; f2 3                 # inc_f is not a variable.
```

- We have to do special things to put a function in a variable:

```
inc_l = lambda { |x| inc_f(x) }
```

- And then we have to use it differently.

```
inc_l 3                          # Doesn't work!
inc_l.call 3                     # => 4
[1, 2].map inc_l                 # Doesn't work!
[1, 2].map &inc_l                # => [2, 3]
```

Functions as First Class Values in Haskell

- In Haskell, as we've already seen, variables hold functions. So it's all very simple.
- In Haskell, given this:

```
filter (\x -> x > 2) [0..6]
```

- We can apply the standard "extract variable" refactoring:

```
f = \x -> x > 2      -- Prefix with 'let' in ghci.
filter f [0..6]
```

- Or, more simply:

```
f x = x > 2          -- Prefix with 'let' in ghci.
filter f [0..6]
```

Side Note: Ruby Gets Weirder

- Ruby has "lambdas" and "procs," and they're not the same thing, though they look awfully similar!

```
def call_lambda
    f = lambda { return; };      puts(f.class)
    f.call;                      puts("Printed!")
end
call_lambda  # Prints "Proc" followed by "Printed!"

def call_proc
    f = Proc.new { return; };    puts(f.class)
    f.call;                      puts("Not printed!")
end
call_proc    # Prints just "Proc"!
```

- More scary details at:

```
http://www.skorks.com/2010/05/ruby-procs-and-lambdas-
  and-the-difference-between-them/
```

But in Ruby the Idea's There

- Though not all functions/blocks/whatever are first-class, some are, and we do use them in the way that functional programmers do.
- So some functional programming ideas are also present in Ruby.
- This is not really a suprise, given that Matz says that he started with Lisp and threw in ideas from Smalltalk, which is itself quite functional (unlike most subsequent OO languages).
- "Let's call it MatzLisp from now on. ;-)"

```
http://blade.nagaokaut.ac.jp/cgi-bin/
  scat.rb/ruby/ruby-talk/179642
```

Using the Compiler

- We're now going to move on from the interpreter to executing programs from files.
- In the git repo, we have a simple `Main.hs`:

```
main = putStrLn "Hello, world."
```

- We can compile it by passing it to GHC, producing a standard binary program that we can run:

```
$ ghc Main.hs
[1 of 1] Compiling Main             ( Main.hs, Main.o )
Linking Main ...
$ file Main
Main: ELF 32-bit LSB executable, Intel 80386...
$ ./Main
Hello, world.
```

Using the Load-and-go Compiler

- But for our purposes, the `runghc` "compile and go" program is faster and easier:

```
$ runghc Main.hs
Hello, world.
```

- Or sometimes, even, the interpreter, with definitions loaded from a file:

```
$ ghci -v0 Main.hs
*Main> main
Hello, world.
*Main>
```

Sequencing Commands I

- `putStrLn` is not actually a regular function, becuase it's "impure" rather than "pure": that is, its effect is different depending on where and when you execute it. We often call these impure functions, "commands."
- Let's sequence a pair of commands using the `do` control structure to show the difference.

```
main = do { putStrLn "abc"; putStrLn "def" }
```

  This prints "abc" on one line, and "def" on the next.
- If we reverse them, the program produces a different result:

```
main = do { putStrLn "def"; putStrLn "abc" }
```

When Order Is and Isn't Important

- That these are two different things may seem obvious:

```
ad = do { putStrLn "abc"; putStrLn "def" }
da = do { putStrLn "def"; putStrLn "abc" }
```

- But consider that with pure functions, we neither need nor want them to be order-dependent. In Haskell (though not in Ruby), these two programs should produce the same result!

```
twice s = s ++ s
main = putStrLn (twice "abc")
```

```
main = putStrLn (twice "abc")
twice s = s ++ s
```

Order Dependency Makes Ruby Difficult

- Allowing much of your code not to have order dependencies makes analysis of it easier.
- This Ruby code is hard to understand and debug:

```
def f; 42; end
# ... hundreds of lines of code here ...
g(f)
# ... hundreds more lines of code here ...
def f; 84; end
# ... and yet hundreds more lines of code here ...
g(f)
```

- In the two uses of `g(f)`, it can be difficult to figure out which `f` is being used.

Lack of Order Dependency Makes Haskell Easy

- In Haskell, without order dependency issues, we merely need to look at the nearest scope with the definition.

```
x   = 100
f   = x + 1
g x = x + 1
h   =  let x = 300  -- "local variable"
        in x + 1
```

- Load up this example file in the interpreter and see what evaluating the various definitions gives you:

```
$ ghci -v0 ex/01_defs.hs
Prelude> g 200
201
```

- For any use of `x`, it's easy to find its definition.

Sequencing Commands II

- The compiler knows which functions are order-dependent and which are not, and will tell us if we use either type in the wrong way.

```
main =  do { putStrLn "abc"; reverse "def" }

Main.hs:1:30:
    Couldn't match expected type `IO b0' with
        actual type `[a0]'
    In the return type of a call of `reverse'
    ...
```

- Many languages, including Ruby, sequence *everything*, forcing you to choose an order even when you don't want to.
- As a fish in water, I bet you never realized that putting one line after another in Ruby is a control structure!

More Syntax, But Actually Less

- You probably noticed the grouping braces and semicolon statement separators appearing in our last example:

```
main = do { putStrLn "abc"; putStrLn "def" }
```

- Like Python, you can leave these out and Haskell will insert them automatically if you use the obvious indentation:

```
main =
     do putStrLn "abc"
        putStrLn "def"
```

- This is a bit more powerful than Ruby because it considers syntax after newlines:

```
s  = "abc"
   + "def"        # Not valid ruby
s  = "abc"
  ++ "def"       -- But ok in Haskell
```

Printing Non-String Values

- Let's go back to our old Main.hs (`git checkout Main.hs`) and print a non-string value:

```
main = putStrLn 3.14159
```

- This gives us a error, because `3.14159` isn't a `String`. We need something like Ruby's `inspect`, which in Haskell is the `show` function:

```
main = putStrLn (show 3.14159)
```

- Much like Ruby's `p` function, we have `print`, which combines `show` and `putStrLn`:

```
main = print 3.14159
```

- `show` (and thus `print`) works for many non-String values, and of course you can write your own routines for your own data types. You can even ask the compiler to write them for you automatically.

Functions that Iterate

- In the comparisons of Ruby and Haskell code in previous slides, we've seen iteration done in both with `map` and similar functions:

```
(0..7).map { |x| x*3+1 } .select { |x| x.even? }
filter even (map (\x -> x*3+1) [0..7])
```

- This is the most frequent way of iterating over data structures, both in Haskell and for many Ruby programmers.
- `inejct` is less commonly in Ruby, but its Haskell equivalent, `fold`, is very common. To sum a list of numbers:

```
[1,3,5,7].inject(0) { |acc, x| x + acc }
foldr (\x acc -> x + acc) 0 [1,3,5,7]
```

- (Here we won't get into the difference between left fold, `foldl` and right fold, `foldr`.)

Recursion

- When we roll our own iteration, as with most functional languages, in Haskell we use recursion.
- Due to tail call optimization, this is very efficient and doesn't grow the stack.
- In Ruby, we're often forced to use `for` and `while` loops due to the lack of tail call optimization.
- Haskell doesn't even have syntatic `for` and `while`, though there are some `for` functions, and you could write a `while` function easily enough.

Recursion Example

- Let's find the first item on a list satisfying a predicate.

```
find p xs =
    if p (head xs) then (head xs)
                   else find p (tail xs)

main = print (find (\x -> x > 3) [1,2,3,4,5])
```

- Try `runghc ex/02_find.hs`
- Change `x > 3` to `x > 7` and try running it again. We get

```
02_find.hs: Prelude.head: empty list
```

because `head` is undefined for empty lists. We'll later see some better ways of dealing with this.

Pattern Matching

- The previous example is not the normal way we'd write this in Haskell.
- There's a much nicer way of building our `find` function that avoids calling functions to take apart data structures and `if` statements to do different things based on the result.
- Instead we use *pattern matching*:

```
find p (x:xs) =
    if p x then x
           else find p xs
```

- Ruby doesn't have real pattern matching, though it sometimes fakes a limited version of it with regular expressions in `switch` statements.

How Does Pattern Matching Work?

- Let's see how this worked:

```
find p (x:xs) = if p x then x else find p xs
```

- `:` is the operator we use to prepend (cons) an element on to a list, constructing a new list:

```
Prelude> 2:1:[]
[2,1]
Prelude> 3:[2,1]
[3,2,1]
```

- The pattern `x:xs` above does the reverse operation, deconstructing the components of the list in to the given variables. The head goes in to `x` and the tail in to `xs`.
- This is not special compiler support for the list constructor; this works on all user-defined data types.
- We use this a *lot*.

Guards

- There's another bit of syntax, *guards*, we can use to make things look even nicer:

```
find p (x:xs) | p x         = x
              | otherwise   = find p xs
```

- This is similar to the mathematical notation for functions with alternatives depending on the input:

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ -(n+1)/2 & \text{if } n \text{ is odd} \end{cases}$$

The Missing Pattern

- But with our new pattern-matching and guarded code:

```
find p (x:xs) | p x         = x
              | otherwise   = find p xs

main = print (find (\x -> x > 7) [1,2,3,4,5])
```

- We're still getting a pesky error when we pass find an empty list:

```
$ runghc 03_find.hs
03_find.hs: ex/03_find.hs:(3,1)-(4,37):
    Non-exhaustive patterns in function find
```

- That missing pattern is the empty list pattern. We need to add a definition of our function to match that one as well:

```
find p [] = ...
```

The Missing Pattern Found

- Though it's not much different from what's happening anyway, let's make `find` throw an exception with a better error message on failure.

```
import Control.Exception

find _ [] = throw (PatternMatchFail "Not found.")
find p (x:xs) | p x       = x
              | otherwise = find p xs

main = print (find (\x -> x > 7) [1,2,3,4,5])
```

- We can have multiple definitions of a function so long as their patterns don't overlap. The first definition with a matching pattern is used.
- The underscore as the pattern for the first argument matches anything, and says that we're ignoring it.

The Type System

- So far, we've seen what you might call the run-time language of Haskell. But there's another whole language that we've not yet even touched on: the type system.
- The type system language (in its basic form) is not as powerful as the run time language, but it's an extremely important part of Haskell, and the source of much of its power.
- The type system language is "run" at compile time, and has finished execution by the time your program is compiled.

Type-checking Advantages: Better than Testing

- The type system, when used well, will find many errors that would be found through testing in other languages.
- But it's actually better than testing:
  - Testing shows that (some) things work.
  - Type checking shows that things cannot fail.
- As Edsger Dijkstra has said:

  > *The first moral of the story is that program testing can be used very effectively to show the presence of bugs but never to show their absence.*

- One way of looking at it is that it lets you *prove* your code has no errors.

Effect of Type Checking on Unit Testing

- You need to do much less tedious unit testing.
  - Non-trivial Ruby project (3000-5000 LOC)
    - about 40% test code.
  - Non-trivial Haskell project (>10,000 LOC)
    - less than 10% test code.
    - Probably even less had I known Haskell well from the start.
- Sometimes you're more quickly led to a good solution than you are with TDD.
- Just as it takes time to learn to use unit testing, it takes time to learn to program with type checking.

Type-checking Advantages: Efficiency

- A type-checked program can be more efficient at run-time because it can do work at compile time to avoid run-time checks.
- Consider: in Ruby, a function's parameter can be any object (in this sense, variables are untyped), and so an object must be queried at run time to see if it supports an operation.
- If the check has been done at compile time, we know at run time what the object must be, because the compiler guarantees it.
- We can sometimes use a (more efficient) wider type at runtime knowing some values will not be present, e.g., a signed int that holds a smaller range of unsigned values (e.g., an enumeration). This is known as *type erasure*.

Enumerated Data Types

- The simplest data type is a straight enumeration of all values that inhabit the type, as we see in our first example of the type-level language:

```
data Color = Red | Green | Blue
```

- This declares a new type named `Color`. Type names start with a capital letter.
- `Red`, `Green` and `Blue` are *constructors*.
- Constructors are declared in the type language but used in the run-time language, and so start with a capital letter to distinguish them from variables.

Using Constructors

- Given our data type, we use the constructors in the run-time language both to construct values of that type and in patterns that match these values:

```
data Color = Red | Green | Blue

main = print (aoi Green)     -- construction

aoi Red   = False            -- pattern matching
aoi Green = True
aoi Blue  = True
```

- Running this program, `04_colors.hs`, will print `True`.

Type Inference

- In the previous example, we didn't explicitly tell the compiler the type of the `aoi` function; it looked at the code and *inferred* it.
- All this time we've actually been using compile-time ("static") type checking, though you may not have noticed this.
- Type inference brings a lot of the convenience of dynamically typed languages without the overhead (delayed checks, run-time cost) of run-time type checking.

Type Declarations

- But often we want to declare our types, either to avoid ambiguity and confusion or because we trust ourselves to write the type more reliably than we can write the code.
- In `ex/05_typedecl.hs` we have our first examples of this:

```
go :: Color
go = Green

aoi :: Color -> Bool
aoi Red   = False
aoi Green = True
aoi Blue  = True
```

- The first declaration says, "`go` is bound to a value of type `Color`."
- The second, "`aoi` is bound to a function that, when applied to a value of type `Color`, evaluates to a value of type `Bool`."

The Type of Multi-argument Functions I

- Of course, we can have functions with more than one argument:

```
matching :: Color -> Color -> Bool
matching c1 c2
    | aoi c1     = aoi c2
    | otherwise = not (aoi c2)
```

- We can read this as, "`matching` is bound to a function that, when applied to two arguments of type `Color` evaluates too a value of type `Bool`.

The Type of Multi-argument Functions I

- Of course, we can have functions with more than one argument:

```
matching :: Color -> Color -> Bool
matching c1 c2
    | aoi c1     = aoi c2
    | otherwise = not (aoi c2)
```

- We can read this as, "`matching` is bound to a function that, when applied to two arguments of type `Color` evaluates too a value of type `Bool`.
- But wait! Those two arrows look the same. Something's funny here...

The Type of Multi-argument Functions II

- When we look at this declaration:

```
matching :: Color -> Color -> Bool
```

- Is this:
  - a function that takes two `Colors` and returns a `Bool`, or
  - a function that takes one `Color` and returns another function, which takes a `Color` and returns a `Bool`?

The Type of Multi-argument Functions II

- When we look at this declaration:

```
matching :: Color -> Color -> Bool
```

- Is this:
  - a function that takes two `Colors` and returns a `Bool`, or
  - a function that takes one `Color` and returns another function, which takes a `Color` and returns a `Bool`?
- Answer: Both! (But really, the first.)

Partial Application

- Functions in Haskell are normally in *curried* form:

```
matching :: Color -> Color -> Bool
```

- This allows *partial application*

```
-- partial application
matchingBlue :: Color -> Bool
matchingBlue = matching Blue

-- completed application
matchingBlueGreen :: Bool
matchingBlueGreen = matchingBlue Green
```

- We use this a lot, too. Remember our `find` function?

```
find (matching Blue) [Red, Green, Red, Blue]
```

- We can do partial application of infix functions as well:

```
find (>3) [1,2,3,4,5]
```

Point-free Style

- In the previous slide, we didn't use an explicit parameter like this:

```
matchingBlue :: Color -> Bool
matchingBlue c = matching Blue c
```

- But no need for the syntax: functions are first-class values.
- Just as these are algebraically equivalent:

$$f(x) = g(x)$$
$$f = g$$

so is this to the above:

```
matchingBlue = matching Blue
```

- This is referred to as *point-free* (or *tacit*) style.
- Point-free style moves the focus from the *data* to the *functions*.

The Point of Point-free Style

- First, a bonus function: `.` for function composition. Just as in math $(f \cdot g)(x) = f(g(x))$, in Haskell:

```
(f.g) x == f (g x)
```

The Point of Point-free Style

- First, a bonus function: `.` for function composition. Just as in math $(f \cdot g)(x) = f(g(x))$, in Haskell:

```
(f.g) x == f (g x)
```

- Point-free style is more concise and, once you're used to it, often more clear.

```
sum = foldr (+) 0   vs.   sum xs = foldr (+) 0 xs
```

The Point of Point-free Style

- First, a bonus function: `.` for function composition. Just as in math $(f \cdot g)(x) = f(g(x))$, in Haskell:

```
(f.g) x == f (g x)
```

- Point-free style is more concise and, once you're used to it, often more clear.

```
sum = foldr (+) 0   vs.   sum xs = foldr (+) 0 xs

fn = f . g . h      vs.   fn x = f (g (h x))
```

The Point of Point-free Style

- First, a bonus function: `.` for function composition. Just as in math $(f \cdot g)(x) = f(g(x))$, in Haskell:

```
(f.g) x == f (g x)
```

- Point-free style is more concise and, once you're used to it, often more clear.

```
sum = foldr (+) 0    vs.   sum xs = foldr (+) 0 xs

fn = f . g . h       vs.   fn x = f (g (h x))

membr = any.(==)     vs.   membr x lst = any (==x) lst
```

The Point of Point-free Style

- First, a bonus function: `.` for function composition. Just as in math $(f \cdot g)(x) = f(g(x))$, in Haskell:

```
(f.g) x == f (g x)
```

- Point-free style is more concise and, once you're used to it, often more clear.

```
sum = foldr (+) 0    vs.   sum xs = foldr (+) 0 xs

fn = f . g . h       vs.   fn x = f (g (h x))

membr = any.(==)     vs.   membr x lst = any (==x) lst
```

- Intermediate form of `membr`:

```
membr x = any (==x)
```

Use of Point-free Style

- Frequently used in Haskell due to concision and clarity.
- Especially good for combinator libraries (a cool DSL technique we unfortunately don't have time to discuss).
- Point-free style can also turn into a confusing mess.
- Use judgement and taste.

Type Constructor Parameters I

- As with constructors in Ruby, type constructors in Haskell can take parameters for the specific data that they "store", and we typically extract these values by deconstructing via pattern matching. `06_Length.hs`:

```
data Length = Length Int

(|+|) :: Length -> Length -> Length
Length x |+| Length y = Length (x+y)

main = print $ Length 3 |+| Length 7
```

- Bonus function: `$` is another way of doing precedence without parens. These two expressions evaluate identically:

```
head   ( [1] ++ [2] )
head $   [1] ++ [2]
```

Type Constructor Parameters II

- Constructors may have multiple params (`07_Point.hs`):

```
data Point = Cart Double Double | Polar Double Double

toCart :: Point -> Point
toCart xy@(Cart _ _) = xy
toCart   (Polar r θ) = Cart (r * cos θ) (r * sin θ)

distanceTo :: Point -> Point -> Double
distanceTo p1 p2 = sqrt $ (x2-x1)^2 + (y2-y1)^2
    where Cart x1 y1 = toCart p1
          Cart x2 y2 = toCart p2
```

- Here we use two different constructors for different representations of the same value.
- More bonus syntax: `@` to bind the constructed value to a pattern, `where` for local definitions, and ability to use Unicode alphabets in GHC.

Variables and Functions at the Type Level

- Real functional languages have variables and functions, and so of course our type language in Haskell has these too:

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

- `List` is a type-level function that, given an `a` (which is a type variable), creates a new type based on whatever `a` it's given.

```
l0 :: List Int
l0 = Nil

l1 :: List Int
l1 = Cons 1 (Cons 0 Nil)

la :: List Char
la = Cons 'c' (Cons 'b' (Cons 'a' Nil))
```

- Note also that this data type is recursive! The `Cons` constructor to create a `List` value takes a `List` value!

Infix Type Operators and Nicer List Construction

- We have infix operators in the type language, too; any punctuation sequence starting with a colon (`:`) is an infix type function.
- Let's use this to make a nicer syntax for constructing `Lists`:

```
data List a = a :. (List a) | Nil
infixr 2 :.

l1 :: List Int
l1 = 1 :. 0 :. Nil
```

Functions Using Our Types

- We can write functions using these new types generated by our type functions. For example,

```
sumListOfInt :: List Int -> Int
sumListOfInt Nil = 0
sumListOfInt (i :. is) = i + sumListOfInt is

sumListOfInt (3 :. 7 :. 12 :. Nil)
```

- But we can also write generic functions that work on any type generated by our type function. For example, when we want the length of the list, we don't care what's in it:

```
len :: List a -> Int  -- Works for List of any a!
len Nil = 0
len (_ :. xs) = 1 + len xs
```

Type Classes

- We've seen one kind of generic function that works on any type generated from a specific type constructor:

```
len :: List a -> Int  -- Works for List of any a!
```

- But sometimes we want to get even more generic than this, and have a function that does the "same thing" on completely unrelated data types that have similar semantics.
- This is the motivation for "duck typing" in Ruby.
- In Haskell, we use a *type class* for this. One big advantage of type classes is that they're type-checked by the compiler, so that they can't go wrong in use.
- Type classes are in some ways a similar concept to OO classes, but they are definitely not the same thing! They're much closer to interfaces.

Our Favorite Type Class

- Let's create a class of types where a value of each type is the favorite value. For each type we'll have a function `isFavorite` that tells us if a value is the favorite value or not:

```
class Favorite a where
    isFavorite :: a -> Bool

data Color = Red | Green | Blue
instance Favorite Color where
    isFavorite Green = True
    isFavorite _     = False
```

- A quick test in `ghci`:

```
$ ghci -v0 ex/09_Favorite.hs
*Main> isFavorite Blue
False
*Main> isFavorite Green
True
```

Type Class Function Default Definitions

- Unlike OO interfaces, we can provide default implementations of functions for a type class.
- Expanding our Favorite class to two functions:

```
class Favorite a where
    isFavorite, notFavorite :: a -> Bool
    isFavorite  = not.notFavorite
    notFavorite = not.isFavorite
```

- For `Color` we supplied only a definition of `isFavorite`, so `notFavorite` uses the default definition:

```
$ ghci -v0 ex/09_Favorite.hs
*Main> notFavorite Red
True
```

- Remember to supply at least one of the two definitions, or the defaults will mutually recurse forever!

Type Classes are Open

- Unlike types, which are *closed* and cannot be changed once defined, type classes are *open*.
- This means that you can add any type to any type class at any time, even if neither the type class nor the type were created by you, and even if the original creators of the type and type class were entirely unaware of each other.
- Let's make a pre-defined type a member of our new type class:

```
instance Favorite Int where
    isFavorite = (== 42)
```

- And test it:

```
$ ghci -v0 ex/09_Favorite.hs
*Main> isFavorite (42::Int)
True
```

Duck Typing in Haskell

- We can make functions that operate on values of any type in a class, so long as the function restricts itself to operations available in that class:

```
hasFavorite :: Favorite a => [a] -> Bool
hasFavorite (x:xs) = isFavorite x || hasFavorite xs
hasFavorite []     = False
```

- This works even on members of the class that are defined long after the function was written.
- Key point: it's statically type checked, so it can't break.

Automatic Derivation

- In the previous example, we had to explicitly write the `isFavorite` function for each type.
- But for some of the more popular type classes, we can ask the compiler to do the work for us.
- We've seen this earlier in the presentation with `deriving Show`.
- One example is the `Eq` class, which has $=$ and $\neq$ functions:

```
Prelude> data Color = Red | Green | Blue deriving (Eq)
Prelude> Red == Blue
False
Prelude> Green /= Blue
True
```

Type Class Inheritance

- Classes can require that their members also be members of another class; this is done via *inheritance*.
- Let's re-implement `Favorite` for types in class `Eq` (`ex/10_EqFavorite.hs`). This lets us use == in our functions.

```
class Eq a => Favorite a where
    favorite :: a
    isFavorite :: a -> Bool
    isFavorite = (== favorite)

data Color = Red | Green | Blue
    deriving Eq
instance Favorite Color where
    favorite = Green

instance Favorite Int where
    favorite = 42
```

And There's More...

- There's a lot more than this:
  - Combinators, used for domain specific languages
  - Algebraic structures: functor, monoid, etc.
  - The infamous Monad (which we can build from what we've seen here)
  - ...
- But you're probably sufficiently overwhelmed already.

Summary

- Haskell has tons of cool stuff.
- But it's all built from a relatively small language with minimal syntactic sugar or other warts.
- We already know some of the basics of functional programming from Ruby.
- You should learn Haskell!