The Monad.Reader/Issue5/Different Language

From HaskellWiki

< The Monad.Reader | Issue5

Contents

1 Haskell: A Very Different Language

by John Goerzen, first published in *Free Software Magazine* June 2005.

Many programmers are fluent in several programming languages. Most of these languages have some things in common. Loops and variables are fundamental features of most languages.

I want to show you a different way of solving problems. Haskell takes a different approach than you're used to -- to just about everything.

2 Why Haskell is interesting

There are quite a few things about Haskell that make it interesting and unique. Haskell has no loops because it doesn't need them. There is no for or while in Haskell.

Haskell has no loops because it doesn't need them.

Haskell has no equivalent of the variables that you're used to; it doesn't need them, either. Haskell is a functional language. In a language like Java or Python, your primary view of the world is an object. In Haskell, your primary view of the world is a function. I like to say that Haskell manipulates functions with the same ease that Perl manipulates strings. In Haskell, it's commonplace to pass around bits of code. This is a powerful concept.

Haskell manipulates functions with the same ease that Perl manipulates strings.

Haskell functions are also pure. Every time they're called with the same arguments, they'll return the same result. Functions in most languages can return different results each time they're called. The results may depend on things like a global counter or I/O. Haskell functions also have no side-effects. They won't stomp over a global variable.

Haskell is a lazy language. It never performs a computation unless it needs to. This is not just an optimization; it is a powerful way to view the world. Code that could be infinite loops or consume vast amounts of memory in other languages are simple, everyday tools in Haskell.

Haskell can be either interpreted or compiled to native machine code. It also interfaces easily with C. You can call C functions from Haskell with a minimum of hassle. Usually, you'll only need 2 or 3 lines of code to accomplish the call. Haskell also has interfaces to Java, .NET, and Python.

Haskell lets you write code in a surprisingly intuitive way. Reading Haskell code is easy, and reasoning about Haskell code is easy, too. You'll have less of a need for a debugger with Haskell.

To get you started, here's an example for a simplistic grep, written in Haskell:

```
import MissingH.List

main = do c <- getContents
 putStr (unlines(filter (\line -> contains "Haskell" line) (lines c)))
```

This will simply read data from standard input and display all lines containing the word "Haskell" on standard output. I'll go through this example with you in more detail and show you how it works.

2.1 The Haskell toolbox

To get started with Haskell, you'll need a compiler or interpreter. The most popular compiler is GHC, available from http://www.haskell.org/ghc/. Some Linux or BSD distributions have GHC packages available; look for a package named "ghc" or "ghc6". If your operating system doesn't have packages available, sources and binaries for many systems are available from the GHC homepage.

The GHC package actually includes a compiler (`ghc`) and an interpreter (`ghci`). Use whichever you like. If you prefer a smaller package that includes only an interpreter, try Hugs from http://www.haskell.org/hugs/. Many distributions also contain Hugs packages.

Both GHC and Hugs come with a basic library of Haskell code called fptools. A reference is available (http://www.haskell.org/ghc/docs/latest/html/libraries/index.html) from GHC's site. (*Editorial note:* fptools is outdated, use 'base'.)

The examples in this article will also use functions from MissingH, a library of useful functions written in Haskell. MissingH can be downloaded from http://software.complete.org/missingh/. Many other Haskell libraries are also available for your use. See the links at the end of this article for more information.

To compile a Haskell program with GHC, you could use a command such as `ghc --make -o program program.hs`. The examples here use MissingH, so you'll need to add `-package MissingH` at the beginning of your GHC command line. You can run Haskell programs with Hugs by saying `runhugs program.hs`.

2.2 Laziness at work

The grep example at the beginning of this chapter probably doesn't make much sense yet. Here's another version of it that does exactly the same thing, but breaks down the code into more manageable pieces:

```
import MissingH.List

filterfunc line = contains "Haskell" line

main = do c <- getContents
 let inputlines = lines c
 let outputlines = filter filterfunc inputlines
 let outputstring = unlines outputlines
 putStr outputstring
```

Let's analyze this version. First, I import the MissingH.List module. This module has the contains function that we'll be using.

Next, I create a function named "filterfunc". It takes one parameter, line. It calls the contains function, passing it two arguments: the string "Haskell" and line. The contains function returns a boolean value (Bool type in Haskell). So, filterfunc takes a string and returns a Bool.

Next, you see the main function. This is the entry point to the Haskell program, similar to "main()" in C programs. In Haskell, main takes nothing and returns an IO action. Actions will be covered in more detail later.

The main function starts by calling getContents. This returns the entire contents of standard input as a string. getContents is an IO action, so we use the <- operator to cause c to represent the result of evaluating the action.

Next, we set up several Haskell variables. The inputlines variable holds a list of strings. Each string represents one line from the input. The lines function takes a string, separates it by newline characters, and returns a list of the component lines.

The outputlines variable also holds a list of strings. It calls filter to eliminate all lines that don't contain "Haskell". filter is a function that takes a function as an argument. In this case, we pass along our own filterfunc. filter returns only those elements from the input list for which the passed function returns a True value. This model is quite popular in Haskell, and is a very simple illustration of passing functions around.

Then, the unlines function is called to combine this list of lines back into a string. Finally, this resulting string is printed.

If you look at this program from a traditional perspective, you'll think that this is a poorly-written program. You might think that it starts by reading the entire file into memory -- a bad thing if your file is huge. Not so in Haskell.

In Haskell, a string is a list of characters. Because Haskell is lazy, elements of a list are only evaluated when their contents are required for computation. And they can be garbage-collected whenever the compiler knows they won't be needed again. So, when you see c <- getContents, nothing actually happens right then.

In fact, nothing at all happens until the very last line in the program. That line demands the content from outputstring, which in turn follows up until it reaches getContents. It's only now that input is read.

2.3 Types and patterns

Haskell is a strongly-typed language like Java or C. However, you probably noticed that I supplied no typing information at all in the grep example. That is because Haskell has another unique feature: type inference. This means that Haskell can automatically determine the type of a piece of data by looking at how it is created and used in a program. Haskell can still catch type errors at compile time, but it saves you from the effort of manually declaring types all the time.

Haskell can automatically determine the type of a piece of data by looking at how it is created and used.

You can manually declare types for clarity or if you wish to make the type more restrictive than the inferred type. Here's an example of the grep program with explicit types given:

```haskell
import MissingH.List

filterfunc :: String -> Bool
filterfunc line = contains "Haskell" line

main :: IO ()
main = do
 c <- getContents
 putStr $ (unlines . filter filterfunc . lines) c
```

The declaration for filterfunc says that it takes a String and returns a Bool. If it took more parameters, you could put more arrows and types in the line; the very last one is the return value.

Types are closely related to patterns. Let's say that we wanted to write our own filter. Here's a way it might be done:

```haskell
import MissingH.List

filterfunc :: String -> Bool
filterfunc line = contains "Haskell" line

myfilter :: (a -> Bool) -> [a] -> [a]
myfilter _ [] = []
myfilter f (x:xs) =
 if f x
 then x : myfilter f xs
 else myfilter f xs

main :: IO ()
main = do
 c <- getContents
 putStr $ (unlines . myfilter filterfunc . lines) c
```

The

```haskell
myfilter
```

function is the new and interesting one here. Before I discuss how it works, there are several interesting things to note about its type declaration. This function is said to be polymorphic because it works on items of many different types. In this case, it can take a list of any type of item, a function that takes one of those items, and returns a list of the same type of items. The a in the type declaration represents this. The first parameter to myfilter is given to be a function itself. The second parameter is a list of items, and the return value is a list of the same type of items. Next, I declare the function itself. The line

```haskell
myfilter _ [] = []
```

means that if you call myfilter with an empty list, it returns an empty list. The underscore is a wildcard and means I don't care what function is supplied. In fact, _ [] is a simple instance of pattern matching in Haskell. Next, you see

```haskell
myfilter f (x:xs)
```

. In Haskell, the colon represents the list you get by adding a single item to the beginning of the list. So, this pattern will put the first item of the list into x, and the rest of the list into xs. Note that xs may be empty if the list has only one item.

Now, we call the passed function, passing in the current item. If the function returns True, we can think of the return value as being the current item plus the result of filtering the rest of the list. So, I say x : myfilter x xs. This becomes the return value; the function calls itself. This is recursion, and is the most common way to achieve in Haskell what would be looping in other languages.

You can also define your own data types in Haskell. Here's an example:

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

This defines a new polymorphic type, Maybe a. You can create a value of type Maybe a in two ways. First, you could simply say Nothing. Secondly, you could say Just x, where x is some value of type a. Pattern matching works just as well with custom types as it does with built-in types.

The Maybe type is, in fact, such a useful pattern in Haskell that it is defined for you in the Haskell Prelude - the set of functions and types available to every Haskell program. Functions that may either compute a value or generate an error frequently use Nothing to indicate a problem, or Just x to indicate a successful calculation.

2.4 Functions

You've seen a little bit of how versatile functions are in Haskell already, when I passed a function to filter. Let's look at some other things you can do with functions.

In the first grep example, you saw this: '\line -> contains "Haskell" line'. This declared a new function on the spot. The backslash begins a declaration. The function took one parameter (line), and calculates its result by applying the part on the right. Functions declared like this are often called anonymous functions because they are never bound to a name.

As you've probably noticed, to call a function, you list its name and all parameters to it, separated by a space. There is a unique twist to that. The contains function is defined in MissingH with this type:

```
contains :: [a] -> [a] -> Bool
```

Since a String is a list of Chars in Haskell, this works well for filtering Strings.

Let's say we call contains with only one argument. In most languages, that will generate an error. In Haskell, however, it returns a new function, with the leading arguments no longer needing to be specified. This is called partial application. So, the type of

```
contains "Haskell"
```

is

```
String -> Bool
```

. Note that the type isn't

```
[a] -> Bool
```

. Because the first argument was given as a String, we know the next argument must also be a String. So, instead of saying

```
\line -> contains "Haskell" line
```

, I could have said simply

```
contains "Haskell"
```

.

Did you notice the last line of the last grep example looked unusual? That line was:

```
putStr $ (unlines . myfilter filterfunc . lines) c
```

The period is a function composition operator. In general terms, where f and g are functions, (f . g) x means the same as f (g x). In other words, the period is used to take the result from the function on the right, feed it as a parameter to the function on the left, and return a new function that represents this computation. The dollar sign is a bit of syntactic sugar that simply removed the need to put everything after putStr in parenthesis.

2.5 Variables

Recall that I said that Haskell has no variables in the conventional sense. You might be wondering about the let statements in the second grep example. Haskell does have "variables", and let is one way to declare them. A Haskell variable doesn't hold a value and can't be modified. Instead, a Haskell variable tells the compiler, "if you ever need to know the value of x, here's how you calculate it." Assigning something to a variable doesn't cause it to be calculated; in fact, if the value is never needed, it will never be calculated. Thus, a variable in Haskell is just a shortcut, similar to a macro in some other languages.

2.6 Monads and I/O

You've seen a very small bit of the power of functions so far. Monads are used to combine functions together in a way similar to the period operator, feeding the result of one to the input of the next. However, Monads provide more capabilities. For instance, a monad can abort the processing of an entire chain when there is a problem anywhere along it. The Maybe monad, for instance, can receive Just 5 from one function, pass 5 to the next, receive Just 6 from it, pass 6 to the next, and continue doing that across many functions. If any function returns Nothing, the computations stop, and the result of the entire computation becomes Nothing. Otherwise, the result of the entire computation is the result of the last function in the chain.

I/O was historically a tricky problem for pure languages like Haskell. A function that reads data from the keyboard obviously can't be guaranteed to return the same thing each time it is invoked.

In Haskell, the IO monad is used. The IO type is opaque, meaning that a Haskell program can't see "inside" it. By using constructs like <-, however, things can be read and written. The <- operator extracts the value from the inside of a monad type and assigns it to a variable. If you were using the Maybe monad and wrote x <- Just 5, then x would evaluate to 5.

The IO monad is inescapable, however. Once you call IO functions, your return value will be in the IO monad. That is, your return type might be IO Int or IO String. This provides a neat way of segmenting impurities.

Typically, Haskell programs are structured so that the outermost layers are in the IO monad, and computations are outside of it. The main function returns IO () -- an empty value in the IO monad. So, to execute a Haskell program, the compiler simply evaluates the I/O action that main represents, calling other functions as needed along the way.

2.7 Typeclasses: OOP in reverse

Object-oriented programming (OOP) is a fixture of many languages. OOP, in general, permits you to write code that accepts an object or any child of that object. It's a way to conceptualize the view of the world.

Haskell provides something similar called typeclasses. Typeclasses let your functions take data of any type, so long as a particular interface for that type exists. Instead of preventing us from accessing the internal representation of data in an object, typeclasses instead provide a way to handle many different types of data in a generic way.

Typeclasses provide a way to handle many different types of data in a generic way.

For instance, there is a built-in function called show. The show function can generate a string representation from many different data types. Its type is this:

```
show :: Show a => a -> String
```

You can read this as "The show function takes any value of type a, such that a is part of the typeclass Show, and returns a String." You can say show "Hi", or show 5.0, or even show True, and get a valid String.

You can add your own data types to the Show typeclass very easily:

```
data MyType = Red | Blue
instance Show MyType where
 show Red = "Red"
 show Blue = "Blue"
The Show class itself could be defined like this in the Prelude:
class Show a where
 showsPrec :: Int -> a -> ShowS
 show :: a -> String
 showList :: [a] -> ShowS

 showsPrec _ x s = show x ++ s
 show x = showsPrec 0 x ""
 showList = ...
```

Here, you can see that to be an instance of Show, one would normally have to provide three functions. However, in this case, defaults are provided, so really, only one function is required.

Typeclasses are powerful abstractions in Haskell. The Num typeclass, for instance, is used to provide an abstraction of arithmetic operators. The type of (+), the function representing the + operator, is `Num a => a -> a -> a`. Numeric types are all instances of Num, and thus + can be used with many different types of numbers. You can invent your own numeric types and, by simply making them instances of Num, all existing numeric operators will work with them.

2.8 Conclusion

Haskell is a powerful and flexible language. Its approach to solving problems is unique and refreshing. The ability to combine functions is powerful and time-saving. There is a great deal of power in Haskell that is easily tapped, but a magazine article such as this can just barely scratch the surface. I encourage you to seek out more detailed resources about Haskell.

2.9 For more information

Here are some resources for more information on Haskell.

For general information, look at:

- The Haskell home page (http://www.haskell.org)
- Tutorials and references
- Yet Another Haskell Tutorial, in my opinion the best Haskell tutorial available; http://www.isi.edu/%7Ehdaume/htut/

Libraries and code:

- Haskell at Freshmeat (http://freshmeat.net/browse/834/)
- Applications and Libraries in Haskell
- Category:Tools