Java in Chains: Sequential Computations Made Easy

Siva Somayyajula

Purely functional programming languages, like Haskell, impose the following restrictions on how programs can be structured: everything is a function, and all functions must be mathematical expressions. This raises the question: is there a way to create complex functions using only mathematical expressions?

The answer is yes: programmers often use *monads*, which are data structures that allow computations to be chained together and executed in sequence.

Mathematically speaking, monadic structures offer two operations: *return*, which takes a regular value and makes it monadic, and *bind*, which pipes monadic values into functions that take regular values but return monadic values. Thus, *bind* is considered to be the chaining/sequencing operation.

While this may seem cumbersome, it as actually *very* liberating. From personal experience, monadic code is almost always shorter, cleaner, and easier to understand. Consider the following Haskell snippet, which takes a list of filenames from standard input, and prints out the contents of each file, à la UNIX's cat.

```
import System.IO (readFile)
import System.Environment (getArgs)
import Control.Monad ((>=>))

cat = getArgs >>= mapM_ (readFile >=> print)
```

">>=" is the bind operation, which pipes the result of the left hand side into the right hand side. Thus, the list of filenames from the command line arguments is piped into mapM_, which takes every argument in the list and applies (readFile >=> print). This left-to-right Kleisli composition of monads is a fancy version of bind, and takes each filename, gets the contents of the files, and pipes it to standard output. Compare that to the equivalent Java 7 program, which is longer and less readable.

```
import java.util.Scanner;
import java.io.*;

public class Cat {
  public static void main(String[] args) throws IOException {
    for (String arg : args) {
      Scanner s = new Scanner(new File(arg));
      StringBuilder contents = new StringBuilder();
      while (s.hasNext())
        contents.append(s.nextLine() + : "\n");
```

```
            System.out.println(contents);
        }
    }
}
```

While monads are powerful, due to the type-strength of purely functional languages, they are limited to operate within their own context type e.g. input/output operations can only be sequenced with other input/output operations. As a result, sequencing computations of different types is decidedly nontrivial.

Thus, by exploiting the lack of type rigor in Java, I was able to make *chains*, a modified version of the monad. Unlike monads, chains can chain computations of different types together.

Chains are just as capable as monads; the following Java 8 snippet performs the same operation as the Haskell source code:

```
import java.util.Arrays;
import chains.core.IO;

public class ChainCat {
    public static void main(String[] args) {
        Arrays.stream(args).map(arg -> IO.readFile(arg).link(IO::print));
    }
}
```

The code is one-to-one with Haskell: map takes every argument from args and passes it to readFile, which gets the content of the file and passes it to print, which displays it on the screen (chain is analogous to bind).

And now to its killer app—here's something chains can do, but what regular monads *cannot* do: take user input which, if not null, gets printed to the screen. If it is null, nothing happens.

```
import chains.core.*;

public class ChainCat {
    public static void main(String[] args) {
        IO.getLine("Enter input here: ").chain(Option::new).chain(IO::print);
    }
}
```

Writing the equivalent in Haskell would require the use of nested monad transformers, or some other data structure, which is more difficult to conceptualize on the part of the developer. In that sense, chains harness both the power of monads and the freedom of Java. Happy coding!