Writing A Lisp Interpreter In Haskell

Monday, October 30, 2006

Introduction

A while ago, after what now seems like eternity of flirting with Haskell articles and papers, I finally crossed the boundary between theory and practice and downloaded a Haskell compiler. I decided to do a field evaluation of the language by two means. I was going to solve a problem in a domain that Haskell is known to excel at followed by a real world problem[1] that hasn't had much exploration in Haskell[2]. Picking the problems was easy. There's a lot of folklore that suggests Haskell is great for building compilers and interpreters so I didn't have to think long to a pick a problem that would be self contained, reasonably short, and fun - writing an interpreter of a Lisp dialect. For the second test case I decided to go with a web application - plenty of people are writing them for money and there hasn't been much Haskell-related work done in this area.

I've now completed the first test case and the results are incredibly promising. I was able to write an interpreter that can evaluate many of the examples from my lisp article in 188 lines of reasonably well designed, extensible, commented Haskell code. I made no effort to squeeze functionality into as few lines of code as possible, furthermore, because I am new to Haskell the code is probably much longer than it normally would be.

Beyond a doubt, Haskell is incredibly expressive. I could get far more functionality in with far less typing than I've normally come to expect of mainstream languages and I still had code that was easy to extend and modify. After getting such impressive results I *had* to tell the world. I quickly jot down the first draft of this article and immediately discarded it because I realized I'm preaching to the choir: people that use Haskell would already know what I was about to tell them and people that don't use it would find lines of code to be a terrible benchmark to draw any valuable conclusions from (and they'd be right).

You're now reading the second draft of this article which is about much more than the results of my first experiment. The article is about my *experience* with Haskell and the *process* of working with it. I will describe which features of Haskell I used to implement particular aspects of my interpreter, how they're different from the object oriented world, why they helped me get things done faster, and how I had to change my program whenever my initial efforts took me to a dead end. While the jury's still out how well Haskell performs in other domains (I'm just starting my web application project) I hope this article sparks your interest in this beautiful language and explains some things programmers new to Haskell often find confusing.

Abstract Syntax Trees

The first thing I did to kick off the interpreter project was to determine its scope. I decided that I would only deal with integers, symbols, functions, and lists - a bare minimum required to make something that resembles Lisp. At this point I needed to convert these concepts into code Haskell would understand - exactly the same thing I'd do with any other language. Immediately I was rewarded. I was able to create a type that supported a required data structure in only four lines of Haskell code! The definition was encoded into what Haskell calls an algebraic data type and looked like this:

```
data Expr = BlaiseInt Integer |
            BlaiseSymbol String |
            BlaiseFn ([Expr]->Expr) |
            BlaiseList [Expr]
```

Algebraic data types are tricky beasts. Because they're very different from everything people with imperative programming backgrounds are used to, they tend to cause much confusion for a long time until intuitive understanding finally develops. Above code simply states than an expression may be and integer, a symbol, a function, or a list, and each version contains a single member field, respectively an integer, a string, a function that takes a list of expressions and returns an expression, and a list of expressions. The closest (*very* lossy) translation to Java that I can think of is this:

```
abstract class Expr {}

final class BlaiseInt extends Expr {
    public int _value;
}

final class BlaiseSymbol extends Expr {
    public String _value;
}

interface BlaiseFunction {
    public Expr evaluateFunction(List arguments);
}

final class BlaiseFn extends Expr {
    BlaiseFunction _value;
}

final class BlaiseList extends Expr {
    List _value;
}
```

Notice how the Java version is already much longer than the Haskell one despite the fact that I took obvious shortcuts and avoided writing constructors and accessor methods. Additionally, as we continue developing the interpreter, extending the Java translation appropriately will become a very costly and verbose task. I will attempt to demonstrate what I mean at various points in the article. One example I can provide immediately is actually *instantiating* the data structure. For example, suppose I want to use it to create a list of three integers. In Haskell I can do this:

```
myList = BlaiseList [BlaiseInt 1, BlaiseInt 2, BlaiseInt 3]
```

A Java alternative would look like this (we'll even give it an advantage, assume *BlaiseList* and *BlaiseInt* have appropriate constructors):

```
BlaiseList myList = new BlaiseList();
myList._value.add(new BlaiseInt(1));
myList._value.add(new BlaiseInt(2));
myList._value.add(new BlaiseInt(3));
```

While we could (unnecessarily) transform the above Haskell code to take up four lines, the clarity is already starting to speak for itself: you can do more with Haskell in less code that's easier to read and maintain in the long run.

Printing

In the previous section we developed a data structure capable of holding Lisp's abstract syntax trees (a famous Lisp list). One common task all Lisp interpreters are able to perform is to print out this data structure. Haskell has an alternative to Java's *toString*, a polymorphic function *show*. By default, *show* will normally print out the values the way you type them in Haskell source code (which makes it far more useful than *toString*, but we won't get into that). Java's *toString* does something similar. Both of these implementations aren't very useful to us because we need our interpreter to print the data structure out via s-expressions. Therefore, we must customize this functionality. In Haskell we can do it like this:

```
instance Show Expr where
        show (BlaiseInt x) = show x
        show (BlaiseSymbol x) = x
        show (BlaiseFn x) = "<function>"
        show (BlaiseList x) = "(" ++ unwords (map show x) ++ ")"
```

Don't let the word *instance* throw you off - the code above has nothing to do with instances in a conventional object oriented sense of the word. Instead, it tells Haskell that *Expr* "implements" a *Show* "interface". This allows anyone to call *show* on our data structure and it will print in out as an s-expression. The code above uses pattern matching - the function *show* is defined four times. When *show* is called, Haskell runtime will examine available definitions and will call appropriate one depending on which "form" of expression it's dealing with. This is similar to Java's virtual functions but a lot more powerful. In Java we can define four versions of *toString* and the runtime will call the correct one depending on the instance type. However, Haskell pattern matching can match on far more than the type - it can also match on values, boolean expressions (using guards), particular structures, etc. The tremendous benefit of this technique will become more apparent later. For now, let's try to implement a subset of this code in Java: a *toString* method to render a list.

```
String toString() {
    StringBuffer output("(");
    Iterator it = _value.iterator();
    if(it.hasNext()) {
        Expr value = (Expr) it.next();
        output.append(value.toString());
    }
    for (; it.hasNext(); ) {
```

```
            output.append(" ");
            Expr value = (Expr) it.next();
            output.append(value.toString());
        }

        output.append(")");
        return output.toString();
    }
```

Yet another Haskell one-liner turned into sixteen lines of Java code. In isolation such examples mean little but once you run into them again and again and again you can see an interesting picture start to emerge.

Parsing

We're now able to create Haskell data structures that represent Lisp's abstract syntax trees and print them as s-expressions. The next logical step is to write a parser so we can build these structures using Lisp's infamous parentheses. One good thing about writing a Lisp interpreter is that the parser is very simple - most of the time we can write it by hand without resorting to advanced parsing tools. This approach has obvious benefits (no separate build steps to generate code, no other tool to learn, ability to easily debug the parser without trying to understand generated code, etc.) and I took it when I wrote Lisp interpreters in C and Java. My initial approach to writing a parser in Haskell was to do the same thing - write it by hand without resorting to parser generator tools. However, very quickly I realized that Haskell is different enough to give me the best of both worlds.

Haskell comes standard with a parsing library called *Parsec* which implements a domain specific language for parsing text. If you're familiar with Boost Spirit library written in C++ you'll understand what I mean. The library "embeds" a parsing language into Haskell. The user of the library can specify a parser's grammar directly in Haskell code! *Parsec* takes some time to get used to, but once you understand it you can drop the conventional parser generators forever. To give you a bit of flavor for *Parsec*, here's a code snippet that parses a symbol:

```
parseSymbol = do f <- firstAllowed
                 r <- many (firstAllowed <|> digit)
                 return $ BlaiseSymbol (f:r)
        where firstAllowed = oneOf "+-*/" <|> letter
```

The code is rather descriptive. It first looks for the first allowed character (traditionally no digits) followed by many of these same characters or digits. You can see the rest of the grammar in the source code of the interpreter. This is but one example of how domain specific languages eliminate crude tools, extra build steps, debugging pain, and repetitive work. Haskell is an excellent host for domain specific languages - perhaps as close to Lisp as any language can be. A word of warning: using JavaCC after learning *Parsec* becomes a very unnerving experience.

Evaluation

We've now come to a point where we can get to the core of the interpreter - the code that evaluates Lisp expressions. This bit demonstrates a number of Haskell features: type inference (along with optional type declarations), pattern matching, additional power of algebraic data types, functions as first class objects, and power of higher order functions. Let's view the code snippet and discuss these features one by one.

```
eval :: Expr -> Expr
eval (BlaiseInt n) = BlaiseInt n
eval (BlaiseSymbol s) = ctx Map.! s
eval (BlaiseFn f) = BlaiseFn f
eval (BlaiseList (x:xs)) = apply (eval x) (map eval xs)
        where apply :: Expr -> [Expr] -> Expr
              apply (BlaiseFn f) args = f args
```

This code defines a single function, *eval*, that evaluates our Lisp expression. The code looks strange if you're unfamiliar with Haskell but it's just a matter of learning a few powerful features before you can warm up to it. Let's start with type inference.

The first line is a *type declaration* - we tell the compiler that *eval* is a function that takes a single argument of type *Expr* and returns a value of the same type. Notice that the declaration is separate from the rest of the code - it almost gives you a hint that it isn't always necessary. If you open the interpreter's source code you'll notice that functions very rarely have type declarations. To the uninitiated it may seem that Haskell is a dynamically typed language but it isn't - everything is resolved at compile time. Haskell does this with type inference and it works surprisingly well. Most of the time you don't have to specify types - Haskell will figure them out for you. Sometimes it will get confused and you'll have to add a type declaration but that's an exception rather than the rule. If you're confused about what the type of something is, you can always start an interpreter and ask (:t something). Most of the type declarations in Blaise aren't necessary - I only put them there for clarity (I wonder how many lines of code I'd save if I removed them). After you get used to type inference it's really hard to come back to work and write my share of Java code:

```
ReallyLongClassName i = (ReallyLongClassName)foo.getBar();
```

The next line is first in a series of *pattern matches*. This feature is a mix of virtual functions, regular expressions performed on live data structures, and an abstraction over large switch statements. The four lines after the type declaration specify four cases of pattern matching and what the value of the *eval* function should be in those cases. For example, the second line states: "if the first (and only) argument is a blaise integer, just let it return itself" (because a value of 5 is 5).

In the example above pattern matching does something similar to what virtual functions do in Java - it executes different code depending on the type of the argument (in Haskell's case we can look at the type as well as the constructor of the algebraic data type, among other things). However, the abstraction is much more powerful. In principle, virtual functions are just a large switch statement like the one in the pseudo-code below:

```
if(typeof(something) == someType) {
    doSomething();
} else if(typeof(something) == someOtherType) {
```

```
    doSomethingElse();
}
```

Traditional object oriented languages let us abstract ourselves from writing such switches and take the task upon themselves freeing us up to do more important work. Haskell does the same thing, however, it doesn't limit the abstraction to types - we can get the compiler to write such switches based on any component we're interested in abstracting! We can, for example, get the compiler to write code for us that will cause the runtime to only execute a particular function if the first argument is equal to five, or if the second element in a list is equal to three. We could do that in traditional languages with *if* statements and switches but Haskell does this for us in a much more elegant and expressive way. After all, why are we only allowed to create virtual functions based on the type of the object they belong to? Why can't we define "virtuality" based on the type (or value) of the function's third argument, for example? Interestingly, this is one of the reasons why Haskell isn't object oriented - it's functions don't need to belong to objects because the compiler and the runtime system allow us to abstract over any argument's type and structure (contrast that to being able to abstract only over an implicit *this*)!

In our case *eval* also takes advantage of functions being first class objects. Take a look at how we evaluate a symbol - we simply grab its value from the map. At this point the symbols are predefined and added to the map on startup like so:

```
ctx = Map.fromList [("*", BlaiseFn blaiseMul)]
```

In this case we added a multiplication operator. Now if we try to evaluate "*", we're going to get a function from the map that we can call! This is similar to getting an instance of a class that implements multiplication, except we don't have to deal with the boilerplate - Haskell will do the work behind the scenes.

We also use *higher order functions* (functions that take other functions in arguments) in order to evaluate the function's arguments. Take a look at how we use *map* - Haskell's function to iterate over a list. We pass it a function and a list, and *map* iterates the list and calls our function on each member. We could do this with a *for* statement (or if we're lucky with a *foreach*), but we don't have to - Haskell lets us build abstractions over boilerplate code we write over and over again without waiting for the compiler developers to write them for us (of course *map* is a standard part of Haskell but if it weren't we could easily implement it ourselves).

Variables

At this point our interpreter boils down to a simple calculator. We can evaluate mathematical expressions like this:

```
(+ 1 2)
(+ 1 (+ 2 3))
(* 5 (+ 3 4))
```

While this is a calculator with the spirit of Lisp, it can hardly be called an interpreter for a programming language. The next natural step in building our interpreter is to i

variables. Given the fact that Haskell doesn't allow mutable data and that our design is currently limited to four elements (integers, symbols, functions and lists), we're going to run into some problems.

The first problem is obvious: how are we going to implement mutable data in Haskell if it provides no support for it? We can't simply declare a global map of symbols that we can update - all Haskell "variables" can only be written to once. In order to solve this problem we're going to have to modify our design: the *eval* function will need to explicitly take the current state of the interpreter (a map of symbols) and return the update state along with the evaluated expression. Additionally we'll have to modify our *Expr* type because functions may end up modifying the symbol table. The resulting code looks like this:

```
type Context = Map.Map String Expr

data Expr = BlaiseInt Integer |
            BlaiseSymbol String |
            BlaiseFn (Context->[Expr]->(Context, Expr)) |
            BlaiseList [Expr]

eval :: Context -> Expr -> (Context, Expr)
eval ctx (BlaiseInt n) = (ctx, BlaiseInt n)
```

Our interpreter can now handle state! However, we still can't implement an assignment operator because of a more subtle problem. Consider the following expression:

```
(set i 5)
```

If we try to implement *set* as a function our interpreter will crash! The reason is rather simple - our code evaluates all arguments to functions as they are passed in. In the above example our interpreter will attempt to evaluate the symbol *i* and because it doesn't exist it will result in an error. Clearly functions aren't proper abstractions to implement such operators. We need to modify our *Expr* type again to introduce special kinds of functions that evaluate their arguments themselves. This way our implementation of *set* may choose to evaluate only the second argument and leave first as is. Lisp normally calls such functions special forms and we can easily define them like this:

```
data Expr = BlaiseInt Integer |
            BlaiseSymbol String |
            BlaiseFn (Context->[Expr]->(Context, Expr)) |
            BlaiseSpecial (Context->[Expr]->(Context, Expr)) |
            BlaiseList [Expr]
```

The last thing we need to do before we can implement assignments is change our *eval* function to treat simple functions and special forms differently. We can easily achieve this with pattern matching like this:

```
eval ctx (BlaiseList (x:xs)) =
        let (new_ctx, fn) = (eval ctx x)
            (last_ctx, eval_args) = mapAccumL eval new_ctx xs
            apply (BlaiseFn f) = f last_ctx eval_args
            apply (BlaiseSpecial f) = f new_ctx xs
        in apply fn
```

We can now implement the actual assignment operator. It turns out to only take up three lines of code.

```
blaiseSet ctx [(BlaiseSymbol s), e] =
        (Map.insert s eval_e new_ctx, eval_e)
                where (new_ctx, eval_e) = eval ctx e
```

That's it! We now assign values to variables and use them in our expressions:

```
(set i 5)
(+ i 1)
(set j (+ i 5))
(set j (+ j 1))
```

While the changes take a rather long time to describe, the actual code modifications are minimal. You can see how well a Haskell program adapts to changing requirements - there are no classes to refactor, fewer dependencies to track down, and most importantly, no implicit state to worry about. The type system is expressive enough to warn us at compile time about most types of errors we may introduce with our changes. We also don't have to worry about breaking the order in which things are called or the possibility that we broke state other components expect - all state management is explicitly managed by us and type checked by Haskell. In the next two sections we'll see two more examples of Haskell's ability to easily adapt to changes.

Refactoring Using Monads

The previous section probably made both Java and Haskell programmers cry out in horror. We have to pass state explicitly?! Well, yes and no. The fact that we have to formalize exactly what state our program will need is a good thing - it lets us minimize dependencies and maximize encapsulation which results in more robust programs. The real problem is accepting and returning the state in every function that needs it. This results in a lot of boilerplate code that we have to waste time writing and maintaining.

Fortunately Haskell lets us avoid these issues with *monads*. I will not go into a detailed description of monads here as they are beyond the scope of this article - your favorite search engine should bring up plenty of monad tutorials. The idea, however, is rather simple - monads use Haskell type system in innovative ways to let us abstract the boilerplate code away. For example, the *State* monad deals with passing the state for us so we can focus our efforts on solving problems.

We can use the *State* monad in our interpreter to clean up the code written in the previous section. The first thing we'll do is clean up our declarations that pass state explicitly:

```
data Expr = BlaiseInt Integer |
            BlaiseSymbol String |
            BlaiseFn ([Expr]->BlaiseResult) |
            BlaiseSpecial ([Expr]->BlaiseResult) |
            BlaiseList [Expr]

eval :: Expr -> BlaiseResult
```

We define *BlaiseResult* in terms of *StateT* monad which allows us to do composition with *IO*:

```
type BlaiseResult = StateT Context IO Expr
```

We can now rewrite our code to take advantage of the monad and avoid passing the state explicitly. Effectively *StateT* monad abstracts this away from us - we only have to write code to modify the state. The code that passes it around is written once when the monad is implemented (*StateT* comes standard with Haskell).

```
blaiseSet [(BlaiseSymbol s), e] =
        do eval_e <- eval e
           modify (\sym_table->Map.insert s eval_e sym_table)
           return eval_e

eval (BlaiseInt n) = return (BlaiseInt n)
eval (BlaiseList (x:xs)) = do fn <- eval x
                              apply fn
    where apply (BlaiseSpecial f) = f xs
          apply (BlaiseFn f) = do args <- mapM eval xs
                                  f args
```

A common misconception among Haskell beginners is that monads are an unfortunate evil necessary in a a lazy, side-effect free language. While it is true that monads solve the lazy IO problem, it is only one example of their power. As I plunged deeper and deeper into Haskell I began to realize that monads are a *desirable* feature that allows the programmer to build otherwise impossible abstractions. Monads, together with higher order functions, provide excellent facilities for abstracting away boilerplate code. In this sense Haskell's monads serve a similar purpose to Lisp's macros (though the concept is entirely different). Languages like Java include a more verbose alternative to higher order functions in the form of classes but completely lack alternatives to Monads and macros. Many language features (including exceptions and continuations, for example) can be implemented via Monads or macros. Languages that lack these features leave us completely at the mercy of compiler writers - there's no way we're going to get continuations in Java until Sun decides this feature should make it into the language.

Error Handling

Perhaps the most compelling feature of monads is *composition* - ability to mix and match monads that serve different purposes to write well designed, extensible code. We can demonstrate Haskell's ability to rapidly adapt to change by refactoring our program one more time - this time to implement error handling. At this point our interpreter will crash if we type in code that cannot be parsed or uses an undefined symbol. We can easily fix it with the standard *Error* monad - a monad that implements exceptions (try implementing something like that as a Java library!)

The first thing we need to do is define a type that will be responsible for handling errors. We'll use the *Error* monad to do most of the work and we'll compose it with the *IO* monad so we can still perform input output:

```
type BlaiseError = ErrorT String IO
```

We should now redefine *BlaiseResult* to account for a possibility of an error:

```
type BlaiseResult = StateT Context BlaiseError Expr
```

That's it! We can now rewrite our code to throw and catch exceptions - a feature implemented in Haskell as a library!

```
eval (BlaiseSymbol s) =
    do sym_table <- get
       if s `Map.member` sym_table == True
       then return (sym_table Map.! s)
       else throwError ("Symbol " ++ s ++ " is unbound.")

parse source =
        case (Text.ParserCombinators.Parsec.parse
                  parseExpr "" source) of
          Right x -> return x
          Left e -> throwError $ show e

do expr <- parse x
   evaledExpr <- eval expr
   liftIO $ putStrLn ((show evaledExpr))
   repl
`catchError` (\e -> do liftIO $ putStrLn e
                       repl)
```

We can now make syntax errors and try to look up symbols that don't exist and get a meaningful error message. I deliberately avoided handling errors that deal with calling functions with a wrong number of arguments or mismatched types. I encourage you to dive into the interpreter's code and implement this yourself - it will be a rewarding experience!

Control Structures

We now have one final touch before we can call our Lisp dialect a programming language: we must implement conditions and function definitions. Once we get this done we can write a program that will, for example, compute the Fibonacci sequence - a sure sign of a language approaching Turing completeness! Fortunately doing this is rather simple - all we have to do is add two more functions to our interpreter. We can implement conditional statements like this:

```
blaiseIf [condExpr, expr1, expr2] =
    do eval_cond <- eval condExpr
       if (0 `notEqual` eval_cond) then eval expr1
                                   else eval expr2
    where notEqual val1 (BlaiseInt val2) = val1 /= val2
```

Function definitions can be done in a similar manner:

```
blaiseFnArgs = ["args", "..."]
blaiseFn = do
    [(BlaiseList args), (BlaiseList body)] <- getSymbols blaiseFnArgs
    let newFn = do evalBody <- mapM eval body
                   return $ last evalBody
    return $ BlaiseFn newFn (map (\(BlaiseSymbol arg)->arg) args)
```

That's it! We can finally define *fib* to compute a Fibonacci sequence like so[3]:

```
(set fib (fn (n)
   (if (eq n 0) 1
       (if (eq n 1) 1
           (+ (fib (- n 1)) (fib (- n 2)))))))
```

We can now call *fib* as if it were any other function!

```
(fib 4)
(fib 10)
```

What's next?

You can download the sources for the interpreter here. Alternatively you can grab them via darcs:

```
darcs get http://sources.defmacro.org/blaise
```

Assuming you have GHC installed on your system, you can compile the interpreter with Cabal as you would compile any other haskell program:

```
runhaskell Setup.lhs configure
runhaskell Setup.lhs build
```

You will then find the natively compiled executable in *dist/build/blaise*. If you don't want to compile the interpreter on your own or you don't feel like installing GHC you can download a native Windows executable here. Start it up and you'll have a read-eval-print loop at your fingertips where you can evaluate samples of Lisp code from this article. If you're new to Haskell I strongly encourage you to play around with the source. Add more functions (logic functions like *or* and *and* are a good start). After you're done with those you can start adding standard lisp functions for dealing with lists (*list*, *car*, and *cdr* are a good idea).

Comments?

As usual, if you have any questions, comments, or suggestions, please drop a note at coffeemug@gmail.com. I'll be glad to hear your feedback.

[1]By "real world problem" I mean a problem that a reasonably large number of people gets paid for solving every day.

[2]There don't seem to be any definitive sources that suggest Haskell isn't good at something.

[3]Because the interpreter doesn't support newlines you have to type the *fib* function definition on one line for it to work.

Contact me with any questions or comments at coffeemug@gmail.com.