Term Project Report
Stabl

Kristoffer Haugsbakk

INF225
Universitetet i Bergen
December 10, 2013

Vision

I was initially planning on making a Forth-like language because of its simplicity, both conceptually and when it comes to implementation. I wanted to make a language that was very simple and also very easy to expand - for example by making it possible to define ones own "control structures" such as if-expressions/statements simply as functions (called "words" in Stabl). I was also intrigued by the idea of homoiconicity, or emphcode is data. I thought that this had a lot of potential for providing easy meta programming and program transformation, and I thought that the simple and flat syntactic design of Stabl would be a good fit for this; a Stabl program is essentially just a list of types and words, likewise with words, and words can easily be in-lined since there is no chance of name-conflicts since Stabl does not use variables. Instead of variables, Stabl uses "stack combinators", functions that manipulate the elements of the stack, instead of letting the programmer define and call variables. This is a common concatenative programming practice, perhaps because it is more natural to manipulate the top elements of the stack on a stack machine rather than calling variables, that might be stored in registers (though they might also be stored on a stack). This practice seems very cumbersome, but it can admit some benefits, such as not having to bother about garbage collection; if you can only create and use a value once, which often is the way you use values with stack combinators, then you do not have to worry about manually freeing or garbage collecting those values (obviously this depends on the implementation of the stack); these are so-called linear types. This might be problematic, though, for large data types, such as large arrays.

Technologies

I implemented the prototype of Stabl in Haskell with the usual tools, like GHC (compiler) and GHCi (interpreter). I also used Parsec, a parser library. There are more specialized libraries for programming language parsing, some of which are sub-libraries of Parsec, but I used the most common ones. I used HUnit for simple testing, which is a Haskell library for unit testing. I could have used QuickCheck, which is a library that tests user-defined invariants on the code. This style of testing should be very natural when implementing a language in another language, such as by defining an invariant on an add operator:

```
interpret(x y add) == x + y
```

for any value of x and y (values of these should be generated by QuickCheck).

Implementation

The implementation consists of a parser, an interpreter, a simple static checker and a REPL. I used Parsec for parsing, as I mentioned, which is a parser combinator library. This basically means that it lets you define parsers and combine them, essentially letting you define composable parsers. If you for example the two parsers `int` and `wordCall`, you can make a new parser that first tries to parse an `int` and, if that is not possible, parses a `wordCall`:

```
token =  int <|> wordCall
```

Parsec is monadic parser library, which means that the parser data type that Parsec uses implements the Monad typeclass[1]. A Monad is a typeclass that lets one model some kind of computation, where the results of later computations can depend on previous computations. When it comes to parsing, that means that how you parse later expressions can depend on what you have previously parsed. There is also another typeclass, called Applicative, that is weaker than Monad, since its computations can not depend on previous computations. What this means for parsing in particular is that an Applicative parser is powerful enough to parse any context-free grammar, but it will probably not be powerful enough to parse a context-sensitive grammar.

The interpreter uses techniques like we used in Rascal for the exercises; Haskell only has pure functions, so one has to pass in all the parameters that a function needs to compute its result (eg. there can not be any implicit, global state that a normal function refers to and uses). This turned out to be fine in the beginning, but I found that it would soon become messy if I had a lot of variables to pass around between each call to the interpreter. The interpreter needs the current stack of the program, the stack that has been consumed and a map of strings to word-definitions (so the environment of the execution). A more thorough implementation would also have used a return stack, which is a stack that can be used to store things like local variables, but more crucially the address of the return-function; this was not a concern since I was building an interpreter in Haskell.

Experiences

Haskell turned out to be a pleasant language to implement a language in. Parsing could be implemented in a very declarative way; simply use Parsec as an Applicative or a Monad and feed the results of successful computations directly to the various type-constructors to store the abstract syntax tree (AST) of the program. Something which was hard was to try to plug other parsers into my own parser in order to something like express that I want to discard all extra whitespace in the whole parser for my language, instead of throwing errors. The reason for this might be that Parsec has a lot of underlying plumbing, such as remembering at what lines an expression was parsed, propagating errors, etc.

Writing the interpreter followed a lot of the patterns that we have used in Rascal; use a recursive interpret function that takes care of all the alternatives that might be encountered in the program, and pass the whole state of the program as parameters to the interpret function. As mentioned, my interpreter has a lot of state; the various stacks and the map of user-defined words. It turns out that this can be abstracted by using monads. In particular, Monads can abstract the pattern on passing the same immutable parameter through a function (this would have been the case if I compiled Stabl and only allowed top-level user-definitions of words). It can also abstract passing "mutable variables". But if you want to express all of that you need to combine monads, which is not always straightforward. I chose to not go down this rabbit-whole, since I thought that my implementation was manageable enough as it was, but this might be a viable alternative in future iterations.

The REPL turned out to give me some headaches. Doing IO in Haskell has perhaps a bad reputation, since you need to do it within very strict boundaries (within a function with a return type of `IO a`, where `a` is any type). But it is in my opinion fairly straightforward to define simple IO actions, such as reading from a file, doing something with that input, and then writing something out (which is exactly what a compiler does). But the IO profile of a REPL is more complicated; you should be able to evaluate expressions, but to also store variables and your own defined functions. I also wanted to store the whole state of the program, ie. the stack, so that I could print it between user-input. This means that you need to store the state of the program between user input and printing output. It turns out that Haskell has specialized monads for purposes like this, but it was hard for me to get into this since the types quickly get opaque and complicated - these monads often get nested within the `IO` type. But working with only `IO` is fairly straightforward, so I chose to just pass the state of the program within the `IO` type around between the `IO` functions.

Results

I have implemented a small part of the imagined language; ints, the most essential built-in words[2], user-defined words and a simple REPL[3]. I also have a small Semantic Analysis module, which were supposed to be help in enforcing static semantics of the language, but this module is not used in this implementation. Some

[1]Typeclasses in Haskell are more or less like Interfaces in Java, only better (or so the Haskell gurus say).
[2]Recall that words are the functions of Stabl.
[3]Read Eval Print Loop.

basic features are lacking, such as some form of branching and looping/recursion. Recursion might be viable in the language as-is, given a way to perform branching. The language also lacks a collection-type, such as a list or an array. This might be possible to implement with quotations, which are sort-of “bracketed-stacks”; they let one push words and primitives on the stack without executing them right away, as they normally would have been in a post-fix-only language like Stabl. The syntax of an if-statement/expression could take on the post-fix-form of:

```
[branch1] [branch2] conditional if
```

Or an anonymous function:

```
[1 2 3 4 5] list [1 add] map
```

We see here the use of a quotation to construct a list with the `list` word.

Conclusion

Haskell turned out to be a good language for implementing the first iteration of my prototype of Stabl. One can do a lot with just basic libraries, but there are also more specialized libraries for implementing languages. It seems that you need to have a good grasp of monads and combining them in order to create expressive and modular code for implementing languages, especially when it comes to interpreting the language. Learning more about monads and how to combine them will be the next thing I do if I continue implementing this language in the future.