Implementing brainfuck in Haskell

Michael Haupt

August 13, 2010

1 Introduction

What is your way of getting your fingers dirty in a little starter project when you're learning a new programming language? I usually try to implement a brainfuck interpreter[1] and document it.

These days, I'm learning Haskell: having spent much pleasant time with dynamically typed programming languages (most prominently: Smalltalk) for some years, I wanted to learn a language that is *really* different. Haskell, having looked interesting for quite a while, was the natural choice, and the book *Real World Haskell* is a great tutorial.

So, without further ado, here's the *bf* interpreter, bit by bit. Note that Haskell supports, out of the box, literate programming. This PDF file was, via LaTeX, generated from a file that contains LaTeX and Haskell source code side by side—that file can be used to run the interpreter or to generate documentation likewise.

Disclaimer: The code described below is that of a beginner. Very likely, nothing is optimised, and bugs may be lurking. I will happily accept suggestions for improvement addressing both code and writing.

2 The Interpreter

2.1 Importing Required Library Functions

The first two lines of code are not really interesting, but required:

```
import Data.Char (chr, ord)
import System.Environment (getArgs)
```

The *chr* and *ord* functions are required for obtaining characters' ASCII values and for turning such numeric values back into characters. Recall that Haskell is statically typed and pretty picky when it comes to implicitly using a `char` as an `int` and vice versa, like in C.

Next, *getArgs* will be important since the interpreter will be compiled to a stand-alone executable that can be passed a *bf* file on the command line.

[1] For the easily offended, I shall henceforth abbreviate this as *bf*.

2.2 Interpreter State

The really interesting part of all this is, of course, the interpreter logic itself. I chose to implement the interpreter as a single function that manipulates a tuple representing execution state. Its type is defined as follows:

$$\textbf{type}\ \mathit{InterpreterState} = ([\mathit{Int}], \mathit{Int}, [\mathit{Char}], [[\mathit{Char}]], [\mathit{Char}], [\mathit{Char}])$$

This makes the name *InterpreterState* a synonym for a 6-tuple with the given elements: a list of integral numbers, a single integral number, a list of characters, a list of lists of characters, and, finally, two more lists of characters. (In Haskell, names of types begin with upper-case characters, and a list type is denoted by putting square brackets around another type.)

The six elements in the *InterpreterState* tuple have the following meanings:

- *[Int]*: this is the memory of the *bf* interpreter. It can be thought of as an array of numbers.
- *Int*: the pointer for reading from and writing to memory.
- *[Char]*: the code to be run by the interpreter. This is a string with all the *bf* characters (and also others, which are ignored). Haskell does have a type *String*, which is a synonym for *[Char]*, but I want to think of *bf* code as a sequence of single characters, so I chose this representation.
- *[[Char]]*: a list of code sequences; actually, continuations to be used for *bf*'s loop constructs. See below for details.
- *[Char]* (twice): the input consumed and output generated by the running program. Note that input and output are part of the interpreter's state: I have chosen a side-effect free way of implementing it. In part, this is because I have not yet fully grasped the ways of dealing with side effects such as output being generated by an interpreter while it is at work; but it is also because a pure solution, given the nature of Haskell, seemed more idiomatic.

2.3 The Interpreter Function

Defining the type signature of the *interpret* function is easy, as it simply transforms interpreter state:

$$\mathit{interpret} :: \mathit{InterpreterState} \rightarrow \mathit{InterpreterState}$$

This code states that the *interpret* function's type signature (*::*) implies that this function accepts one argument of type *InterpreterState* and returns a value of the same type.

2.4 Controlled Termination

Here is the first piece of actual interpreter code:

$$\mathit{interpret}\ (m, p, [\,], \mathit{cts}, i, o) = (m, p, [\,], \mathit{cts}, i, o)$$

Haskell supports pattern matching; i. e., one can give "examples" for values passed to a function, and describe the function's behaviour based on the example. The above definition means that whenever *interpret* is applied to an *InterpreterState* tuple with elements 1, 2, 4, 5, and 6 that can be named m, p, *cts*, i, and o, and an empty list in position 3, that same tuple is the value of the function.

In other words, when arbitrary values are passed in the interpreter state, but the third element (the code array of characters) is empty, then the passed interpreter state is the final state of this run. In yet other words, if there is no more code to execute, the interpreter terminates.

2.5 Memory Pointer Control

Here is how the read/write pointer is manipulated by executing the `>` and `<` *bf* commands:

$$\mathit{interpret}\ (m, p, \texttt{'>'} : \mathit{cs}, \mathit{cts}, i, o) = \mathit{interpret}\ (m, p + 1, \mathit{cs}, \mathit{cts}, i, o)$$
$$\mathit{interpret}\ (m, p, \texttt{'<'} : \mathit{cs}, \mathit{cts}, i, o) = \mathit{interpret}\ (m, p - 1, \mathit{cs}, \mathit{cts}, i, o)$$

Again, the *interpret* function is partially specified by means of patterns and the reactions to them. Most of the patterns on the left-hand side of the equals sign are identical to the pattern given above for termination. The difference lies in one place, namely the third element of the interpreter state tuple, which represents the *bf* code to be executed.

For example, in the first pointer manipulation rule, the code pattern is *'>':cs*. Recall that the code is a list of characters (*[Char]*). In Haskell, *x:xs* matches a list of items in to its head and its tail, binding the head element to x and the tail list to *xs*.

In this particular case, the head is not bound to a name, but matched against the character literal `>`, meaning that this rule matches if the code list consists of a `>` at the beginning, and arbitrary other characters. In other words, when the interpreter encounters a `>` *bf* instruction, this part of the *interpreter* function will be applied.

What happens on the right-hand side of the rule is that *interpreter* calls itself recursively, passing along an updated interpreter state tuple. In fact, the tuple is identical to the matched argument, with two exceptions: the memory pointer, p, is incremented by one, and the code list is reduced by the command that the interpreter has just consumed. The `<` command is treated analogously.

The fact that the *interpret* function calls itself recursively may feel awkward at first, because it looks as if, for very long *bf* programs, very deep call stacks would occur, probably leading to problems as maximum stack depth is exceeded. In fact, recursive function application is a very common idiom in functional programming languages where no loop control structures exist, and their run-time environments and compilers usually circumvent the stack depth problems by applying tail recursion optimisation.

In case the recursive application of itself is the last part of a function, a compiler will not generate a function call, but, basically, a jump to the beginning of the code with the parameters replaced. This allows for arbitrary recursion depths without any noteworthy stack effects.

2.6 Manipulating Memory Contents

The *bf* `+` and `-` commands are responsible for incrementing and decrementing the value stored in the current memory location:

$$\begin{aligned}&\mathit{interpret}\ (m, p, \texttt{'+'} : cs, cts, i, o) = \mathit{interpret}\ (\mathit{incr}\ m\ p, p, cs, cts, i, o)\\&\mathit{interpret}\ (m, p, \texttt{'-'} : cs, cts, i, o) = \mathit{interpret}\ (\mathit{decr}\ m\ p, p, cs, cts, i, o)\end{aligned}$$

Note how the patterns on the left-hand side work precisely as seen above. On the right-hand side, the same structure can be observed as well, only the interpreter state is now modified in the first position, implying an update in memory. The code for the `+` and `-` commands makes use of two auxiliary functions named *incr* and *decr*, which accept the memory and current pointer as arguments and appear to replace the memory with a new one. These functions will be explained below. For now, just accept that they increment and decrement the value stored in m at position p by one.

2.7 I/O

Output and input are next, represented by `.` and `,` in *bf*. Let us consider the output part first:

$$\begin{aligned}&\mathit{interpret}\ (m, p, \texttt{'.'} : cs, cts, i, o) =\\&\quad \mathit{interpret}\ (m, p, cs, cts, i, o +\!\!+ [\mathit{chr}\ (m\ !!\ p)])\end{aligned}$$

Output in *bf* means that the value at the current memory position is to be printed. In this purely functional interpreter implementation, it means that the character to be printed has to be appended to the output part of the interpreter state tuple. This is exactly what this rule does.

The m *!!* p expression obtains the value at position p from the list m. This value (recall it is an *Int*) has to be turned into a *Char* to be appended to the *[Char]* list representing output. The *chr* function achieves this conversion.

This expression is wrapped in square brackets to turn it into a one-element list. This is necessary because the $+\!+$ list concatenation function expects lists on both sides. In this case, the already existing output list o is extended with the one-element list containing the single character obtained from memory.

Implementing input requires a bit more effort:

$$\begin{aligned}&\mathit{interpret}\ (m, p, \texttt{','} : cs, cts, [\,], o) = (m, p, cs, cts, [\,], o)\\&\mathit{interpret}\ (m, p, \texttt{','} : cs, cts, j : i, o) =\\&\quad \mathit{interpret}\ (\mathit{putm}\ (\mathit{ord}\ j)\ m\ p, p, cs, cts, i, o)\end{aligned}$$

Note that there are *two* rules for matching the `,` command. They differ in the last-but-one entry in the interpreter state pattern.

The first rule matches `,` and an empty list, implying that there is no more input to consume. As this would lead to inconsistent state in the running program, the interpreter bails out in such a case, returning the state tuple as is. This behaviour effectively leads to interpreter termination, as above for the case where no more code is available to execute.

The second rule matches `,` and *j:i*, separating input into one single character *j* and the remaining input *i*. That is, input is consumed one character at a time, each time a `,` command is interpreted. The semantics of this command is to store the ASCII value of the input character at the current memory location. This is done by means of applying the auxiliary *putm* function (see below) to the ASCII value, which is obtained using the *ord* function, as well as *m* and *p*.

2.8 Loops

Loops are the most interesting construct in *bf*. To recapitulate, a `[` command checks whether the current memory position holds a zero value. If so, it jumps to the matching `]` and continues execution there. In case the value is not zero, execution proceeds as usual. The `]` command simply jumps back to the matching `[`, starting the loop over.

The `[` loop command is implemented as follows:

```
interpret (m, p, '[' : cs, cts, i, o) =
  if 0 ≡ m !! p
  then interpret (m, p, continue cs, cts, i, o)
  else interpret (m, p, cs, ('[' : cs) : cts, i, o)
  where continue cs = chelp 0 cs
    where
      chelp _ [] = []
      chelp i (r : rest)
        | r ≡ '['           = chelp (i + 1) rest
        | r ≡ ']' ∧ i > 0 = chelp (i − 1) rest
        | r ≡ ']' ∧ i ≡ 0 = rest
        | otherwise         = chelp i rest
```

The rule for `[` checks the value at position *p* in the memory using the familiar *!!* function. If it is zero, the interpreter state is changed in the code position, replacing the code to be executed with the result of applying the *continue* function to the remaining code stored in *cs*.

The job of the *continue* function (conveniently defined local to this particular rule in a *where* clause) is to find a matching `]` for the `[` that was just consumed by the corresponding interpreter rule, and to return all the code after that matching `]`. This is the code where execution should continue after the loop terminates. In the process of finding the correct position, matching pairs of square brackets must not be ignored; it is not sufficient to simply find the first `]`.

Since both the argument to and result obtained from applying *continue* are *bf* code, its type signature is *[Char]* → *[Char]*. The function applies a helper function, *chelp*, which maintains a bracket pair counter (initially, it has the value 0). The helper function is conveniently defined local to the *continue* function in a *where* clause.

The helper function is once again defined using patterns. Its first rule matches an empty list of characters and ignores the counter, which is expressed by the underscore in the position where the counter argument would have to be passed. Ignoring an argument means it is not bound to any name on the right-hand side of the function definition. In case *chelp* matches empty input, it simply returns an empty list: no matching `]` could be found.

All other cases are processed by the second rule, where the counter value is bound to i, and the code character list is split into the first element r and the remaining characters, which are contained in the list *rest*.

The *chelp* function needs to react in different ways to different input characters, and it does so using *guards*, which are expressed as Boolean expressions preceded by a pipe symbol. The first of the guards matches if the head character in the code array is a `[`, and it reacts by recursively applying *chelp* with the counter increased by one. This means that a nested pair of square brackets needs to be found before the matching `]` can be identified.

Matching a `]` can have one of two consequences. On the one hand, if the counter value is greater than zero, a nested pair has been matched, so *chelp* is applied with a decreased counter. On the other hand, if the counter value is zero, the desired matching `]` has been found, and the remaining code is the code to be executed—so it can be returned.

The last guard, denoted by *otherwise* (which is always true), works for all other characters that deserve no special treatment in the course of matching square brackets. In this case, *chelp* is simply applied to the remaining code without modifying the counter.

To summarise, if the interpreter executes a `[` command *and* the current memory location holds a zero value, the code with whose execution the interpreter must continue after loop termination is obtained using *continue*, and execution proceeds there.

In case the current memory location does not contain a zero value, the interpreter continues execution as usual with the code contained in *cs*. Note, however, that the fourth position of the interpreter state tuple is changed: it is replaced with the result of the expression *('[':cs):cts*.

As mentioned above, the *x:xs* construct represents a list with head element x and tail list *xs*. The construct can also be used to *prepend* an element to a list. Here, an element is prepended to the *cts* list, namely the result of prepending `[` to the remaining code. In other words, it is precisely the code that the interpreter rule matched that is restored (by prepending the `[` consumed by pattern matching) and prepended to the *cts* list.

The effect of this becomes clear after taking a look at how the interpreter deals with *bf*'s `]` command:

$$interpret\ (m, p, \texttt{']'} : cs, next : cts, i, o) = interpret\ (m, p, next, cts, i, o)$$

The pattern separates the fourth element of the interpreter state tuple into the *next* and *cts* parts, and the recursive application of *interpret* uses *next* as the code to be

executed. One can think of the elements of the *[[Char]]* item in the interpreter state as *continuations*: sequences of code that are stored for later execution.[2]

Now, the way this *bf* interpreter deals with loops can be wrapped up. In a nutshell, the `[` command, if the loop body needs to be executed, stores the entire code, starting with the respective loop, as a continuation in the interpreter state. At the end of the loop, `]` retrieves the continuation from the state tuple and establishes it as the code to be executed. Recall that the continuation contains the *complete* code until the end of the program.

2.9 The Rest Is Comment

There is one final rule in the *interpret* function that needs to be covered:

$$interpret\ (m, p, \_ : cs, cts, i, o) = interpret\ (m, p, cs, cts, i, o)$$

The pattern matches *any* character not matched by any of the other rules and *ignores* it (underscore). Recursive application of *interpret* simply uses the remaining code (cs). The purpose of this rule is to silently ignore all characters that are not part of the *bf* syntax.

3 Auxiliary Functions

Above, three auxiliary functions were mentioned, namely *incr*, *decr*, and *putm*. Their type signatures are as follows:

$$\begin{aligned} &incr :: [Int] \to Int \to [Int] \\ &decr :: [Int] \to Int \to [Int] \\ &putm :: Int \to [Int] \to Int \to [Int] \end{aligned}$$

Both *incr* and *decr* accept a list of integers (the memory) and an integer (the pointer) and return the updated memory. The *putm* function accepts one more *Int* argument, namely the value to be written to the memory position denoted by the pointer.

These functions all have one thing in common: they are supposed to update a given memory position in some way. The structure of these three operations is the same: the correct position in the *Int* list that is memory must be identified, the modification must be applied, and the updated list must be returned. The point of variation in this is the way the modification is applied. This calls for using higher-order functions to avoid repetition.

Here is a function called *updateAt* that provides the required framework:

$$\begin{aligned} &updateAt :: (Int \to Int) \to [Int] \to Int \to [Int] \\ &updateAt\ f\ m\ p = left +\!\!+ (f\ v) : right \\ &\quad \textbf{where}\ (left, v : right) = splitAt\ p\ m \end{aligned}$$

[2] I am a bit unsure whether it is fair to call these "continuations". Opinions?

Looking at the type signature, it is apparent that *updateAt* accepts a function mapping integers to integers (this is the function used to manipulate memory contents—the variation point mentioned above), a list of integers (the memory), and an integer (the pointer). It answers the updated memory.

The function concatenates two lists, namely *left* and the result of prepending the result of applying f to some value v to a list named *right*. These are defined in the *where* clause, which applies the standard *splitAt* function to the memory list. This function accepts an integer k and a list and returns a tuple with two elements, both of which are lists. The first list in the tuple contains the first k elements of the original list, and the second, the remaining elements.

In the *where* clause, pattern matching is used to directly bind the elements of the tuple (or parts thereof) to the three names required in *updateAt*. The *left* part of the list contains all the elements up to (but not including) the element at the position that needs to be updated. The second part of the tuple returned from *splitAt* is conveniently separated into the element that needs to be updated (v) and the remainder of the list (*right*).

It is now easy to see how *updateAt* works: it leaves the *left* and *right* parts of the list untouched, applies the update function f to the element to update, and returns a newly concatenated list containing all the original elements (and the updated one) in the correct order.

Implementing the three auxiliary functions is now easy:

$$
\begin{aligned}
&\mathit{incr} = \mathit{updateAt}\ (\lambda v \rightarrow v + 1)\\
&\mathit{decr} = \mathit{updateAt}\ (\lambda v \rightarrow v - 1)\\
&\mathit{putm}\ c = \mathit{updateAt}\ (\backslash\_ \rightarrow c)
\end{aligned}
$$

All three functions use *updateAt*, passing a *lambda expression* as the required manipulation function. In the case of *incr* and *decr*, the lambda expression simply increments or decrements its argument. In the case of *putm*, the argument is ignored (underscore) and replaced by c, which is the value to be written to the memory location in question.

It may seem strange that no full argument lists are given on the left-hand sides of these three definitions. They make use of *currying*, which allows for so-to-speak partially applying functions. For instance, *incr* recurs to *updateAt* but passes only one argument, namely the first one required by *updateAt* (a function mapping *Int* to *Int*).

Recall *updateAt*'s signature: $(\mathit{Int} \rightarrow \mathit{Int}) \rightarrow [\mathit{Int}] \rightarrow \mathit{Int} \rightarrow [\mathit{Int}]$. Partially applying a function in Haskell yields another function. In this case, partially applying *updateAt* by passing only the first argument (which is a lambda expression with the type signature $(\mathit{Int} \rightarrow \mathit{Int})$) will reduce the type signature by the applied-to arguments, yielding a function with the type signature $[\mathit{Int}] \rightarrow \mathit{Int} \rightarrow [\mathit{Int}]$. This is precisely the signature of *incr*.

The remaining arguments of *incr* do not have to be mentioned in its definition, as their names do not play a substantial role in its definition. To illustrate this, here is a definition of *incr* with all the arguments: $\mathit{incr}\ m\ p = \mathit{updateAt}\ (\lambda v \rightarrow v{+}1)\ m\ p$. The m and p arguments are merely repeated on the right-hand side. Currying allows for omitting them, abbreviating function definition and reducing it to the bare minimum.

4 Running bf Code

The interpreter is complete, now we need to be able to run some *bf* code. The *run* function is defined as follows:

$$
\begin{aligned}
&\mathit{run} :: [\mathit{Char}] \to [\mathit{Char}] \to [\mathit{Char}] \\
&\mathit{run}\ \mathit{code}\ \mathit{input} = \mathit{output} \\
&\quad \textbf{where} \\
&\qquad (\_, \_, \_, \_, \_, \mathit{output}) = \mathit{interpret}\ (\mathit{mem}, 0, \mathit{code}, [\,], \mathit{input}, [\,]) \\
&\qquad \mathit{mem} = \mathit{take}\ 32768\ (\mathit{repeat}\ 0)
\end{aligned}
$$

It accepts a code string and some input, and returns the output generated by running the interpreter. The *output* variable is bound to the last element of the interpreter state tuple returned from applying the *interpret* function to a start configuration. Note how all other elements of the final state are ignored.

The start configuration is assembled from an initial memory and pointer, the code, an empty list of continuations, the input, and an initially empty output list. The memory is initialised to 32 kB filled with zeros. The *repeat* function returns an infinite list containing elements with the same value, which is passed to the function. Since Haskell is a lazy language where all expressions are not evaluated until their results are actually needed, taking the first 32,768 elements from this infinite list will not exhaust memory.

Now it is possible to run *bf* programs in a simple fashion, e. g., like this: `run "+++." ""`. Unfortunately, *bf* programs that do something useful, such as printing a hello world message, tend to be lengthy. It is favourable to invoke the interpreter and have it run the contents of a file. The function required to achieve this goal can look like this:

$$
\begin{aligned}
&\mathit{runFile} :: \mathit{FilePath} \to [\mathit{Char}] \to \mathit{IO}\ [\mathit{Char}] \\
&\mathit{runFile}\ \mathit{file}\ \mathit{input} = \textbf{do} \\
&\quad \mathit{code} \leftarrow \mathit{readFile}\ \mathit{file} \\
&\quad \mathit{return}\ (\mathit{run}\ \mathit{code}\ \mathit{input})
\end{aligned}
$$

It accepts a *FilePath* (a string) and the input to be passed to the program and returns a value of the type *IO [Char]*. This means that the function has I/O side effects (*IO* is actually a monad, and the result of *runFile* is a list of characters wrapped in the *IO* monad).

The function first extracts all contents from the passed file and stores them in *code* as a string. It then applies the aforementioned *run* function to the code and input and returns the result.

5 Standalone Binaries

To be able to create a native executable of the *bf* interpreter, a main function is required—not unlike in a C program. Here it is:

$$
\begin{aligned}
&\mathit{main} = \textbf{do} \\
&\quad \mathit{file}\text{:}\ \_ \leftarrow \mathit{getArgs}
\end{aligned}
$$

code ← *readFile file*
interact (*run code*)

It retrieves all command line arguments using *getArgs* and discards all but the first element of the resulting list, binding it to *file*. The code to be executed is then read from the file.

Next, the *interact* function is applied. Note how *run* is not applied to code and input, but to code only. Again, currying is used here: internally, *interact* passes standard input as the next argument to the partially applied *run* function. The result of *run* is passed to standard output.

Creating a standalone executable from the source file of this PDF (recall that literate programming is being used here) is as easy as this: `ghc --make brainfuck.lhs`.