The design and implementation of awib, a brainfuck compiler written in brainfuck

Mats Linander
matslina@kth.se

January 13, 2008

Abstract

We have, within the constraints of the course *DD2464 - Bigger Advanced, Individual Course in Computer Science*, designed and implemented *awib*, a brainfuck compiler entirely written in brainfuck.

This document introduces brainfuck and describes our efforts.

Contents

1 Preface

An *esoteric* programming language is a language not only ill-suited for serious software development, but actually designed with this characteristic in mind. Rarely intended for real-world usage, the esoteric language is constructed as a joke, an entertaining puzzle or a proof of concept.

Due to it's compact notation, the minimal instruction set and a very simplistic data model, the *brainfuck* programming language, as the profane name suggests, can definitely be labeled esoteric.

We have designed and implemented *awib*, a brainfuck compiler entirely written in brainfuck. The conceptually simple task of brainfuck compilation is less than trivial when programmed in this fascinating language.

This document describes our efforts. We start off by giving an introduction to the programming language and discuss some common portability problems. We then move on to our development goals and discuss the high level design of awib. After this follow sections on the project's three main components. Finally, we evaluate our results and give an performance comparison against some other compilers and interpreters.

2 Brainfuck

2.1 Background

The *Turing machine* is a hugely influential computational model. Devised in the 1930's by mathematician and early computer scientist Alan Turing, the model has proven to accurately capture the notion of the modern computer and its computational capabilities.

According to the Church–Turing thesis, any problem solvable in a reasonable computational model can also be solved by a turing machine. A computational model equally powerful to the Turing machine is said to be *Turing complete*. Sufficiently expressive programming languages can also be said to be Turing complete. The criterion is that given any problem solvable by Turing machine, an algorithm for solving the same problem can be written in the language.

In reality, no Turing complete machine or language implementation can exist. This is due to the Turing machine being equipped with an infinitely large memory, which is typically hard to come by in the real world. The term is however loosely applied when lack of memory is the only missing feature of a machine or a language.

The brainfuck programming language was created in 1993 by computer enthusiast Urban Müller. It is said that Müllers goal was to design a Turing complete language for which a minimal compiler for the Amiga OS could be constructed[1]. While the Turing completeness of brainfuck may be intuitively obvious to some, there are several proofs available. Frans Faase's arguments [2] appear to be among the earliest. Also worth noting is Daniel B. Cristofani's proof by implementation [3], where a universal Turing machine is implemented in brainfuck.

`>`	Move the pointer a single cell to the right
`<`	Move the pointer a single cell to the left
`+`	Increase the integer in the current cell
`-`	Decrease the integer in the current cell
`,`	Read input into the current cell from the input stream
`.`	Output the current cell to the output stream
`[code]`	While the current cell is non-zero, execute `code`. Here, `code` is a syntactically correct sequence of instructions.

Table 1: The brainfuck instruction set

2.2 Data model

Clearly inspired by the Turing machine, the brainfuck data model consists of a *memory area* and a *pointer*.

The memory area is a sequence of cells, each containing an integer in the range $[0, 255]$. The pointer is a reference to one of the cells. While we often allow ourselves to consider the memory area infinitely large, it is always finite in practice. We often apply the directional concepts of left and right on the memory area, with the first cell in the sequence being considered the leftmost.

Throughout the execution of a brainfuck program, only the cell currently referenced, or *pointed at*, by the pointer can be directly accessed. To access other cells, the pointer must be moved. Initially when a program is run, every cell in the memory area holds 0 and the pointer points at the leftmost cell.

In addition to the memory area and the pointer, a brainfuck program has access to an *input stream* and an *output stream* through which data can be read and written. In a UNIX-environment, the input stream is usually the standard input and the output stream the standard output. However, the details of where and how input/output is read/written, is fully dependent on the implementation.

2.3 Instruction set

A brainfuck program is a finite sequence of brainfuck instructions. The instruction set, listed in table 1, manipulates the pointer and the cell currently pointed at. With the exception of the two control flow instructions, `[` and `]`, instructions are executed in the order that they appear in the program, one after the other. For a sequence of instructions to be syntactically correct, all occurrences of `[` and `]` must be well-balanced. Any non-brainfuck character in a program is considered a comment and is ignored.

Attempting to move the pointer beyond the available memory, in either direction, results in undefined behaviour. Cells wrap, in the sense that increasing/decreasing a cell beyond its maximal/minimal value produces the minimal/maximal value.

2.4 Portability

Unfortunately, the original language specification, as written by Urban Müller, was far from exhaustive and many important details were left out. Müller's original implementation[4] has been given semi-canonical status and is by many

considered to be a reference implementation. However, Müller actually provided two implementations, an interpreter and a compiler, which differ slightly in some key aspects. As a result, different implementations, especially when written by inexperienced brainfuck developers, can have wildly different behaviour.

2.4.1 EOF-behaviour

Almost every brainfuck implementation, including Müller's, use some method of signaling that an end-of-file (EOF) condition has been raised by the underlying hardware or operating system. The EOF condition indicates that the current input stream has been closed and can no longer be read from. While simply waiting indefinitely for further input may seem semantically sane in these cases, it is rarely a viable option. The ability to detect EOF is critical in many applications, and especially so when reading input from a file system.

As mentioned previously, the two implementations provided by Müller differed in some ways and most significantly so in their EOF-behaviour. If the input instruction '`,`' was executed after EOF had been reached, the interpreter wrote -1 to the current cell, while the compiler wrote nothing at all. In addition to these two behaviours, many implementors choose to write 0 on EOF.

All approaches has their advantages and disadvantages. Writing 0 on EOF tends to result in pleasantly compact code. For instance, the program

```
,[.,]
```

will, assuming 0 on EOF, output it's input until EOF is reached. If we rewrite the program for -1 on EOF, we reach the far less elegant

```
,+[-.,+]
```

Finally, the no-change on EOF version would read

```
,[.[-],]
```

where the loop `[-]` serves to set the cell to 0, regardless of what was read.

A key advantage of the -1 behaviour is exposed in implementations featuring a cell size larger than 8-bit, as these allow a program to differentiate between, on one hand, -1 being read as a result of EOF, and on the other, $255 = -1 \pmod{256}$ being read as a result of binary non-ascii data in the input.

We tend to favour the no-change EOF-behaviour, as this effectively allows the developer to decide on an EOF-indicating number (as in: `+++,---[code]`, where `code` will be executed unless 3 was read or EOF was reached).

2.4.2 Cell size

In the original implementations, cells were 8-bit integers. Thanks to two's complement representation of negative numbers, signedness of the cells lacked importance. As one might expect, arithmetic was performed modulo 256.

For various reasons, later implementations have come to deviate from this behaviour and cells holding 16-, 32- and even 64-bit integers are far from uncommon. In addition to the previously mentioned advantage when using -1 to indicate EOF, these cell sizes enable developers to perform arithmetic with fairly large integers, without having to go through the cumbersome process of themselves implementing the arithmetic on top of 8-bit cells.

To illustrate one of the classic portability issues arising from varying cell size, consider the following program for converting lower-case characters into upper-case.

```
++++++++[->++++<],
>[-<->]<.[-]
++++++++++.
```

The program reads a single byte, subtracts 32 from the byte (thereby moving from lower- to upper-case ascii), outputs it, clears the cell and finally outputs the ascii code for a newline (10).

What we're interested in is the clear loop (`[-]`), which, by repeatedly subtracting 1 from a cell until it is 0, can effectively be seen as a single instruction for setting a cell to 0. If the read byte is 31, the cell will hold -1 when entering the clear loop. In an 8-bit environment the program works as expected and terminates quickly, as a mere 255 iterations are sufficient for the clear loop to terminate. Consider the same situation in a 64-bit environment, where $2^{64} - 1$ iterations are required. On most computers, the time required for this loop to terminate is far from acceptable.

While the program above still functions correctly and eventually terminates in a 64-bit environment, this is not always the case. For instance, 8-bit centric developers often implement 16-bit arithmetic by relying on 255 incremented becoming 0, which will fail miserably in any other cell size.

2.4.3 Memory size

The memory area is finite in size. What remains for implementors to decide is just *how* finite it is. The original specification clearly states that the area must span 30000 cells or more. More is generally better, but some implementors go for less. Naturally, portability issues ensue.

2.4.4 General oddness

Finally, some implementations exhibit behaviour which is just plain odd and can hardly ever be seen as beneficial.

The most disturbing example of this may be implementations that forbid integer overflow, i.e. implementations not supporting integer wrap-around when adding/subtracting to/from the maximal/minimal integer value. Our guess is that this type of behaviour typically stems from the implementor using some high-level language or system in which explicit modulo-statements are required for integer overflow not to generate an error.

2.5 Comments in brainfuck code

Since all non-brainfuck characters in a program are considered to be comments, one is free to use any commenting style that excludes the 8 brainfuck instructions. We generally place comments on a separate line and prepend a hash-character ('#') to make it stand out.

To further clarify what goes on in a program, we often insert pre- and post-conditions for pieces of code. For this purpose, we have developed a very simple notation, allowing us to roughly document what the memory area looks like and

where the pointer points. Previous attempts at formally defining a notation for this purpose have all ended badly. It usually turns out that the burden of strictly adhering to a particular notation do more damage than the notation do good. We therefore both use and define the notation in a hand-waving manner.

```
% 0 X (bcode) *1  (where X is the read byte)
->>>+
% 0 X (bcode) 3(0) *1
```

In the example above, the two lines starting with a percent sign (`%`) are pre- and post-conditions for the small piece of code inbetween. The first comment indicates that the cell currently pointed at holds 1, that there is a block of cells called `bcode` at the left (the structure of which is hopefully described elsewhere) and beyond that block lies at least two cells, of which one holds 0 and the other holds some value `X` which has previously been read as input. The second comment indicates that the pointer has moved three cells to the right, that the current cell holds 1 and that the three cells between `bcode` and the current cell all hold 0.

We would like to stress that this is non-standard notation and not in any way part of the brainfuck language.

3 Objective

Our intention is to write a capable and usable brainfuck compiler. We will not settle for the mere fundamentals of brainfuck compilation, but have decided on a more ambitious set of goals. We truly intend for awib to be a real and viable alternative for brainfuck developers.

Portability The compiler should itself be highly portable and should run correctly in all major dialects of brainfuck. To be specific, the code should run under all three common EOF-behaviours and in any cell-size greater than or equal to the 8-bit cells of the original implementation.

Cross-compilation The compiler must support multiple target platforms. Initially, at least Linux on 386-architecture and the C programming language should be supported.

Extensibility The compiler should be structured and documented in such a way that other developers can get involved in and contribute to it. In particular, it should be as easy as possible to develop and add support for new target platforms.

Efficiency The compiler shall, as far as possible, generate fast and efficient code. It's output should be comparable in performance with that of other compilers and interpreters. As far as possible, this should be achieved independently of the target platform, hopefully enabling new target platforms to enjoy good performance with minimal effort.

Consistency Compiled code must execute in an environment that is well-defined and independent of the target platform. Users should be able to test their code against one of the backends, and trust that it will also function properly in the others.

We also intend to, at some point, release the compiler as Free Software [7] and further develop it under an Open Source collaborative model. This is a main motivation for the extensibility objective.

4 Design

In order to achieve the stated objectives, a solid and sound high level design is appropriate. Especially so given the potentially unreadable nature of brainfuck code.

We decided early on to separate the bulk of the source code into a frontend and a number of backends. Each backend would be responsible for one of the target platforms, while the frontend would hold as much as possible of the target independent code. In addition to providing us with a logical separation of code into components, this approach, which is very common in real world compiler design, proves to fit our design objectives quite well.

Communication between frontend and backends should be in the form of an *internal representation* (IR) code. The frontend reads a brainfuck program as input and generates IR code. The IR code is then passed along to one of the backends, which moves from the IR to target specific code and outputs the compiled program.

In the remainder of this section, we discuss our design in the light of the objectives described in section 3.

4.1 Portability

The portability requirement is not immediately affected by our high level design. It does, however, imply that all components must be connected by code into a single source file. The alternative would be to use intermediate files, UNIX pipes or some other system dependent device for frontend-to-backend communication. Clearly, this is not acceptable for an implementation claiming to be portable.

As far as the other portability issues are concerned, it is sufficient to pay close attention during the coding process and take care not to make any unreasonable assumptions on cell size or other properties that may vary between implementations. For instance, whenever input is read, we check for all three EOF behaviours and accept each one as an EOF indicator.

We have also written a simple but flexible brainfuck interpreter in C, which is capable of exhibiting the three common EOF behaviours and of using several different cell sizes. Every piece of code we write is tested against the 12 brainfuck dialects of this interpreter.

4.2 Cross-compilation and extensibility

Our division of the compiler into frontend and backends lends itself particularly well to constructing an extensible cross-compiler. All code specific to a particular target platform is concentrated into a single backend, allowing backend implementors to ignore everything but moving from IR to target specific code.

Another major focus point, as far as extensibility goes, is that of proper commenting. In any language, one must always carefully balance between providing too little and too much information. This is especially true in a language

Operation	Function
`ADD(x)`	Add `x` to the current cell
`INPUT`	Read input into current cell (bf: `,`)
`SUB(x)`	Subtract `x` from the current cell
`OUTPUT`	Output current cell (bf: `.`)
`LEFT(x)`	Move pointer `x` cells left
`RIGHT(x)`	Move pointer `x` cells right
`OPEN`	Open loop (bf: `[`)
`CLOSE`	Close loop (bf: `]`)
`CLEAR`	Set current cell to 0

Table 2: The awib IR

like brainfuck, where the concept of "self documenting code" lacks meaning. In addition to extensibility, the closely related concept of maintainability, is very much a product of how well we comment the code.

When released to the public, the project will be available both as a single file of source code, which is the common way of distributing brainfuck programs, and as a compressed source tree package aimed at developers and advanced users. This package will include documentation written from a developer perspective, detailing each individual component in wider strokes than the source code comments do.

We have done our best to make the code readable, both through source code comments and additional documentation. The frontend-backend design gives us confidence when it comes to extensibility. In spite of this, since large scale brainfuck development isn't a particularly well studied topic, we believe that only time can truly tell whether we have succeeded or not.

4.3 Performance

To achieve ease of backend implementation without sacrificing performance of compiled programs, we plan to focus performance enhancing code to the frontend. We believe that with a well chosen IR, backends should be able to settle for simple operation-for-operation code expansion, and still produce well performing code.

We have settled on an IR (table 2) that is essentially an extension of the brainfuck instruction set and, as such, operates in the brainfuck data model. In implementation, each operation is an integer pair representing the operation and (in some cases) the argument. Additional details are available in the developer documentation.

There is a clear one-to-one correspondence between the brainfuck instruction set and the first 8 IR operations (with the argument `x`=1 where needed). The IR operation `CLEAR` corresponds to the common clear-loop construct `[-]`. Naturally, we will not settle for this trivial correspondence between brainfuck and IR, but instead do our best to fully exploit the expressiveness of the IR.

We expect and hope to extend the IR in the future. In particular, we are keen to add something along the lines of an operation `ADDLEFT(x)`, that adds the value of the current cell to the cell `x` steps away on the lefthand side. This would, in combination with the `CLEAR` operation, allow us to efficiently imple-

ment common copy constructs like `[->+>>++<<<]`.

4.4 Consistency

A critical feature of a cross-platform compiler is to guarantee that compiled code behaves in a well defined and consistent manner, regardless of the chosen target platform. Therefore, we need to settle on a single definition of the brainfuck language and ensure that all backends produce programs that adhere to this standard. Fortunately, this is simple enough:

Cell size Cells are 8-bit wrapping integers

EOF behaviour Executing the input instruction (`,`) after EOF results the current cell being left as is.

Memory size At least $2^{16} - 1 = 65535$ cells are available. Moving the pointer beyond the available memory results in undefined behaviour.

To catch consistency problems, and for other purposes, we have created a catalogue of tests which can be run in an automated fashion. Each test consists of a piece of brainfuck source code and an input-output pair. If the code produced by a backend fails to produce the correct output, given the corresponding input, the test fails. Also worth noting is that the backends are tested through the interpreter mentioned in section 4.1, thereby further ensuring that portability issues will not affect consistency.

Still, the automated tests can only be seen as a complement to backend developers paying close attention to their work. The most important measure to ensure backend consistency may well lie in properly highlighting the issues importance in the developer documentation.

5 Components

5.1 Frontend

The frontend operates in three distinct phases: target identification, IR generation and syntax verification.

5.1.1 Target identification

The default target platform is set by the first few instructions of the awib source code and a confident user can easily modify this by hand. Many users will however prefer to specify the target at runtime. Especially so when using a compiled versions of awib. For this reason, a large part of the frontend is dedicated to target identification.

The convention we have settled on is for users to prepend a string to the source code they wish to compile. The string should be on the format `@target`, where `target` is the name of one of the available target platforms. End-of-file, whitespace and brainfuck instructions are all considered to terminate the string. As for the exact position of the target specification, we only demand that it lies prior to the actual brainfuck code.

5.1.2 IR generation

In the second phase, the frontend reads brainfuck source code and compiles it into IR code. In addition to performing the trivial brainfuck-to-IR translation (section 4.3), a number of optimizations are performed:

Cancellation Sequences of mutually cancelling instructions are reduced. This applies to adjacent occurrences of `+` and `-` as well as `>` and `<`.

Contraction A sequence of n subtractions (`-`) is compiled into the single IR operation `SUB(n)`. The same applies to `+`, `>` and `<`.

Dead-code elimination When a `[` is situated at the very beginning of a program, or immediately after a `]`, the loop the `[` opens will never be entered. These loops are removed entirely from the code.

Quick clear The common clear-loop (`[-]`) is replaced with the single IR operation `CLEAR`.

While we gladly apply the term "optimization" to these transformations, one could argue that they are far too trivial to do so. Considering that they are implemented in brainfuck, improve performance greatly and are commonly ignored by brainfuck implementors, we feel that the term is in fact well deserved.

5.1.3 Syntax verification

In the third and final phase of the frontend, the IR is verified to be syntactically correct. While this may sound impressive, it is simply a matter of verifying that all loops (bf: `[` and `]`, IR: `OPEN` and `CLOSE`) are well balanced. If the code is not correct, an error message is output and a flag is set to indicate that the IR should not be passed along to the backend.

5.2 The lang_c backend

The lang_c backend compiles awib IR code into the widely supported C programming language. In addition to making awib an option for developers on nearly any platform, this backend, in combination with a good C compiler, turns out to produce very fast programs (see section 6).

Translating brainfuck to C code is as old as brainfuck itself, and was first done in Müller's original documentation. This method of describing the language is well suited for describing brainfuck to programmers already familiar with imperative/procedural programming. Table 3 holds the brainfuck to C translation given by Müller. Worth noting is that the practice of writing -1 on EOF, as discussed in section 2.4.1, is a direct result of the C function `getchar()` returning -1 on EOF.

The C language backend translates IR to C according to the pattern listed in table 4. Other than the difference in using direct instead of indexed addressing, our C code has an input instruction crafted to do no-change on EOF, while Müller's writes -1.

In implementation, the full C code isn't actually output for each of the IR operations. Instead, we output a large program header in which 9 preprocessor macros, one for each IR operation, are defined. Each macro is on the form

Brainfuck	C
+	`array[p]++;`
-	`array[p]--;`
>	`p++;`
<	`p--;`
[	`while(array[p]) {`
]	`}`
.	`putchar(array[p]);`
,	`array[p]=getchar();`

Table 3: Brainfuck to C translation table, as given by Müller.

IR	C
ADD(x)	`*p+=x;`
INPUT	`c=getchar();if(c>=0)*p=c;`
SUB(x)	`*p-=x;`
OUTPUT	`putchar(*p);`
LEFT(x)	`p-=x;`
RIGHT(x)	`p+=x;`
OPEN	`while(*p){`
CLOSE	`}`
CLEAR	`*p=0;`

Table 4: IR to C translation table.

`eP(x)`, where the integer `P` is the integer used internally for representing the corresponding IR operation, and `x` is the argument to the operation. The full macro definition is as follows.

```
#define e9(x) *p=0;
#define e8(x) }
#define e7(x) while(*p){
#define e6(x) p+=x;
#define e5(x) p-=x;
#define e4(x) putchar(*p);
#define e3(x) *p-=x;
#define e2(x) c=getchar();if(c>=0)*p=c;
#define e1(x) *p+=x;
```

This method allows the actual IR to C translation to be done in a very small piece of code, making the backend source much more readable without affecting performance of output programs.

5.3 The 386_linux backend

The 386_linux backend generates executable programs for Linux on i386. Just like the C backend, this is done by operation-for-operation expansion to target specific code (386 machine code in this case). The expansions are listed in table 5, but with 386 assembly language instead of machine code.

For this expansion to function properly, a number of conditions regarding register content must be upheld throughout execution. These, along with much

IR	386 assembly
`ADD(1)`	`inc byte [ecx]`
`ADD(x)`	`add byte [ecx],x`
`INPUT`	`mov eax,edi`
	`dec ebx`
	`int 0x80`
	`inc ebx`
`SUB(1)`	`dec byte [ecx]`
`SUB(x)`	`sub byte [ecx],x`
`OUTPUT`	`mov eax,esi`
	`int 0x80`
`LEFT(1)`	`dec ecx`
`LEFT(2)`	`sub ecx,ebp`
`LEFT(x)`	`sub ecx,byte x`
`RIGHT(1)`	`inc ecx`
`RIGHT(2)`	`add ecx,ebp`
`RIGHT(x)`	`add ecx,byte x`
`OPEN`	`cmp bh,[ecx]`
	`jz word Y`
`CLOSE`	`jmp word Z`
`CLEAR`	`mov [ecx],bh`

Table 5: IR to 386 assembly translation table

other useful information can be found in the developer documentation. In addition to compiled brainfuck code, the backend outputs ELF headers (ELF is the file format used for executable files in Linux) and a block of mandatory code that performs memory allocation and related tasks.

Since an in-depth discussion of 386 architecture is beyond the scope of this document, we will not go into detailed discussions of the individual expansions. Worth noting, however, is that the argument carrying operations each have a couple of different code expansions that vary depending on the argument. These operations are all very common and mostly occur with small arguments. Only using the code accepting any argument `x` would produce unnecessarily large programs, and since the 386 backend imposes a 2^{16} byte size limit on produced programs, we do what we can to produce compact code.

To calculate jump offsets for `OPEN` and `CLOSE` we use a stack. The backend parses the IR twice, calculating jump offsets in the first pass and outputting machine code in the second. In implementation, the stack is simply a sequence of 16-bit integers that are arranged to be easily traversable.

6 Performance evaluation

We have performed a series of tests to evaluate the performance of programs compiled with awib, both for the C language and the 386 Linux backend. For the C backend, we compiled the C code with gcc using optimization level O1. We compared awib's performance against three other brainfuck implementations.

The interpreter *bff4* [6], by Oleg Mazonka, is often mentioned as one of the fastest available. We have compiled it with the gcc compiler, using optimization

	bff4	bf	obfc	awib lang_c	awib 386_linux
`long.b`	11.93	15.15	1.06	1.69	3.55
`factor.b`	2.64	1.31	0.30	0.56	1.44

Table 6: Results of the performance evaluation. Runtimes in seconds.

level O2 and with the "linear loop optimization" of bff4 activated.

The compiler *bf*[1], by Brian Raiter, is a very compact compiler for Linux on 386. It is written in assembly language with small size as the primary goal.

Finally, *obfc*[5] is an optimizing compiler for Linux on 386, written in roughly 1000 lines of C code by Manuel Schiller. Unlike awib, obfc generates assembly language as intermediate format, so an external assembler program is required.

The programs we have run are `long.b`, used by the bff4 developer for performance evaluation, and the classic `factor.b` by Brian Raiter. The former performs some resource hungry dummy calculation and the latter is a program for factoring arbitrarily large integers. In our tests, we factored $2^{58} - 1 = 3 \cdot 59 \cdot 233 \cdot 1103 \cdot 2089 \cdot 3033169$.

As illustrated in table 6, awib performs quite well. As expected, the ambitious obfc project performs very well for all programs and is the only compiler to beat both awib backends for both test programs. We also note that the awib C backend, when combined with the optimization of gcc, produces very decent programs. The real surprise is how well Brian Raiter's bf fares on the factor program.

Raiter's compiler only performs the most basic translation from brainfuck to machine code and yet outperform the optimizing 386 backend of awib. After disassembly and analysis of bf's output, we are still not quite certain as to why. Awib produces far more compact code, compiling `factor.b` into a program of half the size and about one third the number of instructions as that of Raiter's compiler.

The most likely explanation lies in how the two compilers implement the loop construct. Awib performs a comparison at the beginning of the loop and unconditionally jumps back to this comparison at the end of the loop. Raiter, on the other hand, always jumps to the end of the loop and performs the comparison there. In situations when a number of nested loop are terminated with a sequence of adjacent `]`, our model results in two jumps and one comparison per loop, while Raiter's model only requires a single comparison per loop. Our model performs better on adjacent `[`, but these are far less common, both in `factor.b` and in brainfuck code in general.

7 The future

As mentioned once or twice before, we intend to keep on developing and extending awib. In addition to adding more target platforms (a 386 FreeBSD backend is nearly completed as of this writing), we believe that far better performance can be achieved. We especially hope to add IR operations `ADDLEFT(x)` and `ADDRIGHT(x)`, as mentioned in 4.3, to optimize copy loops.

Another useful feature would be to allow the size of the memory area to be specified when compiling a program. Ideally, this should be done in a manner similar to how the `@target` flag is used to choose a target platform.

Also, in light of the results of section 6, we will definitely have to consider adopting Raiter's loop model for the 386 backend. Another option is to have comparisons and conditional jumps both at the start and at the end of each loop. This later approach generates larger programs, but should improve performance. Further research is however required before any decision can be made.

References

[1] B. Raiter, "The Brainfuck Programming Language," `http://www.muppetlabs.com/~breadbox/bf/`; accessed October 20, 2007.

[2] F. Faase, "BF is Turing-complete," `http://www.iwriteiam.nl/Ha_bf_Turing.html`; accessed October 20, 2007.

[3] D. Cristofani, "a universal Turing machine," `http://www.hevanet.com/cristofd/brainfuck/utm.b`; accessed October 20, 2007.

[4] U. Müller, "240 byte compiler. Fun, with src. OS 2.0," `http://www.hevanet.com/cristofd/brainfuck/brainfuckorigdistro/brainfuck-2.readme`; accessed October 20, 2007.

[5] M. Schiller, "obfc homepage," `http://hinterbergen.de/mala/obfc/`; accessed October 24, 2007.

[6] O. Mazonka, "Fast Brainfuck interpreter bff4.c," `http://mozaika.com.au/oleg/brainf/`; accessed October 24, 2007.

[7] Free Software Foundation, "The Free Software Definition," `http://www.fsf.org/licensing/essays/free-sw.html`; accessed October 24, 2007.