Session: Software

2005-08-29 PM

Lhogho – The Real Logo Compiler

Pavel Boytchev

Independent researcher

Sofia, Bulgaria

pavel@elica.net

Abstract

This paper announces the beginning of the design and the implementation of a new Logo compiler - Lhogho. Discussed are the motivation for starting this adventure and the current results. The Lhogho compiler is a real compiler. It translates Logo programs into machine code. The first version, written in GCC, targets the Pentium family of processors under Linux and Windows operating systems.

Keywords

Lhogho, Logo, compiler

1. Motivation

Logo has a long history. Its roots can be traced a few decades back in the past, but still it is a live programming language - there are thousands of Logo users and new Logo implementations come forward regularly. There are 140 Logo dialects, and 20% of them are still active (Boytchev P, 2003). Nowadays Logo is considered one of the most dynamic languages. Many Logo environments are interactive – users enter commands for immediate execution, variables can be created at run-time and they can be accessed indirectly by composing their names on the fly, and finally, it is possible to build new commands programmatically.

Although all these features are proudly enumerated when the benefits of Logo are discussed, they are a real nightmare for the developers. It is believed that these Logo features come from the unified way of representation of data and programs. However, this is true only for the user. Internally each implementation uses special techniques to represent and to preprocess executable lists in order to increase performance. The properties of Logo make it ideal for interpreting but not for compiling. Many other languages are interpreted. Why do we need a Logo compiler? Do we need to compile Logo programs at all?

We may consider interpretation as a better alternative for beginners, but when they get more experienced, they often start to write bigger and more complex programs that need considerable time for execution. At this point performance plays a significant role in the decision whether to continue with Logo or to move to a "more professional" language.

Developers can adequately respond to any user request. Multiple turtles, modern GUIs, multimedia? Yes, there are Logo solutions for all these. What about performance? Well, Logo developers make their best to provide the highest possible performance of interpretation. Many efforts are spent in optimizing internal structures and algorithms, but programs still appear to run much slower than similar programs in other languages.

One of the key goals of starting to write a Logo compiler is to make it possible for Logo programs to run fast enough to encourage professional and soon-to-be professional programmers to consider Logo as a valid choice.

Logo gurus will argue that building a compiler will be at the cost of loosing the core benefits of Logo - its spiritual heart that makes it stand out of the crowd of available programming languages. If this were an acceptable price for having a compiler, then it would be not too difficult to write such a compiler. This is not the point, because more or less this is already a routine task for an experienced translator developer. What really does matter is how to make a compiler that supports all interpreter-only features. Now that's what turns the making of a Logo compiler into a real and irresistible challenge.

2. Evolutionary concepts

Writing a Logo compiler is hard. It is so hard, that there is no successful recent attempt to make such. The last know complete Logo compiler, ObjectLogo, appears to be non-supported (Digitool, 1999). This paper will not discuss the difficulties of writing a Logo compiler. Instead it will present one simple idea - the only way to make a Real Logo Compiler[1] is to make it like a living creature.

Any compiled Logo program should have the complete Logo functionality in itself. This is a reasonable way to provide all the dynamic features. When a binary file is generated, it should be transmittable and executable on another compatible machine without the compiler. If the program runs dynamically generated Logo commands and expressions, then it needs the

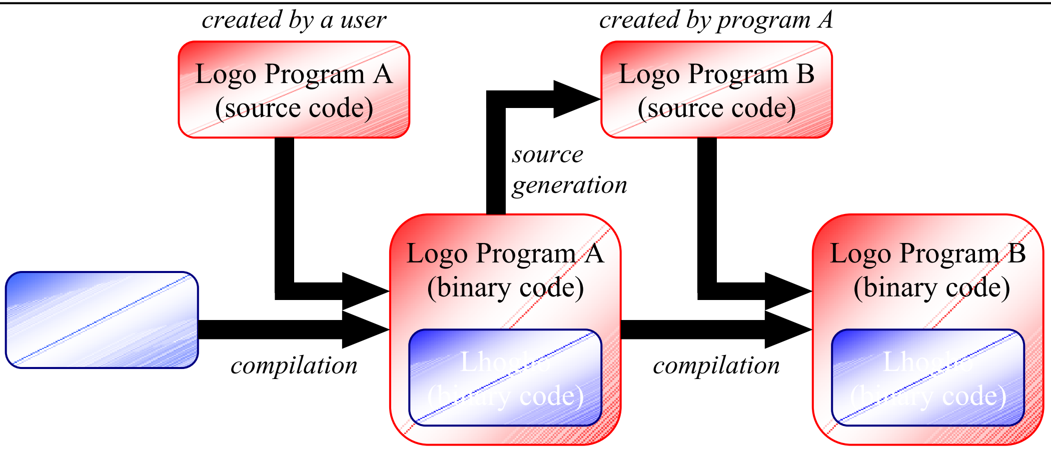

Figure 1. Logo program A is compiled by Lhogho into binary code, which contains the original compiler. Program A can generate the Logo source of another program – program B, which may be a modified version of A or an entirely new program. Program A may compile B into binary code which will inherit only the Lhogho code from A.

[1] The phrase Real Logo Compiler can be read in several ways, depending on the personal preferences of the reader. It could be (Real (Logo Compiler)) or ((Real Logo) Compiler). Both interpretations are equally correct, or at least are considered during compiler design. For infix aficionados we can provide yet another parsing: ((Real Compiler) for Logo).

possibility to compile. A way to resolve the problem is to embed the compiler (or a significant portion of it) inside every compiled by it program.

What is the relation with a living creature? Think of the Logo compiler as of DNA containing all Logo “chromosomes”. Compiling a source program into an executable program is like giving birth to a child – the child contains the characteristics of the parent, but is somewhat different – Figure 1. The difference is caused by the user's Logo program. The rest of the program, the Lhogho core, is the same. It is still not clear whether the third binary program in Figure 1 should contain the code of A or not. Idealy, the binary of B will contain the compiled code of B and the Lhogho core. If we want to have A-B-Lhogho, then A should “append” its source code to the code of B.

When the binary file is run it will do what is "commanded" in its DNA. The presence of genetic code from the parent will make the child inherit all features, including the ability to give birth to other Logo programs that can give birth to more Logo programs, and so on.

Following the same pattern of thoughts, the result of using conventional compilers and interpreters for "professional" languages corresponds to giving birth to a mule - creature that dies without issue.

Organic DNA is well compacted - millions of genetic bits are packed in a cell. This raises an important question about Lhogho. Will it be small enough to be reproduced in the compiled program? A typical modern Logo interpreter can be big up to several megabytes - a size that is quite unacceptable if the actual user program is small. The concern about compiler's size gives additional thrill to the problem.

3. Current status

3.1. Basic design details

The main design goal is to have a multitarget Logo compiler. That is why Lhogho is written in GCC (Free Software Foundation, 2005). This will make it easy to recompile the same sources for different hardware platforms and operating systems. Unfortunately, this is not the whole picture about multitargeting. Processors have different instruction sets and Lhogho should be aware of this. Even if processors are the same, the difference in operating systems might be crucial for the compiler.

The Lhogho sources can be compiled for the following targets: Fedora (Red Hat, 2005b), MS-Dos (Wikipedia, 2005), Cygwin (Red Hat, 2005a) and Windows (Microsoft, 2005). Only Pentium 386 compatible processors and above are supported.

3.2. Primitives

All primitives are embedded in the compiler. It does not rely on extensions like Elica libraries (Elica, 2004) and the UCBLogo macro definitions (Harvey B, 2004), because the evolutionary concept requires a single-file compiler. It is not only efficiency that dicatates this, but also transportability. If we compile a Logo program into a standalone binary, we would like to take it and execute it at another place. If the compiler uses external libraries, they must be copied too. That’s why there are plans to span Lhogho in the minimal possible number of files – ideally just 1 file, for the compiler itself.

Operators:
```
* / + - = < > <= >= <> and or
not all any
```
Selectors:
```
first butfirst bf last butlast
bl item
```
Constructors:
```
word list sentence se fput lput
```
Predicates:
```
word? wordp list? listp number?
numberp empty? emptyp equal?
equalp notequal? notequalp
before? beforep less? lessp
greater? greaterp lessequal?
lessequalp greaterequal?
greaterequalp member? memberp
```
Queries:
```
count char ascii lowercase
uppercase member parse
```
Arithmetic:
```
sum difference minus product
quotient remainder int round
sqrt power exp log10 ln
```
Trigonometric:
```
pi sin radsin cos radcos arctan
radarctan arctanxy radarctanxy
```
Sequences and randoms:
```
iseq rseq random rerandom
```
Transmitters:
```
print pr ? show type form format
```
Control structures:
```
to end true false if ifelse
repeat while until run forever
ignore
```
Variables:
```
" : make local thing
```

Figure 2. A list of reserved words already implemented in Lhogho. More commands and functions are being added continuously.

Transportability raises the question about localization of Lhogho. Namely, the translation of all primitives and error messages into another language. All texts, which Lhogho understands (primitives like `FIRST` and `BF`, and special words like `TRUE` and `FALSE`) as well as all texts that Lhogho generates (mainly error messages) are insulated in a separate source file. To localize Lhogho for a specific language it is needed to translate only the texts in this file and to recompile the compiler.

At the time of writing this paper, Lhogho supports more than 100 primitives including synonyms. Almost all of them are the same as in UCBLogo – see Figure 2. This is another important decision - to maintain a compatibility with the mainstream free Logo dialects UCBLogo and MSWLogo (Softronics, 2000) and functional compatibility with Elica. There are few exceptions only. For example, conditions in Lhogho can be either `"TRUE` or `"FALSE`. Lists are not allowed (as it is in UCBLogo's `WHILE`). Property lists, arrays and mutators are not implemented. Nested `TO..END` are allowed.

3.3. Performance

It is too early to measure performance and to compare Lhogho with other Logo dialects. Only several simple tests are done so far. The preliminary results show a performance around 10 times better than other Logos.

The achieved speed is mainly due to the translation to machine code. However, it is also a result of a redesigned memory handling and garbage collection. Although Lhogho is a true compiler, it supports native Logo datatypes – numbers, words and lists. Like in all other Logo implementations, variables are untyped. The internal garbage collection is distributed along the whole program execution by counting references to allocated memory. Atoms are freed on the fly instead of during a centralized GC process. The primary result of this approach is that no freezing is observed during execution.

Atom allocation and deallocation is handled by Lhogho memory handler. It is optimized for frequent requests. Atom size is only 16 bytes and is fixed for all data types. When compared with other Logo implementations Lhogho tends to use less memory, so spills done by the OS appear only in extreme cases. Even when programs use a huge amount of atoms (lists of several millions of atoms), no memory management delay is observed.

All performance advantages are observed when Logo program uses known in advance variables and commands. In tests where variables' names are generated in real time, performance is decreased. It gets even worse when a dynamic creation and execution of source code is placed inside a loop. In such cases, the compiler may become slower than some interpreters.

The following subsections provide some preliminary benchmarks. Tests are not designed to measure the performance of various Logo implementations, so they should not be used for any significant comparison. The idea it just to see how Lhogho scores against other Logos. Intensive and scientifically valid benchmark test will be done when Lhogho is complete.

All tests are done on the same computer (Pentium 4, 2.8GHz, 512MB RAM, Windows XP Home). The following Logo versions are used: UCBLogo 5.3, MSWLogo 6.5b, aUCBLogo 4.66 (Micheler A, 2005), Imagine 2.0 (Demo), Elica 5.4 and Lhogho.

For all test the performance of UCBLogo is accepted as a standard.

Arithmetic benchmark

This benchmark measures arithmetic calculations. The test program calculates $\Sigma^1/_n$ for $n\in[1..N]$. The value of N starts from 1,000 and for each test it increases by a factor of 10 until it reaches 1,000,000,000. Here is the program for N=1,000:

```
make "s 0
make "n 1

repeat 1000
[
  make "s :s + 1 / :n
  make "n :n + 1
]
```

The results (see Table 1) clearly show that the compiled code is much faster than interpretated. Note that in all test the time is rounded towards the nearest integer second, and the time for Lhogho contains compiler loading, compilation and execution.

Table 1. Arithmetic benchmark

	Elica	MSWLogo	UCBLogo	aUCBLogo	Imagine	Lhogho
N = 10^3	–	–	–	–	–	–
N = 10^4	2	–	–	–	–	–
N = 10^5	22	3	1	1	1	–
N = 10^6	224	26	7	7	7	1
N = 10^7	–	261	157	69	66	6
N = 10^8	–	–	–	682	599	59
N = 10^9	–	–	–	–	–	587
	14.3 times slower	**1.7** times slower	**1.0** standard	**2.3** times faster	**2.6** times faster	**26.8** times faster

Values, marked with the minus sign are either too fast or too slow. Only results from 1 to 1,000 seconds are shown.

List processing and memory managing

This benchmark measures memory allocation and deallocation by creating a long list of numbers and then reversing the list. The number of elements N starts from 1,000 and goes up to 10,000,000. The Logo program from N=1,000 is:

```
make "n 1000
make "a [ ]
repeat :n
[
  make "a fput :n :a
  make "n :n-1
]
make "b [ ]
while not empty? :a
[
  make "b fput first :a :b
  make "a bf :a
]
```

Only results from 1 to 120 seconds are shown. The time limit is set to 2 minutes, because most Logo interpreters use too much memory for lists of 1 million elements and the operating system starts to swap memory. Due to the optimized memory manager and the small atom size, Lhogho is the only Logo that created and reversed a list of 10 million elements in less than 10 seconds.

Table 2. List processing and memory managing benchmark

	Elica	MSWLogo	Imagine	UCBLogo	aUCBLogo	Lhogho
N = 10^3	7	–	–	–	–	–
N = 10^4	–	1	–	–	–	–
N = 10^5	–	12	2	2	1	–
N = 10^6	–	115	18	18	13	1
N = 10^7	–	–	–	–	–	8
	381.2 times slower	**6.4** times slower	**1.2** times slower	**1.0** standard	**1.4** times faster	**22.6** times faster

Indirect access

The last benchmark is for a program that accesses a variable through an expression. Indirect access is used for both reading the value of a variable and changing its value. The program calculates $\Sigma^1/_n$ but distributes the individual members of the series into three variables (selection is done in real time). Here is the program for N=1,000:

```
make "a 0
make "b 0
make "c 0
make "v "abc
make "n 0
repeat 1000
[
  make "n :n + 1
  make "w first :v
  make "v word bf :v :w
  make :w (thing :w) + 1 / :n
]
```

The expectation is that a simple compiler that does not analyze the dataflow of a program could not use optimized access to variables which names are not known at compile time. For such cases Lhogho inserts machine code instructions which scan the names of all accessible variables (the binary program contains not only instructions but some information about the names of the variables and their run-time locations in memory). The following table confirms that indirect access decreases the greatest compiler advantage – performance, but on the other hand it proves that typical Logo features can be implemented in a compiler.

Table 3. Indirect access benchmark

	Elica	Imagine	MSWLogo	UCBLogo	aUCBLogo	Lhogho
$N = 10^3$	1	–	–	–	–	–
$N = 10^4$	4	1	1	–	–	–
$N = 10^5$	45	6	6	4	2	1
$N = 10^6$	459	54	52	35	16	10
$N = 10^7$	-	542	535	375	163	100
	12.2 times slower	**1.4** times slower	**1.4** times slower	**1.0** standard	**2.3** times faster	**3.8** times faster

The results from this benchmark show that Lhogho is only few times faster than interpreted Logos. It is expected that if a program often generates and executes whole Logo instructions and program fragments, the performance of some interpreters may become comparable to the performance of the compiler.

Lhogho supports the `RUN` command, so functionally there is no need to support indirect access to variables as a special feature. However, Lhogho distinguishes between these two cases, because indirect access can be precompiled directly into binary code and it does not require real-time compilation, which is much slower.

3.4. Other functions

The current version of Lhogho compiles a complete Logo program into computer memory and executes it from there. The compiler has several options, which control its behaviour. It can insert additional machine instructions that trace the runtime execution flow; i.e. what functions are called and what are the values of their parameters. Another option is to output the assembly language mnemonics as text. For debug purposes the compiler may embed code to trace memory activities and detect memory leaks.

The current implementation of the compiler is not interactive, because interactivity should be delivered by the Lhogho environment. Thus, Lhogho is just a command-line compiler, which compiles a source file.

4. Future work

The development of Lhogho will pass through three phases until it reaches it first release.

- Phase 1 – support of all control structures and core Logo commands and functions. This includes first-class functions; functions for processing numbers, lists and words; conditional execution; definition of user routines.
- Phase 2 – support of graphical commands and functions (like turtle graphics), file access, etc. When phase 2 is completed, the compiler will be functionally equivalent to UCBLogo.
- Phase 3 – support for advanced graphics (3D modelling and animation), graphical user interface, multimedia, etc. At this phase, the compiler is supposed to overpower MSWLogo, aUCBLogo and Elica.

The first phase is expected to complete by the end of year 2005. The timeline for the other phases depends solely on the availability of external support.

The graphical part will be based on OpenGL (SGI, 2004). Like in Elica (and recently in aUCBLogo) OpenGL is a good mechanism to provide 3D modelling and animations. Because Lhogho is intended to be fast, it should be possible to write complex models, games, scenes, and animations. However, some time must be dedicated to design how all these things will be available to the users. Whether they should be able to control low-level aspects of the models or Lhogho will handle all technicalities.

Right now, it is not decided whether Lhogho should support multithreading, multiple turtles, and object oriented programming. The current design does not rule them out; so most likely Lhogho will support them.

Apart from developing the compiler itself, additional efforts must be dedicated to its environment. Advanced users might be happy with a command-line compiler, but to make Lhogho easier to be used by beginners, it will be good to provide a special integrated environment with suitable graphical user interface, which will organize the process of writing and executing programs. Although the compiler is not interactive, it may have a special interface with the environment to provide an interactive execution of commands.

5. References

Boytchev P (2003), *Logo Tree Project*,
`<http://www.elica.net/download/papers/LogoTreeProject.pdf>`

Digitool, Inc (1999), *Object Logo*,
`<http://www.digitool.com/ol-specs.html>`

Elica (2004), *Elica Logo*, `<http://www.elica.net>`

Free Software Foundation (2005), *GCC Home Page*, `<http://gcc.gnu.org>`

Harvey B (2004), *Berkeley Logo (UCBLogo)*,
`<http://www.cs.berkeley.edu/~bh/logo.html>`

Micheler A (2005), Andreas Micheler's Homepage – aUCBLogo's home,
`<http://www.physik.uni-augsburg.de/~micheler>`

Microsoft (2005), *Microsoft Windows Family Home Page*,
`<http://www. microsoft.com/windows/default.mspx>`

Red Hat, Inc. (2005a), *Cygwin*,
`<http://www.redhat.com/software/cygwin/index.html?id=home>`

Red Hat, Inc. (2005b), *Fedora Project*, `<http://fedora.redhat.com>`

SGI (2004), The Industry's Foundation for High Performance Graphics

http://www.opengl.org/

Softronics, Inc. (2000), *MSWLogo, An Educational programming language*
`<http://www.softronix.com/logo.html>`

Wikipedia (2005), *MS-DOS*, `<http://en.wikipedia.org/wiki/MS-DOS>`

(all links last visited in May 2005)