GNU Thunder

Alice Pascoe*

August 21, 2014

Let us go on to explain lightning and thunder, and further whirlwind, firewind, and thunderbolts: for the cause of them all is the same. Aristotle. *Meteorology.*

Abstract

We first identify some problems with some of the more mechanically fundamental software components in the GNU tool-chain, and then we show that though these are *prima facie* quite different problems, they actually have a common origin and therefore admit of a common solution. The particular components we treat of are the GNU portable assembler, `as`; the *GNU Compiler Collection*, which includes the GNU C compiler `gcc`; the GNU `autoconf` tools, the *GNU Guile* extension language and the just-in-time assembler *GNU lightning*.

1 Introduction

Firstly, let us make it clear that in using the word 'problems' we do not mean to be disparaging in any way. The problems we identify are problems that *we* now face, but which the designers of these tools never intended them to address. Now we need to explain precisely what these problems are, and why we think they *should* be addressed.

*e-mail: c/o `ian.a.n.grant@googlemail.com`.

2 The Problems

2.1 Boot-strapping

The `autoconf` tools are essentially a boot-strap loader for some part of the tool-chain. A source-code distribution is unpacked into a sub-directory on a file-system, and the autoconf-generated `configure` script will endeavour to determine how to invoke the necessary tools to compile/assemble/format that source into a usable set of objects, be they documentation, binary executables or compiled byte-codes of some kind, which can then be installed on the system. The autoconf-generated `configure` script is designed to require only a basic Bourne shell and a few standard Unix utilities such as `cp`, `cat` and the stream editor `sed` to run. All of these are typically C programs. All the GNU packages, including the autoconf package itself, use autoconf-generated configure scripts.

The GCC C compiler `gcc` is written in C. It therefore requires a working C compiler to have already been compiled. Of course, once compiled, the GCC C compiler may be able to compile itself on that system. The GNU C compiler compiles to assembler code and it requires the GNU assembler `as` to turn this into object code. If the C compiler source were distributed as assembler code, it would require only an assembler and a linker to compile, but then it would not be portable source code. A modern C compiler is a large and complex program. On the Intel i686 architecture the GCC 4.9 binary executables `cpp`, `gcc`, `cc1` and `as` total over 74MB, which is probably some 10 million machine instructions.

The GNU portable assembler `as` is written in C. It is capable of assembling text source code files into binary object code on every CPU architecture on which GNU software runs, including *Digital* PDP11 and *Zilog* Z80. There is no standard assembly language, and that implemented by GNU `as` is sometimes different to that of other assemblers on the same platform.

Both the C compiler and the assembler require a program called a *linker*, which takes binary object code in either files, or from archives called *libraries*, and resolves the symbolic references to create executable files. The format of executable files is platform-dependent, but many platforms use a variant of the Unix System V *Executable and Link Format*, ELF. The GNU toolchain includes a linker `ld` which is part of the coreutils package. The linker is written in C.

GNU lightning is a library implementing a just-in-time assembler which inter-

prets a generic assembly language and generates machine code for a variety of common CPU architectures. *Lightning* is also written in C, but it doesn't require `as`: it has its own "instruction encoder," implemented almost entirely by C pre-processor macros. Indeed, version 1 of *GNU lightning* does not contain even one line of compiled code, which shows that such a system could be implemented entirely syntactically, requiring no semantics more complex than purely formal textual substitution.

GNU Guile is, more or less, an implementation of R^{n}RS Scheme—a lexically-scoped dialect of *LISP*. The name *Guile* is an acronym of "GNU's Ubiquitous, Intelligent Language for Extension" and the plan, now over fifteen years old, is to make *Guile* the standard extension language for GNU projects. To this end, *Guile* is implemented as a dynamic object code library and distributed under the LGPL rather than the full GNU General Public Licence: the LGPL allows proprietary applications to use the *Guile* library unmodified, without requiring them to release their proprietary source code under the GPL. *Guile* is written in C and is oriented towards providing scripting and application-specific language processing for C programs.

Clearly then, the C programming language is the semantic basis, which logicians call *the meta-language*, of the GNU tool-chain. It is the semantics of C which define the operation of all the tools. But the *actual* operation of the 10 million or so machine instructions needed to compile the *Hello, World* program is far more than any one person could hope to comprehend, and so we have a security problem: if we cannot *actually know* what the C compiler does with our source code, then we cannot completely trust any of the tools, or any of the software we write using those tools. No, not even *Hello, World* is trust-worthy. See [5].

2.2 Abstracting Common Code

There are many programs which are quite sorely needed, but which are, it seems, too costly to develop. One such is a system to analyse and process C source code. Such a system would allow one to search, and also to edit, large source-code repositories in a semi or fully-automatic way. For example, during a security audit, one might wish to find every potential assignment reference to a certain variable. Tools such as the Unix `grep` and `sed` commands are sometimes used for this, but they are hopelessly inadequate when the criteria are not purely syntac-

tic, as in this example. And even when they are purely syntactic, the possibility of references affected using the pre-processor's *token-paste* operator `##` mean that even direct symbolic references can only be found by fully interpreting the C pre-processor directives. Clearly such a system would share a significant part of its function with the C compiler, and also `make` and the other tools such as the linker, whose input would need to be interpreted in order to fully determine the semantics of a C program consisting in more than one 'translation unit'.

Modifying the C compiler source code to perform this new function is not as simple as it might at first seem. The `gcc` front-end already doubles as a front-end for Objective-C and some other less well-known variatons on the C theme, and it is understandably oriented towards fast and efficient compilation of one translation unit at a time. Thus it uses an internal representation quite different from that which would be required by a source-code analyser, which must deal with many translation units, each appearing in the context of the whole system. Merely identifying what is the relevant fragment of the 740 MB of files which comprise the GCC version 4.9 source-code distribution would be challenging for anyone who has not recently been actively involved in its development. And separating that function in such a way that the two systems could share the neccessary common code would certainly require the cooperation of the `gcc` developers.

An alternative approach is to develop a mechanically interpretable *description* of the syntax and semantics of the relevant languages accepted by the various programs such as the C compiler and the other build tools. Such a formal specification would provide a basis upon which parsers and interpreters could be constructed automatically, resulting in programs, the specification of which would have been derived from a reinterpretation of those languages' grammar and formal semantics.

Of course the C compiler encodes a great deal of information about the syntax and semantics of the C language. Unfortunately, because it is written in C, that information is not in a machine-readable form. This is not solely because of the semantic problem we have described in the previous section. Even if we could trust the operation of the compiler binary, we could still not recover either the syntax or semantics from the compiler source by a mechanical means. This is simply because, to be able to interpret the compiler source, one must already know both the syntax and semantics of the language. If it is to be mechanically interpreted, the syntax and semantics of a language would need to have been

described in another language, the meta-language, and the process would only be worthwhile if that meta-language were in some sense simpler, or at least *better known,* than the language it was being used to describe.

Needless to say, this is just one example of potential code-reuse which does not yet happen. We have already mentioned that *GNU lightning* does not share any code with the GNU portable assembler, even though the functional overlap of these two tools is probably at least 75%, and would be even higher were lightning to Do The Right Thing, and produce relocatable code, with the option of generating object code or assembler source.

GNU Guile was *supposed* to become a sort of universal scripting/extension language for GNU projects, and it was designed to be able to interpret almost any other interpreted scripting language. Guile has now been under development for well over fifteen years, but so far it has not been used to interpret any language other than scheme in any application. The best foreign-language implementation Guile has is Emacs Lisp, but it is still not good enough to replace Emacs' Lisp interpreter.

The problem is this: implementating a good language interpreter takes years and years of hard work. If Guile's criterion for success is measured in terms of the number of good implementations of *other* languages it instantiates then it is most probably going to score zero.

Guile's real strength is not in its frontend, which is in fact weaker than most other interpreted languages around. It certainly has the slowest byte-code compiler we have ever used, taking some fifteen minutes to compile just one of its several hundred modules. Guile's sphere of competence is the fundamental engineering that is necessary if one is to effectively use a garbage-collected, high-level functional programming language from within any other compiled application. Its support for threads and futures with dynamic contexts is superb, and its conservative heap-tracing garbage-collector will co-habit with almost any memory management regime that would be used by a C program.

The only problem is *not all programs are C programs.*

Both *GNU Guile* and *GNU Lightning* make the same fundamental design assumption: that they are meant to provide language interpretation (which is a term that covers JIT compilation too, we think) for C programs. In a sense this is natural: what would be the point of providing language interpretation services

for a program which already implements a language interpeter? At least, this is a natural question when one thinks of languages as all being universal, in the sense that they attempt to provide a means to describe any and every type of computation any-one might want to carry out. But this is a naïve view of languages, because it ignores the most important thing about language: it is *the* tool for *abstraction*. So different languages abstract different aspects of different classes of problem. When we take this view of language, then we see that there must *necessarily* be a plurality of languages, and so it is obvious that there is a real need for a system which provides language interpretation services for other languages: such a common sub-system would allow all these different languages to interoperate smoothly, and it would allow people to use different languages for different parts of the same system, so that they could choose those languages which offer the right kind of abstraction for the task at hand. For example, one program could use Smalltalk to implement the controller of a model-view-controller user interface, and Guile-compiled *Cairo* "display PostScript" for graphics operations (the view), with ECMAScript and HTML for client-server interactions with a distributed database interface implemented in Scheme, and application-specific logic (the model) witten in O'Caml.

3 The Solution

> *We have now explained thunder and lightning and hurricane, and further firewinds, whirlwinds, and thunderbolts, and shown that they are all of them forms of the same thing and wherein they all differ.* Aristotle. *Meteorology.*

The attentive reader will by now have seen that the common theme in the problems of the previous two sub-sections is meta-language. And so she will not be surprised when we suggest that the common solution is the development of a meta-language which can be mechanically interpreted, with a fairly modest effort, by a program simple enough that its intensional semantics can be comprehended by one person after only a few weeks of consciencious study. This meta-language need not, and probably should not be, a so-called *Turing-universal* language. All that it is required to do is *translate* the formal descriptions of syntax and semantics into an operational language, which will of course need to be Turing-universal if it is to interpret the semantics of Turing-universal languages. But the formal syntax and semantics of any language are a finite, well-founded structure, so the translation can always be computed in finite time.

There are several obvious candidates for such an intermediate language. We have tried-and-tested implementations of bytecode interpreters such as those used by *Java* interpreters and *Objective CAML,* and there are also some simple interpreters of lambda calculus terms, such as those used in *Guile*, the *ML Kit*, *SML/NJ* and *Moscow ML*. But the representation we think might prove to be quite spectacularly successful is System **F**, described in *Proofs and Types* by Girard *et al.*[2].

The reason we prefer System **F** to any particular byte-code interpreter is that it is *entirely abstract.* The only primitive operation that is needed is an abstract substitution operator. This means that any specification written in **F** will be abstract, and free of any artefacts of the representation which could render interpretation problematic.

Some may raise grave doubts that *any* implementation of System **F** could in practice be made efficient enough to compile GCC, but we think that they will be pleasantly surprised. The key is the comment we made in the penultimate paragraph of the first sub-section: that *GNU lightning I* implements just-in-time compilation using only textual substitution carried out by the C pre-processor. If one were to substitute something akin to the GNU lightning machine-instruction primitives for the 'ghostly atoms' which only make their appearence as meta-language variables c_i in the System **F** semantics, then one would find System **F** entirely adequate to describe the compilation of primitive operations to some fairly efficient machine-code. Thus one could formally define a translation of System **F** terms to combinatory logic and then use the graph-reduction process[1] for call-by-need evaluation described by Paulson in [3].

The way this would work in practice is that a program would parse the System **F** specification, apply the necessary machine/ABI primitives to instantiate that abstract specification to some concrete representation, and then reduce it to normal form. The result could be written out in C, for example, using lightning calls to generate the machine code at run time. The programs produced would be both portable and efficient. In this boot-strapping process, the system C compiler is relegated to performing a function equivalent to a link-loader.

This could be the basis for an autoconf-like tool. Instead of the primitive shell-script syntax, the configuration process could then be described in terms of, say, Datalog[1] inferences from a database of facts, which could be supplied in a text

[1]Due to Wadsworth[6], according to Reynolds in [4]

file using a suitable special-purpose language, augmented by other facts derived from invoking, say, the system's own dynamic linker, and directly inspecting the contents of archives, or the C pre-processor, to read system header files, parsing and interpreting the resulting output, rather than inferring the results from the C compiler's exit code.

Then, instead of making unreliable and often incorrect deductions from the information in the results of empirical experiments conducted on the C compiler, `autoconf` will have direct access to intensional definitions of the facts about the platform ABI etc. This would make configuration of, e.g. cross compilation, far more practical: a cross-compiler build process cannot make such empirical experiments because, by definition, it doesn't run on the target system.

The role of the existing `autoconf` tools would only extend to what they already do very well: figuring out how to invoke the interpreters, compilers, linkers etc. that are needed to run programs. The remaining higher-level configuration would be done by the program `metaconf`, interpreting the formal specifications. So the ugly duckling will have grown up to become a beautiful swan: an expert system encoding the wealth of knowledge the Free Software community actually has, about how to do reliable, secure, cross-platform software development.

4 Abstracting Algorithms

What we have given so far are only specific *concrete examples* of the much more general idea, which is that to make progress in computing, we have to shift the focus of our efforts away from concrete extensional representations of algorithms as particular implementations in particular languages, towards abstract intensional representations of *pure algorithms,* free from the accidents of any particular implementation. We need to do this so that we can automate the generation of implementations of those algorithms in whatever particular concrete form we require, because these will be many and various. It is something every experienced programmer knows: writing computer programs is a repetitive, mechanical task. And repetitive mechanical tasks are difficult for people to get right, because they get bored. Then they cut corners. Then they make mistakes. And it is only a lack of the necessary tools for automatically composing source-code structures which prevents them from automating the chore of writing programs.

The only programs we need to write now are the tools for automatically composing programs. And the first programs those tools write will be the programs that implement those very tools. Then we will go on, and soon there won't be any trace whatsoever of human involvement in the writing of programs: all the programs we use will have originated from other programs.

That's how God invented the Chicken, you know. And She did such a good job of writing herself out of the *text* that now not even *The Pope* believes in Her!

So much for thunder and lightning. Aristotle. *Meteorology.*

References

[1] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). http://doi.ieeecomputersociety.org/10.1109/69.43410 (Because you couldn't afford the extortionate fee being charged for the right to read their paper.).

[2] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, Cambridge, 1989. http://www.paultaylor.eu/stable/prot.pdf.

[3] Lawrence C. Paulson. *Foundations of Functional Programming, Part IB Computer Science Tripos Lecture Notes*. Computer Laboratory, University of Cambridge, 2000. http://www.cl.cam.ac.uk/~lp15/papers/Notes/Founds-FP.pdf.

[4] John C. Reynolds. Definitional Interpreters Revisited. *Journal of Higher Order and Symbolic Computation*, 11(4):355–361, 1998.

[5] Ken Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 27(8), August 1984. http://www.ece.cmu.edu/~ganger/712.fall02/papers/p761-thompson.pdf.

[6] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, Programming Research Group, Oxford University, Oxford, England, September 1971.