Go 1.3+ Compiler Overhaul

golang.org/s/go13compiler

Russ Cox
December 2013

Abstract

The Go compiler today is written in C. It is time to move to Go.
[**Update**: This work was completed and presented at GopherCon. See "Go from C to Go".]

Background

The "gc" Go toolchain is derived from the Plan 9 compiler toolchain. The assemblers, C compilers, and linkers are adopted essentially unchanged, and the Go compilers (in cmd/gc, cmd/5g, cmd/6g, and cmd/8g) are new C programs that fit into the toolchain.

Writing the compiler in C had some important advantages over using Go at the start of the project, most prominent among them the fact that, at first, Go did not exist and so could not be used to write a compiler, and the fact that, once Go did exist, it often changed in significant, backwards-incompatible ways. Using C instead of Go avoided both the initial and ongoing bootstrapping problems. Today, however, Go does exist, and its definition is stable as of Go 1, so the problems of bootstrapping are greatly reduced.

As the bootstrapping problems have receded, other engineering concerns have arisen that make Go much more attractive than C for the compiler implementation. The concerns include:

- It is easier to write correct Go code than to write correct C code.
- It is easier to debug incorrect Go code than to debug incorrect C code.
- Work on a Go compiler necessarily requires a good understanding of Go. Implementing the compiler in C adds an unnecessary second requirement.
- Go makes parallel execution trivial compared to C.
- Go has better standard support than C for modularity, for automated rewriting, for unit testing, and for profiling.
- Go is much more fun to use than C.

For all these reasons, we believe it is time to switch to Go compilers written in Go.

Proposed Plan

We plan to translate the existing compilers from C to Go by writing and then applying an automatic translator. The conversion will proceed in phases, starting in Go 1.3 but continuing into future releases.

Phase 1. Develop and debug the translator. This can be done in parallel with ordinary development. In particular, it is fine for people to continue making changes to the C version of the compiler during this phase. The translator is a fair amount of work, but we are confident that we can build one that works for the specific case of translating the compilers. There are many corners of C that have no direct translation into Go; macros, unions, and bit fields are probably highest on the list. Fortunately (but not coincidentally), those features are rarely used, if at all, in the code being translated. Pointer arithmetic and arrays are also some work to translate, but even those are rare in the compiler, which primarily operates on trees and linked lists. The translator will preserve the comments and structure of the original C code, so the translation should be as readable as the current compiler.

Phase 2. Use the translator to convert the compilers from C to Go and delete the C copies. At this point we have transitioned to Go and still have a working compiler, but the compiler is still very much a C program. This *may* happen for Go 1.3, but that's pretty aggressive. It is more likely to happen for Go 1.4.

Phase 3. Use some tools, perhaps derived from gofix and the Go oracle to split the compiler into packages, cleaning up and documenting the code, and adding unit tests as appropriate. This phase turns the compiler into an idiomatic Go program. This is targeted for Go 1.4.

Phase 4a. Apply standard profiling and measurement techniques to understand and optimize the memory and CPU usage of the compiler. This may include introducing parallelization; if so, the race detector is likely to be a significant help. This is targeted for Go 1.4, but parts may slip to Go 1.5. Some basic profiling and optimization may be done earlier, in Phase 3.

Phase 4b. (Concurrent with Phase 4a.) With the compiler split into packages with clearly defined boundaries, it should be straightforward to introduce a new middle representation between the architecture-independent unordered tree (Node*s) and the architecture-dependent ordered list (Prog*s) used today. That representation, which should be architecture-independent but contain information about precise order of execution, can be used to introduce order-dependent but architecture-independent optimizations like elimination of redundant nil checks and bounds checks. It may be based on SSA and if so would certainly take advantage of the lessons learned from Alan Donovan's go.tools/ssa package.

Phase 5. Replace the front end with the latest (perhaps new) versions of go/parser and go/types. Robert Griesemer has discussed the possibility of designing new go/parser and go/types APIs at some point, based on experience with the current ones (and under new names, to preserve Go 1 compatibility). The work of connecting them to a compiler back end may help guide design of new APIs.

Bootstrapping

With a Go compiler written in Go, there must be a plan for bootstrapping from scratch. The rule we plan to adopt is that the Go 1.3 compiler must compile using Go 1.2, Go 1.4 must compile using Go 1.3, and so on. Then there is a clear path to generating current binaries: build the Go 1.2 toolchain (written in C), use it to build the Go 1.3 toolchain, and so on. There will be a shell script to do this; it will take CPU time but not human time. The bootstrapping only needs to be done once per machine; the Go 1.x binaries can be kept in a known location and reused each time all.bash is run during the development of Go 1.(x+1).

Obviously, this bootstrapping path scales poorly over time. Before too many releases have gone by, it may make sense to write a back end for the compiler that generates C code. The code need not be efficient or readable, just correct. That C version would be checked in, just as today we check in the y.tab.c file generated by yacc. The bootstrap sequence would invoke gcc on that C code to build a bootstrap compiler, and the bootstrap compiler would be used to build the real compiler. Like in the other scheme, the bootstrap compiler binary can be kept in a known location and reused (not rebuilt) each time all.bash is run.

Alternatives

There are a few alternatives that would be obvious approaches to consider, and so it is worth explaining why we have decided against them.

Write new compilers from scratch. The current compilers do have one very important property: they compile Go correctly (or at least correctly enough for nearly all current users). Despite Go's simplicity, there are many subtle cases in the optimizations and other rewrites performed by the compilers, and it would be foolish to throw away the 10 or so man-years of effort that have gone into them.

Translate the compiler manually. We have translated other, smaller C and C++ programs to Go manually. The process is tedious and therefore error-prone, and the mistakes can be very subtle and difficult to find. A mechanical translator will instead generate translations with consistent classes of errors, which should be easier to find, and it will not zone out during the tedious parts. The Go compilers are also significantly larger than anything we've converted: over 60,000 lines of C. Mechanical help will make the job much easier. As Dick Sites wrote in 1974, "I would rather write programs to help me write programs than write programs." Translating the compiler mechanically also makes it easier for development on the C originals to proceed unhindered until we are ready for the switch.

Translate just the back ends and connect to go/parser and go/types immediately. The data structures in the compiler that convey information from the front end to the back ends look nothing like the APIs presented by go/parser and go/types. Replacing the front end by those libraries would require writing code to convert from the go/parser and go/types data structures into the ones expected by the back ends, a very broad and error-prone undertaking. We do believe that it makes sense to use these packages, but it also makes sense to wait until the

compiler is structured more like a Go program, into documented sub-packages of its own with defined boundaries and unit tests.

Discard the current compilers and use gccgo (or go/parser + go/types + LLVM, or …). The current compilers are a large part of Go's flexibility. Tying development of Go to a comparatively larger code base like GCC or LLVM seems likely to hurt that flexibility. Also, GCC is a large C (now partly C++) program and LLVM a large C++ program. All the reasons listed above justifying a move away from the current compiler code apply as much or more to these code bases.

Long Term Use of C

Carried to completion, this plan still leaves the rest of the Plan 9 toolchain written in C. In the long term it would be nice to eliminate all C from the tree. This section speculates on how that might happen. It is not guaranteed to happen in this way or at all.

Package runtime. Most of the runtime is written in C, for many of the same reasons that the Go compiler is written in C. However, the runtime is much smaller than the compilers and it is already written in a mix of Go and C. It is plausible to convert the C to Go one piece at a time. The major pieces are the scheduler, the garbage collector, the hash map implementation, and the channel implementation. (The fine mixing of Go and C is possible here because the C is compiled with 6c, not gcc.)

C compilers. The Plan 9 C compilers are themselves written in C. If we remove all the C from Go package implementations (in particular, package runtime), we can remove these compilers: "go tool 6c" and so on would be no more, and .c files in Go package directory sources would no longer be supported. We would need to announce these plans early, so that external packages written partly in C have time to remove their uses. (Cgo, which uses gcc instead of 6c, would remain as a way to write parts of a package in C.) The Go 1 compatibility document excludes changes to the toolchain; deleting the C compilers is permitted.

Assemblers. The Plan 9 assemblers are also written in C. However, the assembler is little more than a simple parser coupled with a serialization of the parse tree. That could easily be translated to Go, either automatically or by hand.

Linkers. The Plan 9 linkers are also written in C. Recent work has moved most of the linker in into the compilers, and there is already a plan to rewrite what is left as a new, much simpler Go program. The part of the linker that has moved into the Go compiler will now need to be translated along with the rest of the compiler.

Libmach-based tools: nm, pack, addr2line, and objdump. Nm has already been rewritten in Go. Pack and addr2line can be rewritten any day. Objdump currently depends on libmach's disassemblers, but those should be straightforward to convert to go, whether mechanically or

manually, and at that point libmach itself can be deleted.