The Go frontend for GCC

Ian Lance Taylor
Google
iant@google.com

Abstract

A description of the Go language frontend for gcc. This is a new frontend which is a complete implementation of the new Go programming language. The frontend is currently some 50,000 lines of C++ code, and uses its own IR which is then converted to GENERIC. I describe the structure of the frontend and the IR, issues that arise when compiling the Go language, and issues with hooking up any frontend to the gcc middle-end.

1 Introduction

Go is a new programming language designed by Robert Griesemer, Rob Pike, and Ken Thompson, with major contributions by Russ Cox and the author. Go is being developed as an free software project hosted at `http://golang.org/`. There are currently two Go compilers. The first is based on the Plan 9 C compiler originally written by Ken Thompson; this compiler is known as the gc compiler. This paper describes the second, a frontend to gcc, generally known as gccgo.

A goal of the gccgo project is to be a typical gcc frontend. It is intended to require minimal changes to other parts of gcc. Like all gcc frontends, it generates assembly code which is then processed by ordinary unmodified assemblers and linkers. This approach is different from the less conventional gc compiler, which does significant code generation work at link time.

Gccgo is written in C++. As of this writing it is slightly more than 50,000 lines, including blank lines and comments.

2 Intermediate Representation

The intermediate representation, known as GOGO, is a collection of C++ classes. The initial form of GOGO closely mirrors the Go source code read from the input files. This is intended to make it easy to implement whatever analysis may seem to be desirable. After all the input files have been read, the GOGO structures are lowered to a simpler version, eliminating loop constructs, tuple assignments, and other complex statements. This version of GOGO is eventually converted to GENERIC and passed to gcc's middle-end.

The main structures in GOGO are as follows:

- `Gogo`. The IR for the entire input.
- `Statement`. A statement.
- `Expression`. An expression.
- `Type`. A type.
- `Block`. A block in the program—a list of statements.
- `Named_object`. A named object: a function, variable, type, constant, or package.

Each of these structures is a class in gccgo's source code. The `Statement`, `Expression` and `Type` classes are base clases with pure virtual functions. Base children of these classes, such as `Assignment_statement`, implement the virtual functions. For each child class, there is an enum value such as `STATEMENT_ASSIGNMENT`. The base classes have a `classification` method that returns the enum associated with the child class. Thus there are two different ways to use a pointer to an instance of the base class: by calling a virtual function, or by calling the `classification` method and taking the appropriate action.

Each class in Gogo implements a `traverse` method; the arguments vary depending on the class. Each `traverse` method takes an argument of type

Traverse. Traverse is a base class with virtual methods such as statement, expression, etc. The Traverse constructor takes a bitmask listing the elements of interest. Code that needs to traverse the tree, or a portion of it, creates a child class of Traverse that overrides the virtual functions that it needs to use, and then calls the appropriate traverse method. Calling the traverse method on the single Gogo object will traverse the entire tree.

This approach permits code to be grouped as appropriate. Operations can be implemented as a function of the class or the operation may handle all relevant classes directly. An example of the former approach is conversion to GENERIC, that all classes must implement differently. An example of the latter is the conversion of && and || expressions to if statements, for which all the required work can easily be done in a single function.

Memory usage is always a consideration for any compiler IR. The use of child classes in GOGO has the advantage that it is very natural to only allocate the required amount of space for each node. On the other hand, it adds a virtual function table pointer to each node. In effect the node classification is encoded twice: as an enum field and as a virtual function table pointer. It would be possible to replace the the virtual function table pointer with an explicit table indexed by the enum value, or by making classification a virtual method, at the cost of some ease of debugging.

3 Language

This paper does not provide a general description of the Go language. I will just touch on a few aspects of the language that are worth noting for their effects on the frontend.

Go is defined by an explicit specification that may be found at http://golang.org/doc/go_spec.html. It is not defined by an implementation. An explicit language specification was a requirement for implementing two different compilers, and implementing two compilers was a great help in clarifying the spec.

3.1 Parsing

Go is designed to be easy to parse. Gccgo uses a recursive descent parser that closely follows the grammar provided in the specification.

Go does not require names to be defined before they are used. A consequence of this is that it may not be immediately clear at parse time whether a particular token sequence is a type or an expression. For example, f(v) may be a call of the function f passing the variable v, or it may be a type conversion of the variable v to the type f. The type could even look like an expression, as in (*p)(v); that is either a call to the function to which the variable p points or a conversion to the pointer type *p.

Similarly, the sequence t.m may select the field m from a variable t of struct type, or it may name the method m of the type t.

Cases like these do not complicate the parser. The parser simply creates a IR node representing an undetermined result. After parsing is complete, all the names are defined (an undefined name is an error) and a lowering pass converts the undetermined nodes to the appropriate classification.

The only slightly complex parse in Go occurs in function definitions. When defining a function, parameters names may be all omitted or all present. When parameter names are present, several consecutive parameters may have the same type. Thus, these are both valid function definitions in Go:

```
func f1(t, t, t, t, t, t, t) { }
func f2(v, v, v, v, v, v, v int) { }
```

The first is only valid if t is a type, but the parser may have not yet seen the type definition. Gccgo parses this using arbitrary lookahead, up to the first point where a name is not immediately followed by a comma. This is the only place where arbitrary lookahead occurs in the gccgo parser. The gc compiler uses a different approach: it has a nonterminal for a comma-separated identifier list, which is then later lowered as appropriate.

3.2 Constants

Go constants are untyped and of arbitrary size (the language does permit an implementation to restrict the size). They only acquire a type when they are used outside of a constant expression, e.g., in an assignment. At the point of use, the value of the expression must fit in the type. Before then, there is no limit. E.g., this is valid Go:

```
var v = (1 << 100) >> 99
```

Fortunately gcc is already linked with the GMP, MPFR and MPC libraries, and they provide support for infinite precision constants. Gccgo represents constants using those libraries, and converts to types like `double_int` and `REAL_VALUE_TYPE` when generating GENERIC.

3.3 Symbol Names

Gccgo must mangle all externally visible symbol names.

- Every Go file is always contained within a package. The same name may be used in different packages. Therefore, mangling is required to distinguish them.
- Although Go does not support function overloading, it does support methods on arbitrary named types. Mangling is required for method functions.
- Gccgo must generate a type descriptor for all named types, all pointers to named types, and all unnamed types which are converted to interface types. The names of these type descriptors must be mangled.

The mangling is straightforward and will not be documented here. There is one issue that deserves comment. Go permits different packages to have the same name, so a simple mangling of the package name is insufficient. The gc compiler rewrites the symbol names at link time, but that technique is not available to gccgo. Instead, gccgo supports a `-fgo-prefix` option that may be used to set a unique prefix for the package being compiled. The option takes any string as an argument; a natural string to use would be the name under which the package will be installed.

3.4 Language Extension

Gccgo adds one language extension to the Go language: the ability to specify the assembler name of a function declaration. This uses syntax similar to what gcc supports for C.

```
func close(int) __asm__ ("close")
```

This may only be used for a function declaration, not a definition. The expectation is that the function will not be defined in Go. Any calls to the function will avoid the name mangling described above and simply result in a call to the name given in the `__asm__` declaration. This is used to make it easier for Go code to call C code directly.

4 Passes

The gccgo frontend is arranged as a series of passes over the input.

1. All the input files are lexed and parsed. The input files must all belong to the same package, and all the input files for a given package must be provided to the compiler at the same time. This generates the initial GOGO representation that closely matches the source files.
2. Gccgo walks through the set of predeclared identifiers: predeclared types like `int` and predeclared functions like `new`. For each predeclared identifier, gccgo checks for an undefined reference at the package level. If a reference is found, gccgo defines it as the predeclared identifier. These identifiers can not be declared before compilation, as Go permits them to be redefined at package scope, and the parser may see references to the package scope identifiers before they are defined.
3. Gccgo walks the whole tree looking for types that can have methods: named types, interface types, and struct types. Gccgo finalizes the set of methods for each type. This is where gccgo implements the promotion of methods from anonymous embedded fields to become methods for the enclosing type. Gccgo creates a stub method—a tiny function—for each promoted method where necessary. The stub method simply calls the method on the embedded field.
4. Gccgo lowers the parse tree. This pass walks the whole tree and performs several different operations. The goal of this pass is to simplify the tree to remove complex statements and looping constructs.
    - When statements include expressions with side-effects at the top level, gccgo changes

those expressions into assignments to temporary variables and places the new assignments before the statement. This implements Go's rules about evaluation order within a single statement. The expressions with side effects are function calls and send or receive expressions on a channel.

- Gccgo converts Assignment statements like `a += b` to `a = a + b`, which is safe after side effects have been moved.
- Gccgo converts `a++` to `a = a + 1`, and similarly for `a--`.
- Gccgo converts tuple assignments such as `a, b = c, d` into a series of single assignment statements using temporary variables.
- Gccgo converts special purpose tuple assignments, such as `v, ok = m[i]` into calls to runtime library functions.
- Gccgo converts `return` statements in functions with named result parameters into one or more assignments to the result parameters followed by a `return`. This permits `defer` closures to adjust the results after recovering from a panic.
- Gccgo converts `switch` statements with cases that are not constant or whose switch variable does not have integer type to a series of `if` statements.
- Gccgo converts type switch statements to a series of `if` statements.
- Gccgo lowers `for` statements to use `if` and `goto` statements. If there is a `range` clause, gccgo inserts calls to runtime library functions as needed.
- Gccgo converts references to unknown names to references to variables, functions, constants or types as appropriate. Gccgo issues an error if the name was never defined.
- Gccgo folds constant expressions. This must be done in the frontend because it must be done in infinite precision.
- Gccgo gives any uses of the predeclared constant `iota` a specific numeric value.
- Gccgo converts calls to the special predeclared functions `new` and `make` to special node types.
- Gccgo notes all functions that call the special predeclared function `recover`. This is not a lowering step, but the information is used in a later pass; see section 7.5.
- Gccgo converts cases where a `return` statement returns multiple results from a call, or where multiple results from a call are passed to another call, to use a series of temporary variables.
- Gccgo converts calls to functions with variable numbers of arguments to build a slice for the final argument.
- Gccgo looks up field and method names for references within structs. This can only be done after methods are finalized by the previous pass.
- Gccgo simplifies composite literals, removing key values for struct, array and slice literals.

5. Gccgo walks the IR looking for all types, and verifies that they are correct. This is where the compiler gives an error for variable array lengths, maps whose key type is a struct or array type, or named types that are defined in terms of themselves in ways that can not work. Go permits named types to refer themselves when the size does not matter, as in `type T *T` or `type T []T` or `type T func() T`. This pass detects case which are forbidden, such as `type T T`.

6. Gccgo stops at this point if the `-fsyntax-only` option was used.

7. Gccgo walks the IR and determines the types of all constants, and uses that to determine the types of variables whose type is implied by the initialization expression. In order to determine the type of a constant expression gccgo must know the context in which the expression is used (e.g., if a constant is passed to a function it gets the type of the function parameter), so this pass is done using a specific traversal rather than the general mechanism. During this pass gccgo also looks for global variables whose initializers will require running an initialization function at runtime.

8. Gccgo walks the IR and checks all types in statements and expressions, issuing errors as appropriate.

9. Gccgo walks the IR and issues an error if a function with results can fall off the bottom without an explicit return statement.

10. Gccgo builds export information for all globally visible identifiers.

11. Gccgo finds all interface types with hidden methods. It then walks the IR looking for all named types that implement those interfaces. For each such type/interface pair, gccgo builds an interface method table. Gccgo will use this table if a value of the named type is ever converted to a value of the interface type. Normally gccgo builds these tables as needed. However, when the interface type has hidden methods, the interface method table has to refer to the corresponding hidden methods of the named type. Such a type conversion could occur in a different package, but when gccgo is compiling that other package it would not be able to build the required interface method table, since the hidden methods of the named type will be static to this package. So gccgo must build all such tables here, in case it needs them when compiling some other package.

12. Gccgo walks the IR looking for all uses of `&&` and `||` and converts them into `if` statements. This simplifies the following pass.

13. Gccgo walks the IR looking for all expressions and subexpressions with side effects, and rewrites them to use temporary variables that are set before the statement. Gccgo does this for top level expressions during an earlier pass, in order to split up tuple assignments. Here gccgo does this for all subexpressions. This implements Go's rules about order of expression evaluation.

14. Gccgo looks for functions that call `recover`. For each such function, it renames the function, adds a new `bool` parameter, and creates a thunk under the old name that calls the renamed function. The value passed for the new parameter is whether the call to the `recover` function may recover a panic. Gccgo then walks the IR looking for calls to `recover` and rewriting them so that the function only calls `recover` if the new parameter is true. This is described in more detail in section 7.5.

15. Gccgo walks the IR looking for `go` and `defer` statements. It changes them to gather their arguments into structs, and pass a pointer to that struct to a runtime function. It creates little thunks that receive a pointer to the struct and call the real function from the `go` or `defer` statement, unpacking the arguments from the struct.

16. Finally, gccgo walks the list of global declarations and generates GENERIC for all functions, global variables, global types, and global typed constants. In the future it would be desirable to generate GIMPLE directly, to avoid the conversion to GENERIC. However, no frontend does that currently, there is no interface for it, and the GIMPLE requirements are undocumented.

5 Import and Export

All Go code lives in a package. Packages export data about types, functions, variables and constants, and they import that data from other packages. The exported data is intended to be quickly consumed at compile time.

5.1 Finding Export Data

Gccgo currently puts the export data in a special section in the output file, named `.go_export`. This should work with any object file format that supports named sections. However, the import code, which needs to look for this section in the import file, currently only works for ELF, or when using a build procedure that copies the export data into a separate file.

When gccgo sees the statement `import "p"`, then if `p` is an absolute path it simply opens the file. If not, it searches for the file in the directories specified by the `-I` and `-L` options. For each directory, it tries to open the following file names:

- `p`
- `p.gox`
- `libp.so`
- `libp.a`
- `p.o`

The intent is to permit packages to provide `.gox` files that only contain the export data, and are separate from the actual compiled code.

Gccgo can read the export data in three different ways: as a file containing only export data, as an ELF object file containing export data in the `.go_export` section, or as an archive containing one or more ELF object files. In the last case the contents of the `.go_export` sections are concatenated; this is convenient when packing several different packages into a single archive.

5.2 Export Data Format

The export data format is readable as text, but it is a binary format in the sense that spaces and newlines must appear exactly as expected. The data is more or less Go syntax, although for historical reasons it uses explicit semicolons rather than relying on newlines. In this paper I will describe the information in the export data, but not the precise format.

5.2.1 Header

The export data starts with a header, as follows:

- A version number or magic number, the four byte string `v1;\n`.
- The package name.
- The unique package prefix, as specified by the `-fgo-prefix` option or the default of `go`.
- The package priority. This is used to run package initialization routines in the correct order, such that a package's initialization is complete before starting the initialization of any package that imports it. The priority of a package that does not import any other packages is zero. The priority of a package that does import other packages is one more than the largest priority of any imported package. This simple mechanism works because the Go language prohibits package import loops.
- An optional list of the initialization routine of this package, if any, and the initialization routines for any packages that it imports. Each entry in the list is the package name, the name of the initialization function, and the package priority. This is used when compiling the main package to call all required initialization routines. The list is inclusive, in that it includes the initialization routines of all imported packages and of all packages that they import.

5.2.2 Globals and Types

The export data header is followed by the export data for each exported function, variable, type, and constant. For functions and variables the export data is simply the name and type. For constants it is the constant expression and the type if it has one.

Exporting types is more complex. A type may refer to itself, directly or indirectly; gccgo handles this by giving each type a reference number. The first time gccgo exports a type, it writes out the reference number and the definition. Subsequent references to the type in the export data use just the reference number.

The various predeclared types can not be defined in terms of any other type. Gccgo assigns a unique reference number to each predeclared type. These reference numbers are all negative and range from `-1` to `-19`. Gccgo uses positive numbers for the reference numbers used for types defined in the source code.

An exported function or variable may use a named type imported from a different package. When this happens, gccgo must fully define the type in the export data, as the imported package may not be available during a later compilation. Gccgo writes out the full name of an imported type, including the package name and the package's unique prefix.

When gccgo imports packages and reads a named type imported from a different package, it saves the type information but does not permit the package being compiled to use the type. It is possible for gccgo to import two different packages, both of which refer to the same type that they imported from a third package. Gccgo recognizes that case by the full name of the type, and ensures that all references point to a single definition. If gccgo then sees an import of that third package, it must continue to use to single definition, and it must now make the type available for use by the package which it is compiling. This is complicated somewhat by importing packages under different names, or even importing the same package twice.

When the source code exports a type that is defined in terms of a type which is not exported, gccgo must put the full definition of the unexported type in the export data, using a hidden name. When gccgo imports this package, it must read the definition of the type, but it must not make the type available to the package which it is compiling.

5.2.3 Trailer

The export data ends with a checksum of all the data. This checksum is not currently used. The intent is to provide a way for build systems to see whether the export data of a package has been changed when the package is recompiled. If a package is changed and recompiled, but the export data has not changed, then there is no need to recompile any other package that imports the changed package. It is only necessary to relink any executables. This can be used to speed up rebuilds after a change to, e.g., the body of a function.

6 GCC interface

Gcc is distributed with several different language frontends: C, C++, Fortran, Java, Ada, Objective C, Objective C++. The gccgo project is simply adding another frontend. In this section I'll discuss what is required to add a new frontend.

I will describe the state of the interface as of this writing. It is likely that things will change over time. Gcc's frontend interface is, as of this writing and as far as I know, undocumented. All files are in the language subdirectory, which for gccgo is simply named `go`.

6.1 GCC interface Files

Every gcc frontend has certain files with specified names.

6.1.1 `config-lang.in`

The file `config-lang.in` is a shell script sourced by the top level `configure` script. It sets some shell variables:

- `language`: The name of the language, in this case `go`.
- `compilers`: The name of the compiler proper, the program linked against the gcc middle-end, in this case `go1`.
- `target_libs`: The name of the top level `Makefile` targets for any target libraries that should be built for this language, in this case `target-libgo`.
- `gtfiles`: The names of any files that must be examined for the `GTY` markings used by the gcc garbage collector.
- `lang_dirs`: Other top level `Makefile` targets that should be built if the language is built. Gccgo does not set this.
- `subdir_requires`: Other gcc frontends that must be available in order for this frontend to be built. This exists for the Objective C++ frontend, which requires both the C++ and the Objective C frontends to be present. Gccgo does not set this.
- `boot_language`: Set to `yes` if this language is needed to bootstrap gcc itself. Gccgo does not set this.
- `boot_language_boot_flags`: Options to pass to `make` when building this language. Gccgo does not set this.
- `build_by_default`: Set to `yes` if this language is built by default. Gccgo does not set this.
- `outputs`: A list of files that will have `configure` substitutions applied to them. Gccgo does not set this.

6.1.2 `Make-lang.in`

The file `Make-lang.in` is a `Makefile` fragment that is included in the gcc `Makefile`. It provides the rules for building the program listed in `compilers` in `config-lang.in`. It must also provide several targets whose names are based on the `language` field in `config-lang.in`. That is, the gccgo `Make-lang.in` defines the targets below, with `language` replaced by `go`.

- `language`: this target must build the compiler proper.
- `language.all.cross`: any tools required for a cross compiler. For gccgo this builds `gccgo-cross`.
- `language.start.encap`: any tools required in order to run gcc. For gccgo this builds `gccgo`.
- `language.rest.encap`: anything that must be built after gcc can run.
- `language.info`: build `.info` files.
- `language.install-info`: install `.info` files.
- `language.dvi`: build `.dvi` files.
- `language.pdf`: build `.pdf` files.
- `language.install-pdf`: install `.pdf` files.
- `language.html`: build `.html` files.
- `langage.srcinfo`: build `.info` files in the source directory for releases.
- `language.srcextra`: build any additional source directory files required for a release, such as `yacc` output.
- `language.tags`: build `TAGS` files.
- `language.man`: build man pages.
- `language.srcman`: build man pages in the source directory for releases.
- `language.install-common`: install the compiler proper and any supporting programs. For gccgo this installs `go1` and `gccgo`.
- `language.install-plugin`: install anything needed by compiler plugins.
- `language.install-man`: install man pages.
- `language.uninstall`: uninstall everything.
- `language.mostlyclean`: clean most things.
- `language.clean`: clean everything that can be rebuilt.
- `language.distclean`: clean everything create by configure.
- `language.maintainer-clean`: clean everything that a maintainer can rebuild.
- `language.stage1`: copy all bootstrap files to `stage1/language`.
- `language.stage1`: copy all bootstrap files to `stage2/language`.
- `language.stage3`: copy all bootstrap files to `stage3/language`.
- `language.stage4`: copy all bootstrap files to `stage4/language`.
- `language.stageprofile`: copy all bootstrap files to `stageprofile/language`.
- `language.stagefeedback`: copy all bootstrap files to `stagefeedback/language`.

6.1.3 `lang.opt`

The file `lang.opt` lists language specific options. The format is the same as the general `.opt` files, and is described in the gcc internals documentation.

6.1.4 `lang-specs.h`

The file `lang-specs.h` may optionally exist. It is a C file that consists only of a partial initializer for the `default_compilers` array in `gcc.c`. This may be used to tell the `gcc` driver program how to compile files for a given extension. For gccgo this file gives a spec for compiling files with an extension of `.go`.

6.1.5 `subdir-tree.def`

The file `subdir-tree.def` may optionally exist. Here `subdir` is the name of the subdirectory where the `config-lang.in` file is found, which need not be the same as the `language` defined in that file. In particular, they are different from the C++ frontend, for which `subdir` is `cp` but `language` is `c++`. In any case, the file `subdir-tree.def`, if it exists, contains additional `DEFTREECODE` definitions used with GENERIC. This permits language specific extensions to GENERIC. This feature is used by the C, C++, Objective C, and Objective C++ frontends.

6.2 Driver

Several gcc frontends have a driver program. For gccgo the driver is simply named `gccgo`. This driver program compiles and links code written in the language. The main `gcc` driver may be set up to invoke the appropriate compiler based on the extension of the source file (see the description of `lang-specs.h`, above). The main purpose of the language-specific driver program is to add required libraries to the link line. The driver program is built by the `language.all.cross` and `language.start.encap` targets in `Make-lang.in` described above, and is installed by the `language.install-common` target.

The language specific driver program does not contain a `main` function. Instead, it is linked with the `make` variable `$(GCC_OBJS)`. The language specific driver must provide two functions and one variable.

- The driver calls `lang_specific_pre_link` after doing any required compilation and before doing any linking. This function takes no arguments and returns `int`. It should return `0` on success and some other value on failure. The gccgo version simply returns `0`. This Java frontend uses this hook to implement the `--main` option.
- The global variable `lang_specific_extra_outfiles` has type `int`. The gcc driver uses this to increase the size of the `outfiles` array that the driver code builds to hold the output file names. The gccgo driver simply sets this to `0`. The Java frontend sets this to `1` if it sees the `--main` option.
- The function `lang_specific_driver` does most of the work. The gcc driver invokes the function after parsing the options but before doing any work. It takes pointers to the list of options and the number of added libraries, which it may update. The gccgo driver uses this function to add the `-lgobegin` and `-lgo` libraries when linking.

6.3 Frontend Language Hooks

As noted above, `Make-lang.in` must describe how to build the compiler proper. This is the program that the driver will invoke to compile a source file into an assembly file. Gccgo names this program `go1`. It must be built from object files linked against `libbackend.a`, generally via the `make` variable `$(BACKEND)`. The language compiler may provide a `main` function, but it does not have to.

The language compiler must define a global variable named `lang_hooks`, of type `struct lang_hooks`. This variable should be initialized to `LANG_HOOKS_INITIALIZER`. This initializer value is controlled by a set of macros whose names begin with `LANG_HOOKS_`. These macros are defined in the file `langhooks-def.h`. The usual pattern is for a frontend file to include `langhooks-def.h`, and then for each required language hook to `#undef` the `LANG_HOOKS_` macro and `#define` it to a hook appropriate for the frontend. These `#undef #define` pairs must occur before the use of `LANG_HOOKS_INITIALIZER`. I don't know of any documentation for the language hooks, and I will not document them all here. I will quickly mention the language hooks that must always be implemented.

- `LANG_HOOK_INIT` initializes the frontend, and must also call the following functions:
  - `build_common_tree_nodes`
  - `set_sizetype`
  - `build_common_tree_nodes_2`
  - `build_common_builtin_nodes`
- `LANG_HOOK_PARSE_FILE` must parse all the input files, which may be found at `in_fnames` of length `num_in_fnames`.
- `LANG_HOOK_TYPE_FOR_SIZE` must be defined, and must return a `tree` for the frontend specific integer type for a given number of bits. It will only be called for types of a precision used by the frontend.
- `LANG_HOOK_TYPE_FOR_MODE` must be defined, and return a `tree` for the frontend specific integer type for a given mode.
- `LANG_HOOK_GLOBAL_BINDINGS_P` must be defined. It must return an `int` that is `0` when not processing a global variable. This hook is rather ill-defined, and is only meaningful when the frontend is calling into a backend function for some operation such as constant folding.

- `LANG_HOOK_PUSHDECL` must be defined. It does not have to actually do anything, although it will be called by the default implementation of `LANG_HOOK_BUILTIN_FUNCTION`. This hook is rather ill-defined.
- `LANG_HOOK_GETDECLS` must be defined. It is called when generating STABS debugging information, and by the default implementation of `LANG_HOOKS_WRITE_GLOBALS`.
- `LANG_HOOK_WRITE_GLOBALS` should be defined to write out all global functions and variables. The default definition will suffice for a language that uses GENERIC as the IR, but not for frontends such as gccgo that use a different IR. This hook must call the following functions:
  - `cgraph_finalize_compilation_unit`
  - `wrapup_global_declarations`
  - `check_global_declarations`
  - `emit_debug_global_declarations`
- `convert` is a function that should be a langhook but is not. The middle-end will call this in a few places to convert an expression to a type.

6.4 Frontend GTY Support

In order to support gcc's internal garbage collector, the language frontend must define certain types. These types do not need to be used, but they must be defined and marked with a `GTY` marker. These markers are documented in the gcc internals manual. The following types must be defined.

- `struct lang_type`: language dependent contents of a type in GENERIC. This may have just a dummy field.
- `struct lang_decl`: language dependent contents of a decl in GENERIC. This may have just a dummy field.
- `struct lang_identifier`: language dependent contents of an identifier. This must include a field of type `struct tree_identifier`.
- `union lang_tree_node`: a union of `tree_node` and `lang_identifier`, with a `GTY` marker describing how to follow the chain field.
- `struct language_function`: a field that will be attached to a `struct function` and accessed by the `cfun` global variable, which the frontend may use for any purpose.

The file(s) where these types are defined must be listed in `gtfiles` in `config-lang.in`. The file(s) must use `#include` to include the generated files as usual for `GTY` markings.

7 Runtime

The Go language has a significant runtime component, used to implement garbage collection, concurrency, type reflection, and other features. Go also has a standard library. Gccgo puts this code in the `libgo` library, which has four parts:

- A copy of the Go library distributed with the gc Go compiler, with some minor changes.
- A system call package that replaces the `syscall` package of the gc Go compiler.
- Runtime support copied from the gc Go compiler. This is mainly the memory allocator and the garbage collector.
- Runtime support code called by code generated by gccgo.

7.1 Go Library Differences

These are the differences between the gc Go library and the gccgo Go library:

- Avoid cases where the gc library uses assembly code. The gccgo version uses C code instead. For example, in the gc Go library the `bytes.IndexByte` function has different assembler implementations for each supported target. The gccgo version is written in C, and simply calls `__builtin_memchr`.

- Directory reading and `stat` handling. The gc Go library calls the system calls directly. The gccgo version calls the C functions `opendir` and `stat`.
- Some test code is adjusted for the above changes, and because gccgo does not yet implement `runtime.Caller`.

7.2 System Call Support

The gccgo `syscall` package is completely different from the gc library. The `syscall` package provides low level system interfaces like `open`.

The gc library uses a program that runs gcc and reads the debug information to generate struct definitions in Go format. The gccgo library is similar, but instead uses a new gcc option, `-ggo`, to generate debugging information in Go format directly. The gccgo library uses `-ggo` with a set of system header files at build time, and uses a shell script to massage the output.

The gc library uses assembler code to run system calls directly. The gccgo library instead calls the appropriate C functions, using gccgo's `__asm__` extension.

7.3 Interfaces

Go supports values of interface type. An interface type is simply a set of methods. Any value of a concrete (i.e., non-interface) type may be assigned to a variable of interface type, provided the concrete type implements the methods of the interface type. Given a value of interface type, the program may call any method defined by that interface type. A value of one interface type may be converted to a value of a different interface type; this will cause a runtime failure if the concrete type stored in the interface value does not support all the methods of the new interface type.

In other words, interface types provide type polymorphism, but, unlike C++, it is not tied to a type heirarchy. Runtime conversion of interface types means that there is a dynamic runtime component, similar to `dynamic_cast` in C++ though again without a type heirarchy.

In order to support interfaces, and also the `reflect` package that is used for type reflection, gccgo builds a type descriptor for every type used in the program. The name of the type descriptor is a mangled version of the type. For a named type, the mangled name includes the package name and the package's unique prefix. Type descriptors for named types are defined in the package where the type is defined, but type descriptors for unnamed types are passed to gcc's `make_decl_one_only` function so that they can be shared between object files. I won't describe the format of the type descriptor here.

Gccgo represents a value of the empty interface type `interface{}` as a `struct` with two fields.

```
struct __go_empty_interface
{
  const struct __go_type_descriptor
    *__type_descriptor;
  void *__object;
};
```

The `__type_descriptor` field is a pointer to the type descriptor for the concrete type of the value assigned to the interface. If this field is `NULL`, then the interface value is `nil`. If the value stored in the interface is a pointer type, then the `__object` field is simply that pointer. Otherwise, the `__object` field is a pointer to the actual value, which will have been copied to the heap.

Gccgo represents a value of a non-empty interface type also as a struct with two fields.

```
struct __go_interface
{
  const void **__methods;
  void *__object;
};
```

The `__object` field holds the value in the same way as for an empty interface. The `__methods` field points to a method table. The first field in the method table is a pointer to the type descriptor of the value's type. If this field is `NULL`, then the interface value is `nil`. The subsequent fields are function pointers, in the order of the methods of the interface type. Thus a method table is similar to a C++ virtual table. The method table for a given interface type/value type pair is always the same.

When possible, the method table is constructed at compile time. When a runtime conversion is done, the method table is built at runtime. Interface and type methods are kept sorted by name, so building a new method table is linear in the number of methods. A possible future enhancement will be to use a hash table to map interface type/value type pairs to existing method tables.

Calling a method on a value of interface type requires loading the method table, loading the appropriate function pointer from the method table, and calling the function. Thus it is similar to a virtual function call in C++. The Go method call `iv.Method(arg)` is converted to code that in C would look like `iv->__methods[1](arg)`.

7.4 Stack Splitting

The Go language makes it very easy to create a new thread of execution via the `go` statement. Naturally a new thread of execution must allocate a stack. Making the stack too large will waste address space. Making the stack too small risks stack overrun. Gccgo address this problem via stack splitting. The start of each function has a short sequence of instructions, typically just two instructions, which checks whether there is enough room on the current stack for the function's stack frame. If there is not, a new stack segment is automatically allocated, in such a way that when the function returns the stack automatically moves back to the old stack segment.

On i386, for a function with a stack frame smaller than 256 bytes, the initial instructions are simply

```
cmpl    %gs:48, %esp
jb     .L2
```

Normally the stack does not need to be split, and the branch is not taken. So while stack splitting does introduce additional overhead to every function, that overhead is relatively small.

Stack splitting has the potential to be generally useful, and is implemented in gcc's middle-end via the `-fsplit-stack` option. Gccgo automatically turns on the option.

7.5 Panic and Recover

The predeclared functions `panic` and `recover`, in conjunction with the `defer` statement, serve as a dynamic exception mechanism. When Go code calls `panic`, the Go runtime walks up the stack, executing functions passed to the `defer` statement. If a deferred function calls `recover`, the stack walk is stopped, and the value passed to `panic` is returned by `recover`. If `recover` is called by a deferred function when no call to `panic` is in effect, `recover` returns `nil`.

If `recover` is called by a function which was not the immediate argument to `defer`, it returns `nil`. In other words, if `defer` is used to execute a function, and that function calls another function that in turn calls `recover`, then the call to `recover` should return `nil` even if there is a panic in progress. This permits deferred functions to call functions which use `panic` and `recover` themselves without getting confused by the fact that there is an ongoing panic.

The stack walk does not actually unwind the stack. The calls to the deferred functions are executed as though they were called directly by `panic`. This permits them to use the `runtime.Callers` function to get a stack trace (gccgo does not currently implement `runtime.Callers`, but eventually it will). If the deferred function calls `recover` to interrupt the stack walk, the stack is unwound after the deferred function returns, and execution continues at the caller of the function that ran `defer` (or at the next deferred function if that function ran `defer` more than once).

The most complex issue is that when a program calls `recover`, it is necessary to know whether the call is being made by a function that was the immediate argument to `defer`. The gc compiler determines this by passing the argument frame pointer to `recover`, which uses it to get the stack pointer of the caller of `recover`. The gc stack frame permits it to use that to get the caller of the caller of `recover`, to see whether `recover` was called at the appropriate location on the call stack.

This procedure would be difficult to implement in gccgo. Since gccgo uses the standard ABI for whatever target it is configured for, unwinding the stack is much more difficult. The unwind library does not provide the necessary interfaces. Gccgo's stack splitting code introduces additional stack frames at unpredictable moments, and they are hard to identify.

Instead, gccgo checks whether a `defer` statement may be invoking a function which calls `recover`. The `defer` statement always creates a thunk that calls the actual deferred function with the appropriate arguments. It is that thunk that is stored on the runtime's defer stack. If the deferred function may call `recover`, the thunk calls the runtime function `__go_set_defer_retaddr`. It passes the address of a label immediately after the call to the actual deferred function. This uses gcc's existing address-of-label extension. This label is thus the return address of the deferred function.

For any function that calls `recover`, gccgo inserts a call to `__builtin_return_address`. A function can always reliably determine its immediate return address. That return address is passed to the runtime function `__go_can_recover`. That function can compare the return address to the address saved by `__go_set_defer_retaddr`, if any. If the addresses match, then the call to `recover` can succeed, and `__go_can_recover` returns `1`. Otherwise, it returns `0`. This value is saved in a compiler-created local variable, and the actual call to `recover` is expanded to check that local variable.

That works for normal cases, but it does not handle the case in which the function that call `recover` splits the stack on entry. In that case `__builtin_return_address` will return the address of the stub which restores the old stack, rather than the address of the caller. To avoid that problem, gccgo splits any function which calls `recover` into two functions. A small thunk which uses at most a very small stack frame, and the real function. The small thunk is marked to not split the stack. It calls `__go_can_recover`, and passes the result to the real function via a compiler-created additional hidden parameter. The real function is marked uninlinable to ensure that it is not inlined into the small thunk causing the latter to have a large stack frame.

That is sufficient to let us know whether `recover` should return a panic value if there is one, at the cost of having an extra thunk for every function which calls `recover`.

Now for the `panic` function. It walks the list of deferred functions, calling them as it goes. When a deferred function sucessfully calls `recover` and returns, the panic stack is marked. This stops the calls to the deferred functions, and starts a stack unwind phase. The unwinding is done using gcc's general unwind mechanism. This means that every function which calls `recover` has an exception handler. The exception handlers are all the same: if this is the function in which `recover` returned a value, then simply return from the current function, effectively stopping the stack unwind. If this is not the function in which `recover` returned a value, then resume the stack unwinding, just as though the exception were rethrown in C++.

In order to ensure that all defer handles are run in all cases, any function that uses `defer` is wrapped like this. Here the function `__go_check_defer` is the simple exception handler mentiond above. The function `__go_undefer` runs all functions on the defer stack associated with its argument. Of course, one of those functions may call `panic`, which is why there is an exception handler in the `finally` clause. The compiler generated variable `DEFER.0` is used to handle `defer` correctly when functions are inlined.

```
void *DEFER.0;
try
  {
    try
      {
        // Function body.
      }
    catch (...)
      {
        __go_check_defer (&DEFER.0);
        return;
      }
  }
finally
  {
  lab:
    try
      {
        __go_undefer (&DEFER.0);
      }
    catch (...)
      {
        __go_check_defer (&DEFER.0);
        goto lab;
      }
  }
```

7.6 Goroutines

Currently gccgo runs every goroutine in a separate thread. This is inefficient. It would be much better to multiplex goroutines onto threads, as the gc compiler does.

8 Optimization

The gccgo frontend currently does very little optimization, though of course all of gcc's middle-end optimizations are available. There are various possible optimizations appropriate for the frontend in the future.

- Because Go only permits calling declared functions, and because function declarations either come from source code which the compiler can see or from export data which the compiler generates, there is a lot of scope for automatically generated function annotations.
  - Automatically annotate const/pure functions.
  - Annotate which pointer arguments escape the function, permitting pointers passed to the function to be stored on the stack in some cases.
  - Simple cross-package inlining for small functions.
  - Annotate which array and struct arguments are not changed, permitting passing a pointer instead.
- All array, slice, and string indexes in Go are bounds checked. VRP can not always see when these bounds checks can be eliminated because the middle-end does not know that, e.g., strings in Go are immutable. The frontend could do a better job in some cases.
- Cross package inlining in the frontend makes it possible to devirtualize interfaces in some cases. The middle-end is unlikely to be able to turn the method call sequence back into a direct function call, but the frontend could.
- Switch statements could be better optimized, using a combination of `if` statements and `SWITCH_EXPR` when some but not all of the cases are constants.
- Anything the compiler can do to give hints to the garbage collector could be useful.

9 Debugging

The output of gccgo can be used with gdb today, but it is awkward. Doing a better job is going to require changes to gdb.

- A major debugging issue is the names of function and global variables. The Go name for a function or variable always includes a package name. The package name and function/variable name are separated by a period. Gdb does not expect names to contain a period, so all references to these names have to be quoted and tab completion does not work properly. This makes it painful to set breakpoints by name; I often set them by filename and line number instead.
- There are several Go runtime types which gdb will need to learn about, either via Python scripts or a direct port: strings, slices, interfaces, channels, maps, type descriptors. Right now printing these values is just like printing a `struct` in C.
- Debugging split stack code is quite awkward in gdb today. When you single step into a function, and it splits the stack, you need to single step through the stack splitting code down to the branch back to the function. This needs to be improved.
- Gdb's support for multi-threaded programs is functional but often awkward to use. Go programs tend to have many goroutines. How to debug such programs effectively is an open question.

10 Future Work

The gccgo frontend is a work on progress. These are the some of the goals.

- Increase the separation between the frontend proper and the gcc interface.
- Implement the optimization ideas described above.
- Don't use a single thread per goroutine, but instead multiplex several goroutines onto a single thread.
- Improve the garbage collector, which is currently a simple mark and sweep collector.
- The Go language continues to change, and the gccgo frontend must continue to change with it.