The Go Frontend for GCC

Ian Lance Taylor

GCC Summit, October 25, 2010

Go

- Go is a new experimental systems programming language. Main developers are:
  - Russ Cox
  - Robert Griesemer
  - Rob Pike
  - Ian Lance Taylor
  - Ken Thompson
- It was released as free software in November 2009.
- The language continues to change.
- There are currently two Go compilers, the gc compiler and gccgo.
- This talk is about gccgo, a GCC Go frontend, not about the language.
- There is an upcoming talk on the language itself.
- `http://golang.org/`

Gccgo frontend

- Supports compiling Go with GCC.
- A new frontend, joining C, C++, Java, Ada, Fortran, Objective C, Objective C++.
- Currently lives on `gccgo` branch, should move to mainline very shortly.
- Written in C++.
- Currently 50,000 lines of code including comments.
- Actively changing as the language changes.

New IR

- Gccgo uses a new IR, called GOGO.
- IR is based around three main independent C++ classes.
  - `Statement`
  - `Expression`
  - `Type`
- These are virtual base classes with many different child classes (`Assignment_statement`, `Return_statement`, etc.).
- Other classes are `Gogo`, `Block`, `Variable`, `Function`, `Named_object`, `Package`, etc.

Child class identification

Gccgo handles child classes in two ways.

- The base classes each have an enum type with one value for each child class type.
- The enum values are stored in a field of the base class.
- The base classes each have a `classification` method which returns the enum value.
- The base classes also have virtual functions which the child classes implement.

```
// Use the enum.
Statement* s
switch (s->classification())
  {
    case STATEMENT_ASSIGNMENT:
    // Etc.
  }

// Use a virtual function.
s->check_types(gogo);
```

IR traversal

- A traverse routine walks over everything in the tree.
- You invoke it with a child of the `Traverse` class which implements operations.
- Bitmask of interest short-circuits traversal.

```
class Traverse
{
 public:
  // These bitmasks say what to traverse.
  static const unsigned int traverse_variables =    0x1;
  static const unsigned int traverse_constants =    0x2;
  static const unsigned int traverse_functions =    0x4;
  static const unsigned int traverse_blocks =       0x8;
  static const unsigned int traverse_statements =  0x10;
  static const unsigned int traverse_expressions = 0x20;
  static const unsigned int traverse_types =       0x40;

  Traverse(unsigned int traverse_mask);

  // If traverse_variables is set in the mask, this is called for
  // every variable in the tree.
  virtual int
  variable(Named_object*)
  { return TRAVERSE_CONTINUE; }

  // Etc.
};
```

Parsing

- The Go language is designed to be easy to parse.
- Gccgo uses a hand-written recursive descent parser.
- Names in Go may be used before they are defined.

```
// Function call or type conversion?
var v1 = f1(0)
var v2 = f2(0)
func f1(i int) int { return 0 }
type f2 int

var v3 = (*p1)(nil)
var v4 = (*p2)(nil)
var p1 *func(*int) int
type p2 *int
```

Lookahead

There is only one case where gccgo does arbitrary lookahead in the parser.

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

Constants

- Go constants are untyped and the language does not limit their size (though the implementation may).
- This is valid: `var v = (1 << 100) >> 99`
- Gccgo uses GMP and MPFR libraries to handle constants.
- It should also use MPC in the future.

Gccgo extension

Gccgo extends the language to permit using `__asm__` to specify an external function name for a function declaration.

```
func libc_open(name *byte, mode int, perm int) int __asm__("open")
```

Used to call C functions.

GCC changes

- New `-fsplit-stack` option (already in mainline).
- New `-fplan9-extensions` option (already in mainline).
- New `-ggo` option for runtime library implementation, described later.
- Don't optimize division by zero with `-fnon-call-exceptions` (already in mainline).
- Recognize `.go` files in driver.
- Minor changes to configuration scripts.
- Autoconf and libtool support (already proposed upstream).
- New directories `gcc/go`, `libgo`, and `elfcpp`.

GCC interface

- Basic frontend attachment is straightforward though undocumented.
- I copied the Fortran and Java frontends.
- Create some files in the `gcc/go` directory:
  - `config-lang.in`
  - `Make-lang.in`
  - `lang.opt`
  - `lang-specs.h`
- Write a C file, in this case `go-lang.c`, which implements language hooks defined in mainline file `langhooks.h`.

At that point you have a working frontend which doesn't do anything. There are more details in the paper.

Language hooks

- The only documentation for the language hooks is comments in `langhooks.h`.
- Many language hooks are specific to frontends which use extensions to GENERIC. Some must be implemented by all frontends even though they are meaningless for frontends which do not use GENERIC.
  - `LANG_HOOKS_PUSHDECL`
  - `LANG_HOOKS_GETDECLS`
  - `LANG_HOOKS_GLOBAL_BINDINGS_P`
- The `LANG_HOOKS_TYPE_FOR_MODE` and `LANG_HOOKS_TYPE_FOR_SIZE` hooks must be implemented. This breaks the middle-end type abstraction. They are even called when expanding to RTL, although at that point the frontend should be long gone.
- A function named `convert` must be implemented. It should be a language hook, and even then it breaks the abstraction.

Language hook changes

- It would be a good idea to separate the language hooks into three parts.
  - Hooks specific to the C language family.
  - Hooks specific to frontends that use GENERIC.
  - General purpose hooks.
- It would be a good idea to stop calling language hooks after conversion to GIMPLE.

Generating GENERIC

- Gccgo generates GENERIC.
- Detailed interaction of how to generate GENERIC—which functions to call and when—is obscure and undocumented.
- For example, for a function:
  - Build `RESULT_DECL` for function.
  - Set `current_function_decl` global variable.
  - Call `allocate_struct_function`.
  - Call `gimplify_function_tree`.
  - Call `cgraph_finalize_function`.
- After conversion to GENERIC, middle-end then immediately converts to GIMPLE, then briefly to trees, then to RTL.
- It would be more efficient for gccgo to generate GIMPLE, but that is undocumented and there are no current hooks for it.

Passes

- Parse input files and build GOGO IR.
- Define predeclared identifiers.
- Finalize methods of all types.
- Lower parse tree.
- Verify that all type definitions are valid.
- Determine types of constants and variables.
- Check types and issue type errors.
- Check for missing return statements.
- Build export data.
- Generate type method tables for interfaces with hidden methods.
- Convert `&&` and `||` to `if` statements.
- Move expressions with side effects to temporaries.
- Adjust calls to `recover`.
- Simplify `go` and `defer` statements.
- Convert to GENERIC.

Export

Export currently uses a readable format, stored in the `.go_export` section of the object file.

```
package p
type T1 struct { p *T2 }
type T2 struct { p *T1 }
var V T1
```

```
v1;
package p;
prefix go;
priority 1;
import p go.p..import 1;
type <type 1 "T1" <type 2 struct
     { .go.p.p <type 3 *<type 4 "T2"
       <type 5 struct { .go.p.p <type 6 *<type 1>>; }>>>; }>>;
type <type 4>;
var V <type 1>;
checksum 5BED3D5A7EA98BB0F530838B7E2B8BA2D9A8AB21;
```

- Priority is used for `init` function and for initialized global variables. They must be run in import order.
- Checksum is SHA1, and is currently unused.
- Export data may be concatenated; it will all be read.

Import

- Import of a file `foo` looks for several files.
- Currently these files must be found both at compile time and at link time.
- The frontend uses `-I` and `-L` options to find the files.

- `foo.gox`
  - May be created using `objcopy -j .go_export`.
- `foo.o`
- `libfoo.so`
- `libfoo.a`
  - Will read export data from all files in the archive.

Runtime support

- Many operations in the language require runtime support.
- The compiler generates calls to runtime support routines.
- Runtime routines are found in `libgo`, along with Go library routines.
- Runtime routine names all start with `__go_`.
- Runtime routines are written in C (not C++, not Go).

Interfaces

- An interface type in Go is a set of methods.
- An interface variable can hold any type which implements the methods.
- Gccgo builds a type descriptor for every type used in the program.
- The descriptor is accessible at runtime and is used for type reflection.
- The descriptor lists the type's methods, sorted by name.
- The descriptor for an interface type lists the interface's methods, sorted by name.
- Assigning a value to an interface requires building an interface method table for the value's type.
  - Similar to a C++ vtable.
  - Specific to the pair of the interface type and the non-interface type.
  - Built at compile time when possible, but in general must be built at runtime.
  - Building at runtime merges the interface methods

Go statements

- The `go` statement creates a new parallel thread of execution.
- Gccgo currently implements these using pthreads.
  - This is different from the gc compiler in which the runtime includes a scheduler, and goroutines are multiplexed onto OS threads.
- Creating a pthread can only pass a single parameter.
  - The `go` statement simplification pass rewrites go statements.
  - Any `go` statement with more than one parameter, or which calls a method, is simplified.
- The same simplification is applied to `defer` statements, in which the function call must be stored on a stack maintained by the runtime.

```
go f(a, b)  // Assume a and b are int.
```

becomes

```
type __go_struct_type { f func(int, int); a, b int; };
go func(p *__go_struct_type) { p.f(p.a, p.b) }
   (&__go_struct_type{f, a, b});
```

System call support

- Go library sources make system calls and use system structs.
- Gccgo reads the system header files and declares these automatically.
- This is implemented using a new option, `-ggo`.
- There is a new `gcc_debug_hooks` structure.
- Where possible, debugging information is printed in Go syntax.
- Used with `-S` and `grep` to extract the debug info.

Sample output from `#include <sys/stat.h>` with a 64-bit compiler:

```
#GO type _time_t int64
#GO type _timespec struct { tv_sec int64; tv_nsec int64; }
#GO type _off_t int64
#GO type _stat struct {
        st_dev uint64; st_ino uint64; st_nlink uint64; st_mode uint32;
        st_uid uint32; st_gid uint32; __pad0 int32; st_rdev uint64;
        st_size int64; st_blksize int64; st_blocks int64;
        st_atim _timespec; st_mtim _timespec; st_ctim _timespec;
        __unused [2+1]int64; }
```

Stack splitting

- In Go it is very easy to create a new thread of execution (a goroutine).
- Each thread of execution requires a stack.
- Making the stack too large wastes address space, a serious consideration on a 32-bit system.
- Making the stack too small risks stack overrun.
- To avoid this issue, gccgo uses stack splitting.
- Each function starts with a short sequence of instructions to check whether there is enough stack space. If not, a new stack segment is automatically allocated, and released on function exit.
- On i386, for a function with a stack frame smaller than 256 bytes, the initial instructions are simply

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

- Stack splitting can be useful for C code as well.
- The gold linker supports calling from split stack code to non-split stack code.

C interoperability

- Scalar values may be passed back and forth.
- Go strings are a two element struct in C.

```
struct { const char *data; int length; };
```

- Arrays are passed by value, not reference.
- Slices are a three element struct.

```
struct { void *values; int count;
         int capacity; };
```

- Functions are the same when the types are convertible.
- Go functions which return multiple values return a C struct.
- Pointers are the same, but garbage collection acts differently.
  - The Go runtime will ignore C pointers.
  - If no Go code references a pointer, it may be deleted even if the C code still has a copy.

Optimization opportunities

- Function declarations are always either visible in the source code or come from export data which the compiler generates. The compiler can annotate functions in the export data.
  - Annotate const/pure functions.
  - Annotate which pointer arguments escape, so pointer parameters can be stored on the stack.
  - Cross-package inlining for small functions.
  - Annotate unchanged array and struct arguments, to pass by reference rather than by value.
- All array, slice, and string indexes in Go are bounds checked. VRP does not always eliminate these bounds checks. The frontend could do a better job.
- Cross package inlining in the frontend makes it possible to devirtualize interfaces.
- 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.
- Garbage collector hints could be useful.

Debugging

- `gdb` works today.
- Single-stepping works as expected.
- Local variables may be inspected.
- Values are printed in their C form.
  - E.g., Go strings are a struct.
- Function names are prefixed with the package name and a period, which is awkward when setting breakpoints on a function.
- Split stack can confuse backtraces.
- A Google engineer will be working on improving gdb support.

Frontend work

- 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 collectors.
- The Go language continues to change, and the gccgo frontend must continue to change with it.