CC-Summer-2010: GoGo

Team:
Andreas Unterweger <andreas.unterweger _at_ sbg.ac.at>
Michael Lippautz <michael.lippautz _at_ sbg.ac.at>

Presentation: **Attach:gogo.pdf**
Sources: **Attach:gogo-src.tar.gz**
Documentation: **Attach:gogo-doc-final.pdf**

Input language

Project **GoGo** uses a subset of Go as input language. The complete specification for the Go language can be found on **http://golang.org/**, the subset used is described in the EBNF section below.
Go uses a syntax similar to C with additional keywords and some minor formatting rules concerning the position of opening curly brackets after expressions in if conditions and for loops as well as after function declarations. The most notable differences compared to the C language are that there is no while loop and no distinction between field access to structures and pointers to structures.

Example code

The following code illustrates the syntax of the subset of Go used based on a part of the Go code for **GoGo**'s object descriptor management.

```
// Copyright 2010 The GoGo Authors. All rights reserved.
// Use of this source code is governed by the MIT
// license that can be found in the LICENSE file.

package libgogo

type ObjectDesc? struct {
    Name string;
    PackageName? string;
    Class uint64;
    ObjType? *TypeDesc?;
    PtrType? uint64; //If 0, type objtype, otherwise type *objtype
    Next *ObjectDesc?;
};

//
// Classes for objects
//
var CLASS_VAR uint64 = 1;
var CLASS_FIELD uint64 = 2;

//
// Convert the uint64 value (returned from Alloc) to a real object address
//
func Uint64ToObjectDescPtr?(adr uint64) *ObjectDesc?;

//
// Appends an object to the list
//
func AppendObject?(object *ObjectDesc?, list *ObjectDesc?) *ObjectDesc? {
    var tmpObj *ObjectDesc? = object;
    if list != nil {
        for tmpObj = list; tmpObj.Next != nil; tmpObj = tmpObj.Next {
        }
        tmpObj.Next = object;
        tmpObj = list;
    }
    return tmpObj;
}
```

Output language

The output language of project **GoGo** is Plan9 assembler for **AMD64**? (official documentation: **http://doc.cat-v.org/plan_9/4th_edition/papers/asm**) which is also used by the Go compiler available from Google. As the latter is used for boot strapping, I/O and memory management routines from the compiler code which are coded in Plan9 assembler in order to be independent from libraries can be reused.

Example code

The following code illustrates the syntax of Plan9 assembler based on the assembler part of **GoGo**'s memory manager.

```
// Copyright 2010 The GoGo Authors. All rights reserved.
// Use of this source code is governed by the MIT
// license that can be found in the LICENSE file.
```

```
//
// GoGo Memory manager functions (ASM)
//

TEXT ·Brk(SB),$0-16 //Brk: 1 parameter, 1 return value
  MOVQ $12, AX //sys_brk (1 parameter)
  MOVQ 8(SP), DI //brk (first parameter => SP+8*64bit)
  SYSCALL //Linux syscall
  CMPQ AX, $0xFFFFFFFFFFFFF001 //Check for success
  JLS BRK_SUCCESS //Return result if successful
BRK_ERROR:
  NEGQ AX //Get errno
  MOVQ AX, 16(SP) //Return errno to indicate that an error occurred (return value after one parameters => SP+2*64bit)
  RET
BRK_SUCCESS:
  MOVQ $0, 16(SP) //Return 0 to indicate success (return value after one parameters => SP+2*64bit)
  RET

TEXT ·GetBrk?(SB),$0-8 //GetBrk?: no parameters, 1 return value
  MOVQ $12, AX //sys_brk (1 parameter)
  MOVQ $0, DI //brk (first parameter => SP+64bit)
  SYSCALL //Linux syscall
  CMPQ AX, $0xFFFFFFFFFFFFF001 //Check for success
  JLS GETBRK_SUCCESS //Return result if successful
GETBRK_ERROR:
  MOVQ $0, 8(SP) //Return 0 to indicate that an error occurred (return value after no parameters => SP+1*64bit)
  RET
GETBRK_SUCCESS:
  MOVQ AX, 8(SP) //First return value of syscall is in AX (return value after no parameters => SP+1*64bit)
  RET

TEXT ·TestMem?(SB),$0-16 //Write: 1 parameter, 1 return value
  MOVQ 8(SP), AX //Move address to AX (first parameter => SP+64bit)
  MOVQ $1234567890, (AX) //Move some value to address
  MOVQ (AX), BX //Move value back to BX
  XORQ $1234567890, BX //value XOR value => 0
  MOVQ BX, 16(SP) //Return value (0 if successful)
  RET
```

EBNF

comments := // to EOL | /* ... */

Basic

single_char = CHR(32)|...|CHR(127).
char = "'" single_char "'".
string = """ {single_char} """.
digit = "0"|...|"9".
integer = digit {digit}.
letter = "a"|...|"z"|"A"|...|"Z"|"_".

identifier = letter { letter | digit }.
selector = { "." identifier | "[" (integer | identifier selector) "]" }.

Expressions

cmp_op = ">" | "<" | ">=" | "<=" | "==" | "!=".
unary_arith_op = "+" | "-".
binary_arith_op = "*" | "/" .

factor = identifier selector | integer | char | string | "(" expression ")" | "!" factor.
term = factor { (binary_arith_op | "&&") factor}.
simple_expression = [unary_arith_op] term { (unary_arith_op | "||") term }.
expression = "&" identifier selector | simple_expression [cmp_op simple_expression].
expression_list = expression { "," expression }.

Statements

function_call = "(" [expression_list] ")".
function_call_stmt = identifier selector function_call.
assignment = identifier selector "=" expression
if_stmt = "if" expression "{" stmt_sequence "}" [else_stmt].
else_stmt = "else" "{" stmt_sequence "}".
for_stmt = "for" [assignment] ";" [expression] ";" [assignment] "{" stmt_sequence "}".
stmt_sequence = { stmt }
stmt = assignment ";" | function_call_stmt ";" | if_stmt | for_stmt | ";".

package_stmt = "package" identifier ";".
import_stmt = "import" string.

import_stmt_list = { import_stmt }.

Declarations

struct_var_decl = identifier type ";".
struct_var_decl_list = { struct_var_decl }.
struct_decl = "type" identifier "struct" "{" struct_var_decl_list "}" ";".
struct_decl_list = { struct_decl }.
type = (["[" integer "]"] identifier | "uint64" | "byte") | "string".
var_decl = "var" identifier type ["=" expression] ";".
var_decl_list = { var_decl }.

Functions

identifier_type = identifier ["*"] type.
identifier_type_list = [identifier_type { "," identifier_type }].
func_decl_head = "func" identifier "(" identifier_type_list ")" [type].
func_decl = "{" var_decl_list stmt_sequence ["return" expression ";"] "}".
func_decl_raw = ";".
func_decl_list = { func_decl_head (func_decl | func_decl_raw) }.

go_program = package_stmt import_stmt_list struct_decl_list var_decl_list func_decl_list.

Scanner

The scanner reads the input file char by char and generates the corresponding tokens. The tokens are simple tokens (referring to identifiers, strings, integers, ...) at first. It then converts identifying tokens with reserved keywords to the corresponding complex tokens (i.e. a 'for' token). Numbers are converted to an Integer token. The scanner also recognizes single quotes bytes, i.e. 'A', and converts them into the respective complex token. The scanner ends on unknown characters (failing), or if the end of file is reached.

Additionally the scanner tracks the current state in the file (line, column). This is used to track errors and provide a more readable output.

Features (in short)

- Dismiss white space and comments (either /* ... */, or //)
- Simple tokens generated from text
- Complex tokens generated from simple tokens by keyword table (if, for, else, ...)
- Generation of Integers from text
- Support for single characters with single quotes, such as 'a', 'A', ...

Parser

The parser is a **LL1**[?] (in general, see note) parser that represents the EBNF stated above. Every non-terminal symbol is therefore represented by a function that parses its contents. Error handling consists of two cases (compiling is stopped in both). The parser inserts the expected tokens if they are weak (and can be guessed), such as an ';' or ')'. On unresolvable errors, the parser syncs up on functions (strong token). The printed messages use the line/column counter from the scanner.

The parser also initializes a symbol-table and fills it while parsing the code.

Note: In order to fullfill Go's requirements on providing an extra namescope (which is mandatory) for included libraries the parser uses a look-ahead of 3 in one special case. It consists of checking whether the tokens that follows an "=" indicates a function call or an identifier (and thus an expression). (Example: a.b.c = lib.function(); -> The function call cannot be seen with **LL1**[?]). Since we wanted to stay compatible with Go we introduced this special handling.

Code generation

GoGo is currently able (including the possibility of bugs) to generate code for the following code constructs:

- Arithmetical expressions using addition, subtraction, multiplication and division operators
- Constant and variable field and array access, including pointers to structs (requiring dereferring)
- Arithmetical expressions with the aforementioned operators, together with numbers (constants) and variables (with optional field and array access as mentioned above) as operands
- Assignments with the arithmetical expressions described above on the RHS and the aforementioned variables (with optional field and array access) on the LHS

Additionally, a symbol table has been implemented which distinguishes between local and global variables and is able to handle namespaces (in Go language: package names) on both a global and a local level in order to access variables from other modules.
Moreover, there is basic type checking, including first attempts for different op code sizes based on the operand size (p.e. 1 byte operands for bytes/characters, 8 byte operands for uint64, pointers and address calculation). There are also first attempts to allow additions/subtractions etc. of bytes and uint64s, resulting in a uint64 sum/difference etc.

Run time memory

Libgogo provides a very simple memory manager using a bump pointer which can allocate, but not free memory. By using the `sys_brk` function, the memory manager expands the data segment of the running program in steps of 10 KB if necessary in order to deal with subsequent allocations.
As the Go compiler used for boot strapping uses a custom memory manager in its run time environment, the libgogo memory manager and the **GoGo** compiler take measures to avoid conflicts with the former. First and foremost, all implicit and explicit memory allocations in the **GoGo** compiler rely on the libgogo memory manager in order to keep the amount of memory allocated by the Go run time constant. Additionally, the memory manager does not allocate any memory in the original data segment to not overwrite any string constants or other information stored there by the Go run time. This is achieved by directly expanding the data segment during the initialization of the libgogo memory manager which also allows to store the first address allocated, thus being able to distinguish between memory allocated by the libgogo memory manager and memory allocated by the Go run time.

I/O system

In order to be able to perform I/O operations and memory management, a library called libgogo is implemented which wraps Linux syscalls and provides an easy to use interface to the **GoGo** compiler. As **GoGo** generates assembly code for 64 bit Linux operating systems and the Go compiler allows to mix assembly and Go code, the operating system's built-in functions can be used via syscalls in order to provide the functionality described above.
On Linux 64 bit operating systems with Intel architecture, these syscalls can be invoked by the assembly mnemonic `SYSCALL` where the register `RAX` contains the syscall number defining the syscall, and the registers `RDI`, `RSI` and `RDX` contain the first, second and third parameter respectively.
The following list enumerates the syscalls used by libgogo, together with the value of `RAX` representing the syscall number. The latter were derived from the constants defined in `/usr/src/linux-headers-2.6.32-22/arch/x86/include/asm/unistd_64.h` of the current Linux kernel source. The syscall function prototypes (for semantics and formal parameters) were derived from the corresponding Linux man pages:

- `sys_read` (0) - reads from a file
- `sys_write` (1) - writes to a file
- `sys_open` (2) - opens a file
- `sys_close` (3) - closes a file
- `sys_brk` (12) - see section "Run time memory"
- `sys_exit` (60) - exits the program

Separate compilation

In order to perform separate compilation, a symbol table is generated at the top of each assembly file. This table holds parameters and return values for functions and overall sizes for types. Functions and types (only pointers!) can be forward declared using this approach. The linker then uses this symbol table to fix two cases in which the compiler has not been able to insert the correct stack offsets. Unfortunately the files can not be linked so far because global variables need to be initialized in an assembly init, which also includes static strings. Since offsets at global variables have not been taken into account, linking with files containing global variables and static strings cannot be done. (Type pointer, and calculation work though.)

Until all references are satisfied, a blocking command prohibits the assembler to process the gogo assembly output.