GoGo

A Go compiler written in Go

Michael Lippautz
michael.lippautz@sbg.ac.at

Andreas Unterweger
andreas.unterweger@sbg.ac.at

July 3, 2010

Contents

1 Introduction

GoGo is a self-compiling Go compiler written in Go which implements scanning, parsing and code generation for a subset of the Go language[Goo10]. It is mostly compatible with the Go compiler available in [Goo10] and outputs valid Plan9 assembly code which can be assembled by the latter as well as the Plan9 assembly tools `6a` and `6l`[Pik00]. Lorem ipsum dolor sit amet...

2 Input Language

Go is a programming language developed by Google, based on a C like syntax and fully specified in [Goo10]. The input language follows the one defined by Go. This results in programs being able to be compiled by the official Go compilers and GoGo.

2.1 Differences to Go

1. GoGo only provides only a **very** basic featureset. Expect every advanced and interesting feature to be missing.
2. GoGo forces the usage of semicolons at the end of statements. This restriction was made to make parsing easier.
3. Go is fully Unicode compatible, while GoGo uses ASCII characters only.
4. Simplified expressions, following Wirth's [Wir96] defintions.

2.2 EBNF

Lorem ipsum dolor sit amet...

Atoms

The following listing described the basic atoms that are possible in GoGo programs.

Listing 2.1: Atoms

```
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

Although not as expressive as the ones from Go, these rules define expressions that have comparisons, relations and arithmetical terms.

Listing 2.2: 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 ].
```

Types and Variable Declarations

Listing 2.3: Types

```
type = ([ "[" integer "]" ] identifier | "uint64" | "byte")
    | "string".
var_decl = "var" identifier type [ "=" expression ] ";".
var_decl_list = { var_decl }
```

Structs

Listing 2.4: Structs

```
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 }.
```

Statements

Listing 2.5: Statements

```
package_stmt = "package" identifier ";".
import_stmt = "import" string.
import_stmt_list = { import_stmt }.

stmt_sequence = { stmt }
```

```
stmt = assignment ";" | function_call_stmt ";" | if_stmt
    | for_stmt | ";".

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 "}".
```

Functions

Listing 2.6: Functions

```
expression_list = expression { "," expression }.
function_call = "(" [expression_list] ")".
function_call_stmt = identifier selector function_call.

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)
```

The GoGo Program

Finally, the main program structure is defined by `go_program`. The sequence of the various program parts has been forced to the following to make parsing easier.

Listing 2.7: GoGo Program

```
go_program = package_stmt import_stmt_list struct_decl_list
    var_decl_list func_decl_list.
```

3 Output Language

The output language is Plan-9 assembler [Pik00]. It is a modified version of 64 bit assembly for Intel x86 processors with AT&T syntax that has been created by Bell Labs to be used in their compiler and assembler collection.

3.1 Assembly output

GoGo creates an output file with assembly instructions and comments using mnemonics for op codes and operands/registers which means that it outputs in text form, not in binary form. Therefore, an assembler is needed to process the output in order to make it executable. Like the Go compiler, GoGo relies on `6a` and `6l` of the Plan9 tools in order to acomplish this[Pik00].
The assembly output consists basically of three sections: the data segment, the initialization segment and the code segment. GoGo's assembly output framework provides basic output routines which make it possible to switch between those three segments. Whereas the data segment is used to reserve space for global variables and strings in the data segment, the initialization segment and the code segment contain the code for global variable initialization and the functions from the input, respectively. All other functions (code generation for arithmetical expressions etc.) rely on the assembly output framework which is also able to place comments with the corresponding input file name and line number for debugging purposes in the output file.

4 Scanner / Parser

The scanner is basically the provider of tokens that are used by the parser to interpret the code. In order to generate these tokens the scanner reads the file character by character. If a sequence is known, it converts this sequence of characters into the corresponding token. Tokens generated this way are called simple tokens, as they can be generated right away. For instance, a sequence *'A'* can be directly converted into a token representing a byte value.
Before providing a token to the parser, the scanner may convert such simple tokens one more time. These complex tokens are generated from simple ones that represent `identifiers`. `Identifiers` are compared to a predefined list of keywords. If a keyword matches a token value, the token is converted to the one representing the keyword. Table 4.1 lists some of these tokens.

Simple tokens	`&&`, `+`, `-`, {, }, (,), ...
Complex tokens	`for`, `if`, `else`, `func`, `type`

Table 4.1: Token examples

The scanner also implements a very simple escaping mechanism that allows sequences like *'\n'* to be used in strings.
Comments can be written as */* ... */* blocks or \\ till line ending, like in C/C++.

The parser then takes these tokens and represents the language defined by the EBNF from section 2.2. The parser is basically implemented as LL1 like in [Wir96], with one minor difference. In order to be compatible with Go it was necesarry to include one namespace hierachy that is represented by packages. Since this namescope is prefixed using `package.` the parser needs a lookup of three in this case.

5 Symbol table

In order to be able to lookup local and global variable names as well as function names, a symbol table is required. Based on [Wir96], object and type descriptors were used, each containing the information required for lookup and code generation. Object descriptors are used to store information about variables and parameters whereas type descriptors are used to store information about types and functions.
The following tables 5.1 and 5.2 summarize the fields of the object and type descriptors and their respective purpose. Some fields had to be added in order to support forward declarations and the distinction between values and pointers.

Field	**Type**	**Purpose**
Name	`string`	The object's name
PackageName	`string`	The object's package (Go name space)
Class	`uint64`	The descriptors's kind (variable, field, parameter)
ObjType	`*TypeDesc`	The object's type
PtrType	`uint64`	If 1, the object's type is `*ObjType`; if 0, `ObjType`
Next	`*ObjType`	Next object (linked list)

Table 5.1: ObjectDesc

Field	**Type**	**Purpose**
Name	`string`	The type's/function's name
PackageName	`string`	The type's/function's package (Go name space)
ForwardDecl	`uint64`	If 1, the type/function has not yet been fully delcared/implemented
Form	`uint64`	The descriptor's kind (simple type, array type, struct type, function)
Len	`uint64`	For simple types: type size in bytes, for arrays: array size
Fields	`*ObjDesc`	For struct types: struct fields, for functions: function parameters
Base	`*TypeDesc`	For array types: the array base type
Next	`*TypeDesc`	Next type/function (linked list)

Table 5.2: TypeDesc

For Both symbol tables the build-up and lookup is integrated into the parser. Whenever sufficient information (variable name, function name, type name; optionally preceeded by a package name) is encountered, a lookup in the symbol table is issued. Declarations issue new symbol table entries with the corresponding descriptor properties as described above.
Forward declarations are currently only supported for type pointers (as pointers are always 64 bits in size) and functions (as the affected offsets can be fixed by the linker if necessary). Whenever supported forward declarations are encountered, a new symbol table entry is created with `ForwardDecl` set to 1. Forward declared function can then be called, although forward declared type pointers cannot be dereferred until the size of the type they are pointing to is known (`ForwardDecl` is 0). When forward declared functions are implemented, the corresponding symbol table entry is modified (`ForwardDecl` is set to 0) instead of creating a new one.

5.1 Supported data type

GoGo supports 4 built-in value types and the declaration of new struct and array types as well as pointers to value, struct and array types. The 4 built-in data types are based on the data types supported by the Go compiler and form a minimal subset of them in order to perform basic integer and string operations. The following table 5.3 lists the built-in value types, together with their purpose and size.

Type	**Size**	**Purpose**
`uint64`	8 bytes (64 bits)	Unsigned integer with the target platform's register size
`byte`	1 byte (8 bits)	Single ASCII character or unsigned 8 bit integer value
`string`	16 bytes (see section 6.8)	Character sequences
`bool`	8 bytes (64 bits)	Internal type used for comparisions and jumps

Table 5.3: Built in types

5.2 Local variables and offset calculations

In order to be able to distinguish between parameters, local and global variables, a global and a local symbol table as well the function's parameters as third, virtual symbol table are used. Local variables hide global variables of the same name by performing the symbol table lookup for local variables and parameters first and returning the first match if there is any.
The memory layout for local and global variables as well as parameters is equal: the object's offset address contains the first 64 bits of the object, the next highest address

Address	Content	Source code
`SP-0`	Saved `IP`	
`SP-8`	`a`	`var a uint64;`
`SP-16`	`b`	`var b uint64;`
`SP-24`	`c`	`var c uint64;`
`SP-32`	`s` (higher 8 bytes)	`var s string;`
`SP-40`	`s` (lower 8 bytes)	

(offset address plus 64 bits) contains the next 64 bits etc. All local and global variable offset addresses are 64 bit aligned. The offset of an object can be calculated by summing the aligned sizes of its predecessors in the corresponding variable list. Doing this, it has to be taken into consideration that pointers always occupy 8 bytes (64 bits), regardless of the type they are actually pointing to.
Global variables start at offset 0 of the data segment, referred to as `data+0` in the output. Subsequent global variables use ascending offsets as described above (p.e. referred to as `data+8` for offset 8). Local variables and parameters are addressed relative to the stack pointer `SP`, starting at offset `SP+8` for parameters with ascending offsets as described for global variables (`SP` is reserved for the saved instruction pointer `IP`, see 6.6). Local variables start at offset `SP-8` in descending order (`SP-16` for the second 64 bit variable, `SP-24` for the third etc., see table below) in descending order. Ignoring the sign, the offset relative to SP is still in ascending order, so the offset calculation method as used for global variables and parameters can be used.

As global and local variables as well as parameters share the same offset calculation as described above, they can be treated equally with no change of the offset calculation mechanism. When printing a reference to a variable address in the output, the relative offset does not need to be changed, only the reference address (the beginning of the data segment, `data`, or `SP` respectively) and the offset's sign. This requires the variable's kind (local, global, parameter) to be stored during code generation until both address and offset are needed. This is done by introducing a new field named `Global` to the `Item` type (see next section), indicating whether an object is a global or a local variable or a parameter, respectively.
Offset calculations within types (p.e. calculating the field offsets in a struct or an array index) require a slightly different handling. Global variables as well as parameters can be treated the same way as explained above as their internal offsets' ascending order corresponds to their memory layout (ascending addresses). Local variables require a different calculation as their memory layout (descending addresses) differs. This is necessary in order to be able to assign global to local variables and vice verse so that their internal memory layouts correspond from the programmer's point of view. This is also done by a distinction based on the item's `Global` flag as explained above: during code

generation, the internal offset of a local variable has to be calculated by subtraction instead of addition due to the negative sign of the offset (also see table above).

6 Code generation

GoGo emits assembly code in text form based on the Go input files. This section briefly explains the main features implemented in the code generating functions of GoGo.

6.1 Register allocation

The target architecture provides 8 general purpose registers (`R8-R15`) as well as the registers `RAX`, `RBX`, `RCX` and `RDX`[Int09]. The latter are not being used by GoGo to store variables as their values may change when performing arithmetical operations (p.e. `RAX` and `RDX` are always used as the destination registers for multiplications), thus possibly overwriting values previously stored there.
GoGo stores a list for every one of the 8 registers currently free, returning the first free register if required by the code generator. Whenever a register is no longer required (freed), it will be reinserted into the "free" list in order to make it available for future use. Due to the limited amout of registers, the list described is implemented in form of a bit array in the compiler.

6.2 The generation of arithmetical expressions

As described in [Wir96], code generation for arithmetical expressions basically relies on an operand stack and delayed code generation based on `Items`. For constant operands, constant folding is applied; variable operands are loaded into a free register in order to perform arithmetical operations on them.
GoGo makes use of the capabilities of the target architecture by not loading constants into registers, thus reducing the number of registers required. Consider the expression `a + b` where both `a` and `b` are variables of type `uint64` with negative offsets 8 and 16 relative to the stack pointer. As the target architecture is able to perform an operation like `ADDQ R8, -16(SP)` (add the value at address `SP-16` to the register `R8`), only `a` needs to be loaded into a register, whereas `b` can be directly incorporated into the instruction itself.
Multiplication and division on the target architecture both require special treatment: The multiplication instruction only takes the second operand and requires the first operand to be in the register `RAX`[Int09]. Therefore, the first operand has to be loaded into `RAX` prior multiplication. The multiplication result is stored as 128 bit value in `RDX` (upper 64 bits) and `RAX` (lower 64 bits). As GoGo does not support data types other than `byte` and `uint64`, the upper 64 bits in `RDX` are ignored, and the lower 64 bits are moved to one of the 8 registers to save the result before another multiplication is being

performed. Similarly, division allows for an 128 bit operand (also in `RDX` and `RAX`). As GoGo does not support 128 bit size data types, `RDX` is always being zeroed prior division.
The addition, substraction and multiplication operations are also used for offset calculations. Thus, an additional distinction per `Item` is required in order to be able to distinguish between addresses and values stored in registers. As arithmetical operations on `byte` and `uint64` types always operate on a value, the actual value to be calculated with has to be loaded prior calculation. As offset calculations always require the adress to be loaded into the register instead of the value, it has to be made sure that the address is loaded, not the value. In order to distinguish between addresses and values in registers, the `A` field of the `Item` structure is used. Additionally, the code generation routines for addition and subtraction have an additional parameter specifying whether to calculate with addresses or values, issuing the necessary dereferencing operations if required.

6.3 The generation of assignments

As pointer types are supported, type checks in assignments as well as the assignments themselves get harder to implement as additional cases have to be dealt with. Additionally, the possible occurrence of the address operator (`&`) on the right hand side of an assignment doubles the number of cases. The following table 6.1 illustrates the distinctions made and the code generated for some of the cases allowed by the EBNF (* denotes pointer types, LHS and RHS are the `Items` on the left and right hand side, respectively). For the sake of clearity, only the cases with a non-pointer type variable `Item` on the left hand side and no address operator on the right hand side using `uint64` types are shown. The compiler is also able to assign `byte` values to one another as well as to `uint64` types.

LHS type	**RHS type**	**Code/Error generated**
Variable	Constant	`MOVQ $RHS.A, (LHS.A)`
Variable	Constant*	Type error
Variable	Variable	`MOVQ (RHS.A), R`*temp* `| MOVQ R`*temp*`, (LHS.A)`
Variable	Variable*	Type error
Variable	Register with value	`MOVQ RHS.R, (LHS.A)`
Variable	Register with value*	Type error
Variable	Register with address	`MOVQ (RHS.R), RHS.R | MOVQ RHS.R, (LHS.A)`
Variable	Register with address*	Type error

Table 6.1: Assignment types

String assignments are handled separately as they require 16 bytes to be assigned. As one register can only hold 8 bytes, a second register needs to be allocated in order to perform the assignment. Although this may not be necessary in trivial cases where one string variable is assigned to another, strings in structures which require offset calculations force the use of a register when performing the offset calculation and therefore require a second register to dereference the value of the other 8 bytes. In order to be able to do this, the `C` field of the `Item` structure is being used as it is not needed for other purposes outside conditionals.

6.4 The generation of conditional expressions

Lorem ipsum dolor sit amet...

6.5 The generation of loops

Lorem ipsum dolor sit amet...

6.6 The generation of functions calls

Consider the following example with the current function having 3 local variables – 2 of type `uint64` and one of type `string` – calling the function `foo` which has the following prototype:

```
func foo(param1 uint64, param2 uint64, param3 string) uint64;
```

In order to be compatible to the Go calling convention, the stack is organized as follows before calling `foo`:

Address	Content
`SP-0`	Saved `IP`
`SP-8`	Local variable 1 (uint64)
`SP-16`	Local variable 2 (uint64)
`SP-24`	Local variable 3 (string), higher 8 bytes
`SP-32`	Local variable 3 (string), lower 8 bytes
`SP-40`	Placeholder for return value of foo
`SP-48`	`param3` of `foo`, higher 8 bytes
`SP-56`	`param3` of `foo`, lower 8 bytes
`SP-64`	`param2` of `foo`
`SP-72`	`param1` of `foo`

As can be seen, the parameters are pushed in reverse order and share the address calculation algorithm of local variables (considering the offsets of the local variables), allowing to reuse the latter for this purpose. In order to move the parameters to their corresponding positions on the stack, the offsets are calculated and an assignment statement with the stack offset on the LHS (as variable `Item`) is being performed which allows

to reuse the code for assignment generation, including all type checks and conversions. Optionally, currently used registers are pushed onto the stack between the local variables and the return value, yielding an offset correction to be considered when pushing the parameters.
When calling `foo` in the above example, the `SP` is first decreased by 72, followed by another implicit decrease of 8 due to the `CALL` instruction which pushes the current `IP` onto the stack and decreases the `SP` automatically. The stack is then prepared for `foo` to which it appears similar as to the callee, with the old `IP` at offset `SP`, the parameters on positive offsets relative to `SP` and the local variables (if any) with negative offsets.

Address after call	**Address before call**	**Content**
`SP+40`	`SP-40`	Placeholder for return value of foo
`SP+32`	`SP-48`	`param3` of `foo`, higher 8 bytes
`SP+24`	`SP-56`	`param3` of `foo`, lower 8 bytes
`SP+16`	`SP-64`	`param2` of `foo`
`SP+8`	`SP-72`	`param1` of `foo`
`SP`	–	`IP` of callee
`SP-8`	–	Possible local variable of `foo`

As can be seen, the offsets before and after the call differ by `72+8=80` as explained above. GoGo does not use base or frame pointers, but relies only on the stack pointer to perform all offset calculations. The return value is always located "after" the other parameters and can be used as the RHS of an assignment with its known offset as explained with the parameters above. After the function call, 80 is added to the stack pointer in order to restore the previous stack "view" for the callee. Optionally saved registered are restored considering their additional offset in reverse order.
Internally, a function is represented by a `TypeDesc` (compare table 5.2) where the fields represent the functions parmaeters, including an optional return value at their end. When a function is being called without prior declaration, the parameter types have to be derived from the types of the expressions encountered. As the total number of parameters is unknown before the end function call in the input file, a marker is inserted in the output code and comments to indicate that the final offset has yet to be determined. As the offset due to local variables is known, it is included. Due to the marker which includes information about the name of the function called, the linker is able to adapt the unfinished offsets in order to correct them accordingly.
When functions have been called, but not declared, and then called again, the type check included in the code generation for assignments is already effective and detects improper assignment types. However, it is necessary to extend this check as the first call can p.e. be with a parameter of type `byte` and the second call with a parameter of type `uint64`. This is allowed, as a `byte` type represents a subset of the `uint64` type, requiring a conversion for the former in order to assure that the 7 additional bytes are zero. This can be done by simply extending the type check of the assignment. This also

applies to the return value (as it is treated like a parameter), although it has to be considered that calls of functions without return value assignments do not necessarily mean that the function called has no return value. Therefore, additional checks are necessary, including the check of the return value's existence and type when the actual function declaration is being made.

6.7 Global variable initialization

Besides the compiled functions from the input file, the code segment contains a function called main.init which performs the initialization of global variables. In contrast to local variables which can be initialized directly at point of their declaration, global variables need to be initialized before any other methods are called, thus requiring the main.init function.
Global variables in general are stored in the data segment, called `data` in the output. They are addressed by their corresponding symbol table offsets relative to the beginning of the data segment. For the special treatment of string constants, please refer to the next section.

6.8 String constants

Strings in Go are 16 bytes in size, containing an 8 byte address (pointer) to its character buffer and an 8 byte length (of which only 4 bytes are used). This makes string length calculations unnecessary and also explains why strings in Go are read-only and have to be reallocated when being changed.
Whenever a string constant is found in the input code, a new byte array with the string's length is declared and initialized with the string's characters. Next, another 16 bytes are allocated which represent the actual string. Using the main.init function as described in the previous section, the string's length and the previously allocated byte array address are assigned to the according offsets of the string in the data segment. When assigning the string constant (or using it as a parameter), an item representing a data segment variable with the string's address is used, therefore eliminating the need for any further special treatment.

7 Library and run time

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.

7.1 I/O syscalls

Besides read and write operations to files (and the console), exiting the program as well as opening and closing files requires the use of syscalls. 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[Var08].
The following table 7.1 lists the syscalls used by libgogo, together with the value of `RAX` representing the syscall number. The latter were derived from the C constants defined in `/usr/src/linux-headers-2.6.32-22/arch/x86/include/asm/unistd_64.h` of the current Linux kernel source[Var10]. The syscall function prototypes (for semantics and formal parameters) were derived from the corresponding Linux man pages (see [Var97] and others).

Syscall number (`RAX`)	Syscall function	Purpose
0	`sys_read`	Reads from a file
1	`sys_write`	Writes to a file
2	`sys_open`	Opens a file
3	`sys_close`	Closes a file
12	`sys_brk`	See next section
60	`sys_exit`	Exits the program

Table 7.1: Syscalls

7.2 The memory manager

Libgogo provides a very simple memory manager using a bump pointer which can allocate, but not free memory. By using the `sys_brk`[Var97] 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.

7.3 String memory management

Based on the memory manager described above, functions to copy and append strings are implemented in libgogo. As strings in Go are read-only once they are created (or appended), subsequent appending operands require the string to be entirely copied to a new memory location first. In order to avoid this in most cases, the functions handling string appending in libgogo allocate more memory than needed for the current operation in order to be able to reuse this memory in subsequent append operations if the string appended is short enough to fit in the memory already allocated.
Empirical testing showed that it is most convenient to allocate the next power of two of the string length required (p.e. 16 if 9 bytes are initially required). This reduces the number of copy operations and thus memory consumption by more than a power of 10 for very large strings and small appended string (which is common appending code output).
The string manager also has to take care of strings which have been allocated by the Go run time as those strings don't have any "spare" bytes left. Thus, a reallocation of memory for these strings is necessary and cannot be avoided. Subsequent allocations (after the reallocation) can then be dealt with as described above, leading to the performance gains mentioned. The distinction between strings allocated by the Go run time and the libgogo memory manager can be performed easily by comparing the string's address with the first address available to the libgogo memory manager as described in the previous section. If the string's address is smaller, the string is not yet managed by the libgogo memory manager.

7.4 Program parameter determination

Usually, a program's parameters are accessible through `argc` and `argv`, positioned on the stack as function parameters of the main (entry) function. As the Go compiler used for bootstrapping adds run time routines which are invoked before the main function,

the parameters cannot be fetched from the stack as their position cannot be determined. The Go compiler allows to access these parameters using a separate package (library) which could not be used in order to remain independent of third party libraries.
The current approach to fetch parameters requires the activation of the `proc` file system in the Linux kernel which is enabled by default in most Linux distributions[Var06]. The `proc` file system allows (among other information) to access parameters of all processes running in the system, including the current process. The latter's parameters can be through the virtual file `/proc/self/cmdline` where `self` refers to the current process. The virtual file contains all parameters, separated by zero bytes and terminated by a sequence of two zero bytes. When parsed, this allows to access the program's parameters without having to access the stack.

8 Building / Self-compilation

This section deals with the building process of GoGo and how self-compilation is achieved.

9 Testing

In order to test the compiler, a test suite has been constructed that may be used to verify results against an already existing result set.

The test suite offers the following functions:

- **newvalids/ackvalids/fullclean** – These commands are used to create a new result set as reference for further tests. While `fullclean` deletes the old set, `newvalids` is used to create a new one. After verifying that the compiled output is correct (by manually checking it), the command `ackvalids` can be used to acknowledge the set (resulting in a checksum file).
- **test/clean** – `test` is used to perform a compilation and compare the results against the last valid result set. In order to do so, checksums of the tests are compared. If they are not equal, a `diff` is printed to the user.

Bibliography

[Goo10] Google Inc. The Go Programming Language Specification. `http://golang.org/doc/go_spec.html` (4.6.2010), 2010.

[Int09] Intel Inc. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M. `http://www.intel.com/products/processor/manuals/index.htm` (9.6.2010), 2009.

[Pik00] Pike, R. A Manual for the Plan 9 assembler. `http://doc.cat-v.org/plan_9/4th_edition/papers/asm` (4.6.2010), 2000.

[Var97] Various. brk(2) – Linux man page. `http://linux.die.net/man/2/brk` (9.6.2010), 1997.

[Var06] Various. proc(5) – Linux man page. `http://linux.die.net/man/5/proc` (9.6.2010), 2006.

[Var08] Various. syscalls(2) – Linux man page. `http://linux.die.net/man/2/syscalls` (9.6.2010), 2008.

[Var10] Various. The Linux Kernel. `http://www.kernel.org` (9.6.2010), 2010.

[Wir96] Wirth, N. *Compiler Construction*. Addison-Wesley, 1996.