Go:
A New Systems Language?

SIMON GERBER — SYSTEMS GUY

While you may have heard of Go[1] (the relatively new programming language designed by Google), most of you probably have not yet used it. Here I will try to demonstrate what makes Go unique.

In this article I will try to highlight Go's most interesting features using a fairly small example program (not "Hello, world!", sorry!) which is a systems tool. I am reusing an idea from the Networking/OS lecture in FS08: a server that takes timer requests over TCP and responds after the given time.

Tool-chain setup

To get an impression of the language, a compiler tool-chain is necessary. You can find detailed instructions on Go's homepage[2]. Personally I feel that the Go tools are named curiously: you have `6g/8g/5g` and `6l/8l/5l` (the compiler and linker respectively) for amd64 (prefixed with a 6), x86 (8) and ARM (5). After playing around a bit I wrote a quick script which allows me to compile a Go program using only one command[3]. An alternative would be the `gccgo` front-end for the GNU compiler collection.

The timer server

```
package main
```

Go is structured in packages. These packages work like namespaces in C# but are not hierarchical. Each Go program must have a package "`main`" containing a function `main` which is the entry point of the program.

```
import (
  "net"
  "bufio"
  "strconv"
  "fmt"
  "flag"
  "os"
  "time"
  "strings"
)
```

"`import`" is used to import other packages. Here we are importing various useful packages, such as `net` for networking, `fmt` for `printf` and consorts, `flag` for easy command-line parsing and various others. All these packages belong to Go's standard library and should work on all supported platforms. Fairly good documentation for Go's standard packages can be found at [4]. ➔

```
var port = flag.Int("p", 10000,
  "port to listen on")
```

Using snippets like the above, we can easily define command-line parameters (here "p") with a default value (10000) and a description that is also used by the Go runtime to print a nice usage message.

These flags can then be specified on the command line prefixed with either one or two hyphens and with an equals sign or a space between flag and value. There are predefined flag types for integers, floats, booleans and strings but other custom flag types can be easily defined.

Functions

Function calls are qualified with the package the function is defined in or the variable on which the function is invoked. Function visibility in Go is easy: All functions that start with an upper-case letter are visible outside a package, all others are local to the package.

```
func main() {
  flag.Parse()
  server, err :=
    net.Listen("tcp", ":" +
    strconv.Itoa(*port))
  if server == nil {
    panic("net.Listen: " +
      err.String())
  }
  conns := clientConns(server)
  for {
    go handleConn(<-conns)
  }
}
```

This is the main function for our program. First off, it calls `flag.Parse()` which does all the command-line parsing magic. After that, a TCP connection is opened for listening. Go supports multiple return values, which—among other things—makes error handling easier. Thus you will often see code in the form

```
v, err := someCall()
```

followed by

```
if v == nil { // handle error }
```

The `strconv` package provides all sorts of conversion routines from and to strings. Here we're using `Itoa` which converts a integer to a string.

Another point of interest is the `:=` operator, which simultaneously initializes and declares the left-hand side.

On the last four lines of the main function we see Go's most intriguing concepts: Channels and goroutine invocation.

Channels

Go is designed for parallel computation and borrows the concept of a channel from Hoare's CSP (Communicating Sequential Processes[5]). A channel is basically a type-safe Unix pipe, allowing one process (the producer) to write to it and another process (the consumer) to read from it.

Go Channels can be buffered and unbuffered: An unbuffered channel is synchronous, that is reading from a channel blocks until an item is available and writing to a channel blocks until someone is ready to read from it. A buffered channel has a backing buffer and allows asynchronous reads as long as there are items in the buffer and asynchronous writes as long as there is some free space in the buffer.

➔

ROMY WANTED TO BRING IT STRATEGY TO LIFE. WE HELPED HER DELIVER.

Romy set up a training program for IT management during the implementation of a new operating model. We helped her develop it into a strategic program for department best practices which gave her exposure to IT top management. Read her story at credit-suisse.com/careers

CREDIT SUISSE

Goroutines

Go's main form of concurrent execution units are goroutines, which are small lightweight threads. Starting a goroutine usually costs little more than allocating a stack for it to run on. Goroutines are executed in parallel with other goroutines including their caller. While they do not necessarily run in different threads, mostly a group of goroutines is multiplexed onto a number of threads, to be able to handle blocking events easily.

Syntax and language constructs

Syntax-wise the following points stand out (especially coming from the C world):

- Conditions in conditional statements are not enclosed by parentheses
- Every loop body and `if` block must be enclosed by braces.
- The only loop form is the `for`-loop, which can be used as infinite loop (as above), as "normal" `for`-loop

```
for  idx := 0; idx < 10;
  idx++ { }
```

  and as `while`-loop

```
for condition() { }
```

- The return value(s) of a function are specified after the formal parameter list and look exactly the same syntactically (you can omit the parentheses when there is only a single unnamed return value and completely omit the return value for a function returning `void`).
- Variable and parameter types are specified after the names and multiple variables of the same type can be separated by commas and share the type definition.
- Pointer and array types have the asterisk/brackets before the type name instead of after.
- The keywords `import`, `var`, `const` and `type` introduce declarations, which can be grouped with parentheses.
- Variable types are inferred where possible.
- Usually no semicolons are necessary.

```
func clientConns
  (listener net.Listener)
  (ch chan net.Conn) {
  ch = make(chan net.Conn)
  go func() {
    for {
      client, err :=
        listener.Accept()
      if client == nil {
        fmt.Printf(
          "listener.Accept: " +
          err.String())
        continue
      }
      fmt.Printf("accepted %v\n",
        client.RemoteAddr())
      ch <- client
    }
  }()
  return ch
}
```

This function is called by the `main` function to handle all incoming client connections. First off, a new channel of connections is created. Then a goroutine which accepts client connections and puts them into the channel is started.

Here you can see the use of a function literal as a goroutine. In Go, function literals act as closures and therefore the newly created channel value remains in scope after `clientConns()` returns.

Note the parentheses after the closing brace of the anonymous function: you have to call the function literal in the goroutine invocation.

Data Allocation

Go has two different allocation primitives: `new()` and `make()`. They apply to different types and do different things:

`new(T)` is a built-in function which allocates zeroed storage for a new object of type `T` and returns its address (a value of type `*T`).

`make(T, args)` is another built-in function which serves a different purpose than `new(T)`. It can only create slices, maps and channels and it returns an initialized (not zero) value of type `T`, not `*T`. The reason for this distinction is that all these types are references to data structures that must be initialized to be usable. For example a slice is a triple of values (pointer to data inside an array, length, capacity) and until those three items are initialized the slice is nil.

```
func handleConn
  (client net.Conn) {
  b := bufio.NewReadWriter
    (bufio.NewReader(client),
    bufio.NewWriter(client))
  for {
    fmt.Printf
      ("duration request\n")
    b.WriteString
      ("Enter duration in
       seconds: ")
    b.Flush()
    line, err :=
      b.ReadBytes('\n')
    if err != nil {
      // EOF, or worse
      break
    }
    timer(line, b)
  }
}
```

In the client handler we first encapsulate the client connection using a buffered reader and writer. This gives access to methods to read from and write to the connection without constantly having to manipulate byte arrays. Then we run an infinite loop that terminates when the client sends EOF or worse and which repeatedly asks for a duration and runs a timer with the requested duration. ➔

```
func timer(line []byte,
  b *bufio.ReadWriter) {
  defer un(trace("timer"))
  ns, err := getDurationNs(line)
  if err != nil {
    b.WriteString(
      "Error, aborting: " +
    err.String() + "\n")
```

```
    return
  }
  err = time.Sleep(ns)
  if err != nil {
    b.WriteString("Error: " +
      err.String())
    return
  }
  b.WriteString("BEEP!\n")
}
```

This is the code which actually runs the timer. First we add a nice tracing mechanism to our code for this function: Using Gos `defer` keyword, we specify a function `un(s string)` to run on exit of the function `timer()`. As the arguments of a deferred function are evaluated when the defer statement executes, we execute the function `trace(s string) string` when entering the function `timer()`.

After that we convert the line we got from the client into an integer representing the duration in nanoseconds and use that value in the call to `time.Sleep(ns int64)`. Finally, we send "BEEP!" to the client and return.

```
func getDurationNs(line []byte)
  (ns int64, err os.Error) {
  time, err :=
    strconv.Atoi64(
    strings.Trim(string(line),
    " \t\n\v"))
  if err != nil {
    return 0, err
  }
  ns = time*1000*1000*1000
  return
}
```

Nothing really spectacular happens in this function; the input byte array is converted to a string, leading and trailing white-space is removed using `strings.Trim()` and we try to convert the cleaned string to a 64-bit integer. If this succeeds we convert the input (which is assumed to be in seconds) to nanoseconds and return. On failure we pass on the error from `Atoi64`.

```
func trace(s string) string {
  fmt.Printf("entering %s()\n", s)
  return s
}

func un(s string) {
  fmt.Printf("leaving %s()\n", s)
}
```

You can download the full code at http://pages.vis.ethz.ch/visionen/2011-3/go/.

Types

Go has the familiar `int` and `uint` types which represent values of an appropriate size for a machine word. It has also sized integer types such as `int8` and `uint64`. There is a byte synonym for `uint8` which is the element type for strings. Go also has `float32`, `float64` as well as `complex64` and `complex128` (two `float32` and `float64` respectively). Also, `string` is a built-in type with immutable values—strings are not just arrays of byte values.

Arrays are mutable value types and include the size of the array as well as its element type, which makes talking about pointers to arrays meaningful (and useful) as opposed to C.

There are also slices which can hold a reference to any array with the same element type as the slice. In short slices have the type `[]T`, where an array has the type `[n]T`. ($n \in \mathbb{N}$).

Custom types can be defined using the `type` keyword which works similar to Cs `typedef`. ➔

«Nur eines ist schöner als gute Software, die komplexe Aufgaben ganz einfach löst: Diese Software zu erfinden. Wir freuen uns auf deine Bewerbung.»

Fabian Laubacher, Software Engineer bei BSI

BSI Business Systems Integration AG
www.bsiag.com/zukunft

bsi
meet your clients

OO and Inheritance

The definition of a custom type in Go contains only the data members. Member functions are implemented using the `func (RT) funcName(...)` syntax which implements the member function `funcName` for the type `RT`.

Go has no type hierarchy but every type can implement an arbitrary number of interface types. An interface type is simply a set of method declarations. A variable of an interface type can store any value of any type with a method set which is a superset of the method set of the interface type. Such a type is said to implement the interface[7].

This design also means that Go has no such things as constructors, there's just the `new(T)` built-in which allocates space for a value of type `T` but does not initialize it. Initialization is handled by convention, modules usually provide a public method `New` with returns an initialized value.

To make use of these features (and many others like function literals) Go requires garbage collection and some other supporting runtime code.

Conclusion and personal feeling

I like Go as a language and can see myself writing network services and other inherently concurrent software in Go in the future. I especially (as a long-time Linux user) like the fact that one of the key concepts of the Unix command line (pipes) has made its way into the Go language in the form of channels.

I also like the way the designers of Go decided to have a C-like language with garbage collection and some basic object-oriented features based around the notion of interface types and the possibility of defining methods on any type instead of going full out for a class-based approach à la C++/Java/C#.

Hello, world!

An article about a programming language is not complete without a "Hello, world!" example, so here it is, regardless of what I said in the beginning.

```
package main
import "fmt"
func main() {
  fmt.Println("Hello, world!")
}
```

✦

Links

[1] http://golang.org/

[2] http://golang.org/doc/install.html

[3] http://pages.vis.ethz.ch/visionen/2011-3/go/goc

[4] http://golang.org/pkg/

[5] http://en.wikipedia.org/wiki/Communicating_sequential_processes

[6] http://golang.org/doc/go_tutorial.html

[7] http://golang.org/doc/go_spec.html#InterfaceType

Documentation

Introductory Tutorial:
http://golang.org/doc/go_tutorial.html

Idioms, advanced features:
http://golang.org/doc/effective_go.html