Chapter 5

A short note on Google Go

Google Go (http://golang.org/) is an open source programming language designed to facilitate building reliable, and efficient software. It is a compiled language, statically typed, garbage collected, concurrent and dynamic, originally developed by Ken Thompson, Rob Pike, and Robert Griesemer. Google Go comes with functional and OO features, together with a powerful and light type system.

Compiled, statically-typed languages (like C, C+ +, Java) require too much typing (in the sense of writing code) and too much typing (in the sense of writing explicit types): they tend to be verbose, with lots of repetition. Also they have poor concurrency.

Dynamic languages (like Python, JavaScript) fix some problems (no more types, no compiler) but introduce others: there are many errors at run time that should be caught statically, and no compilation means slower code.

Go tries to take the best of the two worlds: it is compiled to machine code, has static types with some type inference (not full, as in ML), and most of all it has nice concurrency primitives.

The Go project starts at Google in 2007 (by Griesemer, Pike, Thompson), and the first open source release is in November 2009 (more than 250 contributors). Version 1.0 has been released in May 2012 and the current Version 1.4 is from December 2014.

5.1 Concurrency in Google Go

A function can be launched in a separate lightweight thread

```
go f(x)
```

Goroutines run in the same address space and basic synchronization primitives such as mutual exclusion locks are provided by the package sync, but programmers are encouraged to use higher-level synchronization, which is better done via channels and communication. The concurrency primitives of Google Go are inspired by those of the π-calculus. The key idea is:

> *Do not communicate by sharing memory*
> *instead, share memory by communicating*

Channels can be created and then passed to concurrently executed functions:

```
ch = make(chan int) // creates a channel for transmitting integers
ch1 = ch // ch1 and ch access the same channel
go f(ch)
go g(ch)
```

Channels are always created bidirectional, but channel types can be annotated with directionality:

```
var rCh <-chan int // rCh can only be used to receive integers
var sCh chan<- int // sCh can only be used to send integers
rCh = ch // channels can be assigned to unidirectional ones
```

The communication primitives are written:

```
ch <- 2 // sends 2 on ch
v = <- ch // receives from ch and assign value to v
<- ch // receives from ch and throws the value away
```

By default the communication is synchronous: receive and send are blocking. Asynchronous channels can be created by allocating a buffer of fixed size to the channel

```
ch = make(chan int, 100) // creates an async. channel of size 100
```

Receive over an asynchronous channel is of course blocking, but send is blocking only when the buffer is full. Note that there is no dedicated type for asynchronous channels: buffering is a property of values not of types.

Channels can be sent over channels (like in the π-calculus)

```
cch = make(chan chan int)
cch <- ch // sends channel ch over cch
```

Channels can be closed by the sender and the receive can test them for closure[1]

```
close(cch) // closes cch
```

[1] In Google Go functions can return tuples of values.

```
v, ok = <- ch // either value,true or 0,false
```

The select primitive allows to choose between different options.

```
select {
   case x = <- ch1: { ... }
   case y = <- ch2: { ... }
   default: { ... }
}
```

The selection is made pseudo-randomly among the enabled cases. If no case is enabled, then the default choice is selected. If no case is enabled and there is no default option, the select blocks until (at least) one case is enabled.

For example, a non-blocking receive can be written as

```
select {
   case x = <- ch: { ... } // receives on x from ch, if data available
   default: { ... } // otherwise proceeds
}
```