Programmiersprachen im Multicore Zeitalter
Google GO und Nebenläufigkeit

Hendrik Donner

2. Februar 2012

Inhaltsverzeichnis

1 Einleitung

Die vorliegende Seminararbeit beschäftigt sich mit der Programmiersprache GO im Zusammenhang mit Nebenläufigkeit und Parallelität.
GO ist eine von Google entwickelte an C angelehnte systemnahe, imperative Programmiersprache, die nativ Nebenläufigkeit, Garbage Collection, ein statisches Typsystem mit der Syntax dynamisch typisierter Sprachen, und eine bewusst leichtgewichtige Objektorientierung unterstützt. Verzichtet wird auf Typhierarchien und somit auch auf Vererbung. Strukturiert wird GO Quellcode modular durch Pakete ähnlich wie in Java. Die Sprache wurde entwickelt um eine schnell zu Maschinencode kompilierende, vergleichsweise weniger fehleranfällige Konstruktion zu erhalten, mit der man leicht verteiltes Rechnen auf Multicoresystemen erreichen kann. Weiteres Ziel ist es GO zu einer guten Programmiersprache zu Serverprogrammierung, die wenige, einfache Konstrukte nutzt, zu machen. Die Unterstützung von generischer Programmierung wird diskutiert. Eine Standardbibliothek ist in Entwicklung, sehr viele Pakete lassen sich schon produktiv nutzen.
Maßgeblich entwickelt wird GO von Ken Thompson, vorher schon beteiligt an Multics, UNIX, Plan 9, UTF-8, welches in GO für Sourcefiles genutzt wird, und der Vorgängersprache von C B; Rob Pike, ebenfalls an Plan 9 und UTF-8 beteiligt, und Robert Griesemer. Die Entwicklung der Programmiersprache wurde 2007 begonnen, mittlerweile wird GO intern von Google produktiv genutzt.
Wichtigste Informationsquelle zu Google GO ist die Website golang.org, welche auch, da GO noch nicht final spezifiziert ist, die einzig aktuelle Quelle ist. Eine erste finale Spezifikation Go version 1 ist für Frühjahr 2012 angekündigt. Die bislang erhältlichen Veröffentlichungen gelten bereits als stabil.
Momentan existieren zur Entwicklung der Compiler gc, ein Formatierungstool gofmt, ein javadoc ähnliches System namens godoc, ein Tool namens gorun um mit GO Skripte zu schreiben und ein GCC backend von Ian Taylor namens gccgo. Als Debugger wird gdb von beiden Compilern unterstützt.
GO steht unter einer BSD-style Open Source Lizenz, die Dokumentation, Tutorials und Spezifikation zu großen Teilen unter einer Creative Commons Attribution 3.0 License.

2 Kurze Einführung in die Syntax von GO

Im Folgenden möchte ich kurz auf die wichtigsten Sprachelemente von GO eingehen bevor ich zur Nebenläufigkeit komme.

2.1 Hello World

Das obligatorische Hello World in GO:

```
package main

import fmt "fmt" // Paket für printf etc.

func main() {
    fmt.Printf("Hello, world")
}
```

Mit package wird ein Sourcefile einem Paket zugeordnet, mit import werden Pakete eingebunden, func deklariert eine Funktion. Kommentare werden wie in C/C++ mit // für eine Zeile oder über /* ... */ für mehrzeilige Kommentare eingeleitet. Sourcefiles sind UTF-8 codiert, man hätte also auch Hello World in Schwedisch schreiben können:

```
fmt.Printf("hall å världen")
```

Anders als in den meisten anderen Sprachen werden Semikola nur noch bei Auflistungen von mehreren Befehlen genutzt, der Compiler erkennt wenn eine Zeile nur einen Befehl enthält und ergänzt das Semikolon. Dazu muss die Formatierung des Sourcefiles allerdings gewissen Ansprüchen genügen. Das Programm gofmt formatiert ein Sourcefile dementsprechend und wird von den Entwicklern empfohlen.
Sourcefiles enden auf .go, kompiliert werden sie durch einen Aufruf des architekturentsprechenden Compiler: z.B. 6g helloworld.go für 64bit Systeme. 8g kompiliert für 386 und 5g für ARM. Die erzeugten Dateien enden auf die entsprechende Nummer z.B. helloworld.6 und werden mit 6l gelinkt, das Executable ist dann 6.out, für die anderen Zahlen entsprechend.
Alternativ gibt es ein GCC backend gccgo mit dem vom GCC bekannten Verhalten.

2.2 Variablen und Konstanten

Variablen bzw. Konstanten werden folgenderweise deklariert:

```
var sum int
const pi = 3.1415... //ohne Typ, maximale Genauigkeit
var x, y *int  // 2 pointer!
var a [10]int  //array mit 10 Elementen, Länge Teil des Types
```

Für Konstanten stehen 1024 bit zur Verfügung für hohe Genauigkeit.
Möglich sind auch Gruppierungen von mehreren Deklarationen:

```
var (
        pi float64
        gamma float64
)
```

Dies funktioniert auch mit const, func und import.
GO unterstützt auch Strukturen:

```
type Vector3D struct {
        x, y, z float64
}
```

Zuweisungen erfolgen durch =, möglich sind allerdings auch mehrere Zuweisungen gleichzeitig:

```
a,b = b,a //swap
```

Möglich sind auch:

```
var s string = " " //direkte Initialisierung
var s = " "        //kann nur string sein, da " "
s := " "           //s ebenfalls string,
                   //nur in Funktionen erlaubt
```

Typen haben in GO feste Länge wie z.B. float64, uint8; int bzw uint ist 32 oder 64 bit, byte ist ein alias auf uint8, string ist ein nativer Type.

2.3 Kontrollstrukturen

An Schleifen kennt GO nur eine erweiterte for Schleife:

```
//Klammern optional, Codeblock allerdings immer erforderlich!
for i :=0; i<100; i++ {
        ...
}
//Endlos
for {
        ...
}
//wie while
for x < y {
        ...
}
```

```
//für bestimmte Typen wie z.B. map:
var m map[string]int // map[keytype]valuetype
for index, value := range m { //2 Rückgabewerte
        ...
}
```

Dementsprechend sieht if aus:

```
//Das erzwingen des Codeblocks umgeht dangling else
if a < b {
        ...
}
if u < v {
        ...
} else {
        ...
}
```

Darüberhinaus gibt es noch switch und select, welches im Zusammenhang mit Nebenläufigkeit erklärt wird.

2.4 Funktionen

Funktionen werden über das func Schlüsselwort deklariert:

```
//der Typ des Rückgabewerts steht hinter der Parameterliste
func Add(x,y int) int {
        return x+y
}
//Multiple Rückgabewerte
func Sqrt(x float64) (float64, bool) {
        if x < 0 {
                return 0, false
        }
        return math.Sqrt(x), true
}
```

Rückgabewerte sind Variablen, können also benannt werden und man kann ihnen Werte zuweisen. Benannte Rückgabewerte müssen dann nicht explizit nach return angegeben werden, sondern werden automatisch übergeben:

```
func Sqrt(x float64) (y float64, ok bool) {
        if x < 0 {
                y, ok = 0, false
        } else {
                y, ok = math.Sqrt(x), true
        }
        return
}
```

2.5 Methoden

Auf eigenen Datentypen lassen sich Methoden definieren:

```
func (p Point2D) Length() int {
        return math.Sqrt(p.x*p.x + p.y*p.y)
}
...
SomePoint.Length()
```

Darüber hinaus gibt es noch weiter Ansätze objektorientierter Programmierung mit Ausnahme von Vererbung.

3 Nebenläufigkeit mit goroutines und Nachrichtenkanälen

Nebenläufigkeit funktioniert in GO mit Hilfe von goroutines. Eine goroutine ist eine nebenläufig ausgeführte Funktion (oder Methode), die im selben Adressraum läuft wie die anderen goroutines. In einem Prozess laufen eine oder mehrere goroutine, die vom Compiler gc auf Threads gemultiplext werden. Der gccgo Compiler nutzt intern einen Thread für jede goroutine. In beiden Fällen sind goroutines möglichst leichtgewichtig mit kleinen, automatisch wachsenden Stack implementiert und belegen wenig Arbeitsspeicher.
Kommunikation und Synchronisierung erfolgt implizit über typisierte Nachrichtenkanäle.

3.1 goroutine

Goroutines sind nebenläufig ausgeführte Funktionen (oder Methoden) und werden sehr simpel durch das Schlüsselwort go gestartet. Dazu ein vollständiges Beispiel:

```
package main
import (
        fmt "fmt"
        time "time"
)
func Concurrent(thread int, minutes int64) {
    time.Sleep(minutes*60*1e9) // Nanosekunden
    fmt.Println("Thread", thread, "has finished") //Java ähnlich
}
func main() {
    go Concurrent(1, 4)
    go Concurrent(2, 2)
    fmt.Println("Main waiting...")
    time.Sleep(5*60*1e9) // 5 Minuten
}
```

und die dazugehörige Ausgabe:

```
Main waiting...
Thread 2 has finished
Thread 1 has finished
```

Wichtig ist hierbei die Nebenläufigkeit: go macht keine Aussagen oder Garantien über tatsächliche Parallelität, die Implementation ist momentan abhängig vom Compiler. Go-

routines sind deshalb auch zur Strukturierung des Codes bzw. Modellierung des Ablaufs gedacht, selbst wenn keine Hardware mit Parallelität genutzt wird. Sie stellen ein eigenes Programmierparadigma da.
Da goroutines auf Threads gemultiplext werden, kann man beim gc Compiler durch die Umgebungsvariable GOMAXPROCS oder dem Aufruf runtime.GOMAXPROCS(n) die Anzahl der Threads einstellen. Es kann auch mehrere goroutines pro Thread geben. Diese Einstellungen sollen allerdings obsolet werden, wenn der Scheduler der Laufzeitumgebung besser arbeitet. Der gccgo Compiler benutzt wie schon erwähnt einen Thread pro goroutine. Das Konzept ist vergleichbar mit einem Fork/Join Ansatz.
Goroutines haben kleine Stacks, die automatisch gemanagt werden; sind allerdings leichtgewichtiger als echte Threads.

3.2 Nachrichtenkanäle

GOs Motto zu Nebenläufigkeit lautet "Do not communicate by sharing memory. Instead, share memory by communicating."
Dementsprechend ist GOs zwar speichergekoppelt, nutzt aber Kommunikation mit implizierter Synchronisation über einen eigenen Variablentyp, den typisierten Nachrichtenkanal-Variablen:

```
var ch chan int //channel für integer
var chch chan chan int //channel für channel für integer
type message struct {
        data int
        answer chan int
}
msg chan message //channel für eigene Typen
```

Da Nachrichtenkanäle einfache Typen sind kann man auch channel von channel Variablen erzeugen und channel Variablen von beliebigen Typen.
Allokiert werden channel Variablen über die eingebaute make Funktion:

```
ch := make(chan int) //ähnlich wie new, nur kein pointer
```

Nutzen lassen sich channel Variablen über einen eigenen Operator:

```
ch := make(chan int)
//senden
ch <- 1 //1 auf dem channel senden, blockiert bis empfangen wird
ch <- 2 //2 senden
...
//empfangen (sinnvollerweise in einer anderen goroutine)
res := <- ch //res = 1, erst jetzt wird 2 gesendet
```

Synchronisation erfolgt also implizit durch das Blockieren von Kanälen.
Man kann auch bei der Deklaration schon zur feineren Strukturierung eine Richtung angeben:

```
var recv <-chan int //integer channel nur zum empfangen
var send chan<- int //integer channel nur zum senden
```

Der bei for vorgestellte range Befehl funktioniert auch mit Nachrichtenkanälen, man kann also

```
for v := range ch { //... }
```

schreiben. Dadurch werden nacheinander die in den channel gesendeten Werte v zugewiesen.
Schließen lassen sich channel mit der eingebauten Funktion close(chan). Ob ein Nachrichtenkanal geschlossen ist oder noch sendet lässt sich folgendermaßen überprüfen:

```
c := make(chan int)
value, ok := <-c
if ok { //chan offen }
```

3.3 Ein sinnvolleres Beispiel mit goroutines und Kanälen

Ein Producer/Consumer Programm zur Demonstration der Einfachheit von GOs nebenläufiger Programmierung:

```
package main

import fmt "fmt"

func Producer(out chan int, done chan bool) {
        for i:=0; i < 100; i++ {
                fmt.Println("Produced: ", i)
                out <- i //blockiert, bis konsumiert wird
                fmt.Println("Consumed: ", i)
        }
        done <- true
}

func Consumer(in chan int) {
        for {
                <- in
        }
}
func main() {
        ch := make(chan int)
        done := make(chan bool)
        go Producer(ch, done)
        go Consumer(ch)
        go Consumer(ch)
```

```
        <-done //Signal zum beenden, wird verworfen
}
```

und ein Teil der dazugehörigen Ausgabe:

```
Produced:   1
Consumed:   1
...
Produced:   99
Consumed:   99
```

Wie man sieht ist es sehr leicht in GO nebenläufige Programme im Vergleich zu einem Ansatz mit Mutexen/Monitoren/Semaphoren zu entwerfen und zu synchronisieren. Der Verwaltungsaufwand von goroutines ist durch die Kopplung von Synchronisation und Kommunikation quasi nicht vorhanden.

3.4 Buffered channel

Buffered channel-Variablen ermöglichen es mehrere gesendete Daten zu puffern, das heißt also, dass buffered channel erst blockieren wenn der Puffer voll ist. Die Puffergröße gibt man bei der Initialisierung an:

```
ch := make(chan int, 10)//Platz um 10 int zu puffern
```

3.5 Select

Select funktioniert ähnlich wie Switch in anderen Sprachen, nur wählt es zwischen Nachrichtenkanälen aus:

```
ch1, ch2 := make(chan int),make(chan bool)
select {
        case <-ch1:
                fmt.Printf("Received from ch1!\n")
        case <-ch2:
                fmt.Printf("Received from ch2!\n")
        default:
                fmt.Printf("No ch ready!\n)
}
```

Der default Fall trifft ein, falls kein Kanal gerade Daten enthält.

4 Idiomatische Anwendungen

Im Folgenden einige Anwendungsfälle die demonstrieren sollen, wie sich die Entwickler von GO den Einsatz von goroutines und Nachrichtenkanälen vorstellen.

4.1 channel-Factory

Channel-Factories erlauben es, eine Funktion nebenläufig zu starten und gleichzeitig einen passenden channel zur Kommunikation bereitzustellen:

```
func StartServer() chan<- message {
        ch := make(chan message)
        go Server(ch)
        return ch
}
```

4.2 Timeout

Mit dem Timeout-Pattern lässt sich über select das Warten auf einem Kanal zeitlich beschränken:

```
ch, timedout = make(chan int), make(chan bool,1)
//anonyme Funktion
go func (){
        time.Sleep(10*1e9)//10 sek schlafen
        timedout <- true
        }() //Parameterübergabe
//nebenläufige lange Berechnung, die eventuell auf ch sendet
go DoSometing(ch)
select {
        case <-ch:
                fmt.Printf("Received from ch!\n")
        case <-timedout:
                fmt.Printf("Timed out!\n")
}
```

4.3 Pumping

Mit Pumping bezeichnet man das ständige Senden auf einem Kanal:

```
//Random float Generator
for {
        ch <- rand.Float()
}
```

Diese sehr einfache Pattern wird in GO intensiv benutzt, z.B.: für Iteratoren eigener Typen, die range unterstützen: man pumpt einfach fortlaufende Werte eines Containers in einen Ausgabekanal:

```
//Funktion auf einem Beispielcontainer List
func (l *List) Iterator() <-chan ListItem {
    ch := make(chan ListItem);
    go func () {
        for i := 0; i < l.size; i++ {
            ch <- l.items[i]
        }
    } ();
    return ch
}
...
for value := range List.Iterator() {
        ...
}
```

4.4 Futures

Mit Futures bezeichnet man rechenintensive Aufgaben, die man Hintergrund vorausberechnen lässt, um das Ergebnis wenn es gebraucht wird bereit zu haben:

```
func FutureAdd(x, y int) <-chan int {
        ch := make(chan int)
        go func() {
                ch <- (x+y)
        }()
        return ch
}
...
a := FutureAdd(1,2)
b := FutureAdd(3,4)
... //irgendetwas Anderes berechnen
res <- a //Resultat abholen
```

Futures lassen sich also sehr einfach mit Hilfe von Factories realisieren.

4.5 Chaining

Chaining schaltet mehrere Kanäle hintereinander, was sehr einfache Filtersysteme ermöglicht:

```
func filter(filt int, in chan int) chan int {
        ch := make(chan int)
        go func(){
                for {
                        if i := <-in; i % filt == 0 {
                                ch <- i
                        }
                }
        }()
        return ch
}
...
div2 := filter(2, in)
div2or3 := filter(3, div2)
```

4.6 Parallel for

Mit Hilfe von Kanälen ist es auch einfach parallele for Schleifen zu erzeugen:

```
done := make(chan int)
for index,value := range array {
    go func (index int, value data) {
        res[index] = doSomething(value)
        done <- index //res[index] fertig
    } (index, value)
}
```

4.7 Pseudo-Semaphor

Mit GOs Nachrichtenkanälen ist es auch möglich Pseudo-Semaphore zu erzeugen, auch wenn dies nicht die bevorzugte Art ist, Kanäle zu nutzen:

```
ch := make(chan int)
...
func Worker() {
        ch <- 1 //channel voll,
                //nächster Worker blockiert hier
        ...
        <-ch //channel leeren,
             //nächster Worker darf in den Bereich
}
```

Sinnvoller ist da der Einsatz von Read/Write-Semaphoren aus GOs Standardbibliothek, welche gleichzeitige Lesevorgänge erlaubt und Lese/Schreib- bzw. Schreib/Schreib-Konflikte lösbar macht. Damit lässt sich vergleichbar zu pthreads in C arbeiten.

4.8 Multiplexing

Ein nützliches Pattern ist es auch, Datenstrukturen zu senden, die intern einen Nachrichtenkanal enthalten:

```
type Request struct{
        req String
        reply chan String
}
func Server(in chan Request) {
        for {
                req := <- in
                req.reply <- Compute(req.req)
        }
}
```

Damit erhält jeder Client seinen eigenen Antwortkanal.

4.9 Netchan

Mit Hilfe des Pakets netchan aus der Standardbibliothek lassen sich Nachrichtenkanäle auch über ein Netzwerk nutzen. Damit ist also auch Interprozesskommunikation möglich. Die Kanäle verhalten sich dabei wie gepuffert mit einer Puffergröße von mindestens 1. Das Paket ist allerdings noch experimentell. Das Prinzip der netchan basiert auf dem Exportieren/Importieren von Nachrichtenkanälen über eine Netzwerkverbindung anhand eines Namens:

```
import "netchan"

exporter := netchan.NewExporter()
exporter.ListenAndServe("tcp",":42") //tcp auf Port 42
In := make(chan string)
Out := make(chan string)
exporter.Export("in", In, netchan.Send) //zum Senden
exporter.Export("out", Out, netchan.Recv) //zum Empfangen

//In einem anderen Prozess importieren
import(
        "netchan"
        "os"    //für error
)
```

```
importer, err := NewImporter("tcp", ":42")
In := make(chan string)
Out := make(chan string)
//zum empfangen, buffersize 1
importer.Import("in", In, netchan.Recv, 1)
//zum senden, buffersize 1
importer.Import("out", Out, netchan.Send, 1)
```

Die so exportiert/importierten Nachrichtenkanäle lassen sich dann genauso wie normale Kanäle nutzen, mit der Ausnahme, dass sich über Kanäle keine Kanäle verschicken lassen, da jeder Kanal exportiert/importiert werden muss. Eine Lösung wird diskutiert, die Exporter/Importer könnten dann in der Zukunft über Nachrichtenkanäle verschickte channel Variablen automatisch exportieren und importieren.

5 Die Inspirationsquelle hinter GOs Nebenläufigkeitsmodell - CSP

Google GOs Nebenläufgkeitsmodell ist maßgeblich durch C. A. R. Hoare's Communicating Sequential Processes (kurz CSP), eine Prozessalgebra, die auch die Interaktion zwischen Prozessen abbilden kann, inspiriert. Im Folgenden wird CSP kurz erläutert.
In CSP sind Prozesse als Abfolge von Ereignissen definiert, die nacheinander auftreten. Ein spezielles Ereignis ist STOPP, welches einen Prozess beendet. Dargestellt werden Prozesse durch die Abfolge der Ereignis mit Richtungspfeilen [1]:

$$Start \rightarrow STOPP$$

Brauchbare Prozesse entstehen durch Rekursion, dabei muss der Prozess benannt werden:

$$P = e \rightarrow P$$

Der obige Prozess entspricht einer endlosen Folge des Ereignisses e.
Die möglichen Ereignisse eines Prozesses nennt man sein Alphabet, auch αP.
Eine nichtdeterministische Auswahl eines Ereignis erreicht man mit dem Operator |:

$$Q = (e \rightarrow Q)|(f \rightarrow Q)$$

Die Ereignisse e und f werden quasi zufällig gewählt, aber immer richtig, vergleichbar wie bei nichtdeterministischen Automaten aus der theoretischen Informatik.
Prozesse lassen sich nebenläufig auswerten mit dem || Operator, dabei unterscheidet sich das Verhalten bei Prozessen mit demselben oder verschiedenen Alphabeten: sollten sie dasselbe Alphabet besitzen, müssen sie mit demselben Ereignis beginnen, sonst entsteht ein Deadlock:

$$C = (c \rightarrow A)||(c \rightarrow B)$$
$$C = (c \rightarrow A)||(d \rightarrow D) = STOPP$$

Sollten die Alphabete verschieden sein, so müssen Ereignisse, die in beiden Alphabeten enthalten sind, gleichzeitig erfolgen, alle anderen Ereignisse können unabhängig von einander ablaufen.
Die so erhaltenen Prozesse lassen sich prinzipiell in sequentielle Prozesse umformulieren,

[1] Alle folgenden Definitionen zu Prozessen und Operatoren stammen aus Communicating Sequential Processes, Hoare 2004; auf weitere Einzelnachweise diesbezüglich wird im Laufenden verzichtet.

in dem man etwa alle möglichen Abfolgen von Ereignissen in einem neuen Prozess mit Hilfe des | Operators zusammenfasst [2]:

$$\begin{aligned} P &= (up \rightarrow down \rightarrow P) \\ Q &= (right \rightarrow left \rightarrow Q | left \rightarrow right \rightarrow Q) \\ & P||Q \end{aligned}$$

lässt sich ersetzen durch:

$$\begin{aligned} P||Q &= R_{12} \\ R_{21} &= (down \rightarrow R_{11} | right \rightarrow R_{22}) \\ R_{11} &= (up \rightarrow R_{21} | right \rightarrow R_{12}) \\ R_{22} &= (down \rightarrow R_{12} | left \rightarrow R_{21} | right \rightarrow R_{23}) \\ R_{12} &= (up \rightarrow R_{22} | left \rightarrow R_{11} | right \rightarrow R_{13}) \\ R_{23} &= (down \rightarrow R_{13} | left \rightarrow R_{22}) \\ R_{13} &= (up \rightarrow R_{23} | left \rightarrow R_{12}) \end{aligned}$$

Kommunizieren können Prozesse über Kanäle, dabei ist eine Kommunikation ein Ereignis c.v, wobei c der Name des Kanals und v der gesendete Wert ist. Die Ereignisse c.v bilden ein eigenes Alphabet, das Alphabet der Werte, die ein Prozess auf einem Kanal senden kann. Gesendet wird mit Hilfe des ! Operators, empfangen mit dem ? Operator:

$$\begin{aligned} (c!v \rightarrow P) &= (c.v \rightarrow P) \\ (c?x \rightarrow P(x)) &= (y : \{y | channel(y) = c\} \rightarrow P(message(y))) \end{aligned}$$

wobei channel(c.v) = c, message(c.v) = v liefert. Das senden entspricht also dem spezifischen Ereignis c.v, empfangen einem der möglichen Ereignisse des Alphabet der Ereignisse c.v.
Kommunikation erfolgt dann, wenn ein Prozess auf einem Kanal sendet, während ein Anderer auf demselben Kanal gleichzeitig empfängt. Das Alphabet der Ereignisse des Kanals ist dann für beide Prozesse gleich.
Prozesse, die auf ausschließlich einem Kanal empfangen und auf ausschließlich einem Anderen senden werden als Pipes bezeichnet:

$$PIPE = (left?x \rightarrow right!x \rightarrow PIPE)$$

Benutzt werden dann reine Empfangs- bzw. Sendekanäle. Werden Pipes aneinander gefügt, schreibt man auch:

$$P >> Q$$

[2]Das folgende Beispiel stammt aus Communicating Sequential Processes, Hoare 2004, S.50f

für 2 Pipes P und Q.
CSP enthält noch wesentlich mehr, unter anderem auch LISP Implementierungen der Operatoren und Funktionen; der gezeigte Teil enthält allerdings schon alle Elemente von GOs Nebenläufigkeitsmodell: die goroutines in Verbindung mit der Nutzung von Nachrichtenkanälen spiegeln die Hauptinnovation von CSP - sequentielle Prozesse durch Kanäle zu verbinden - wider.

6 Fazit

GOs Nebenläufigkeit lässt sich einfach benutzen und fügt sich in den Code natürlich ein. Durch die Speicherkopplung bleibt der Overhead vergleichbar klein, vermeidet aber die sonst nötige Verwaltung mit Hilfe von Mutexen und Conditions. Der Ansatz ähnelt dem einfacheren Send/Receive nachrichtengekoppelter Ansätze, vermeidet allerdings Marshalling bzw. das komplexe Exportieren eigener Datentypen. Problematisch könnte sein, dass man channel Variablen nicht ansieht ob sie gepuffert oder ungepuffert sind. Auf der anderen Seite sollte ein Programm robust genug sein um in beiden Fällen zu funktionieren.
Die Inspiration durch CSP ist allerdings nicht neu, die ersten Formulierungen von CSP glichen selbst einer Programmiersprache und mit occam gibt es seit 1983 eine ernstzunehmende Sprache mit Nachrichtenkanälen. Möglich ist aber, dass Google mit GO diesem Konzept den Durchbruch ermöglicht, den weit angewandt wird ein Konzept basierend auf Nachrichtenkanälen heute noch nicht.
Syntaktisch wird Nebenläufigkeit einfach durch das Schlüsselwort go, einem eigenen Variablentyp zum verschicken von Nachrichten samt passenden <- Operator realisiert. Wer unter UNIX "ps -e | grep net" versteht wird sofort mit GO arbeiten können. Die Ähnlichkeit zu Unix pipes ist allerdings laut den Entwicklern von GO zufällig, Unix pipes und GOs Nachrichtenkanäle ähneln vielmehr beide der Pipe Definition von CSP.

Literaturverzeichnis

[1] C. A. R. Hoare. Ed.: Jim Davies. *Communicating Sequential Processes*, Prentice Hall International, 2004

[2] *golang.org*, http://golang.org, [Abruf: 2.2.2012]

[3] *golang-nuts Mailing List*, http://groups.google.com/group/golang-nuts, [Abruf: 2.2.2012]