

Hochschule Luzern
Departement für Informatik

PCP Projekt - Programmiersprachanalyse

Analyse der Programmiersprache GO

Studierende: Frederico Fischer, Oliver Werlen
Dozenten: Prof. Dr. Ruedi Arnold, Marcel Baumann
Abgabedatum: 26. Mai 2021

1 Einleitung

In dieser Projektarbeit wurde die Programmiersprache **Go** analysiert. Es handelt es sich, wie auch bei vielen anderen Sprachen, um eine Multiparadigmen Sprache (funktional, imperativ und objektorientiert). Dabei wurde Bezug auf dessen Entstehungsgeschichte sowie Spezialitäten und Feinheiten genommen. Motivierungsgründe für die Wahl dieser Sprache waren, dass es sich dabei um eine sehr junge und oft verwendete Sprache handelt, die bei vielen neuen Anwendungen genutzt wird und eine hohe Popularität aufweist.

2 Vision, Geschichte & Verbreitung

Die Sprache „Go“ ist durch Google von den Entwicklern Robert Griesemer, Rob Pike und Ken Thompson entstanden. Gründe für dessen Entstehung waren die Beseitigung der Langsam- sowie Schwerfälligkeiten und die Erhöhung der Produktivität und Skalierbarkeit ihrer Prozesse. Das Zielpublikum dieser Sprache sind insbesondere Software Entwickler, welche sich mit grossen Softwaresystemen auseinandersetzen. Diese Sprache zeichnet sich durch seine Performance, Lesbarkeit, unkomplizierte Einbindung von Abhängigkeiten sowie simple Nebenläufigkeitsimplementierung aus. Insbesondere in der DevOps-Gemeinde wurde diese sehr beliebt. Beispielsweise wurde Kubernetes in dieser Sprache entwickelt

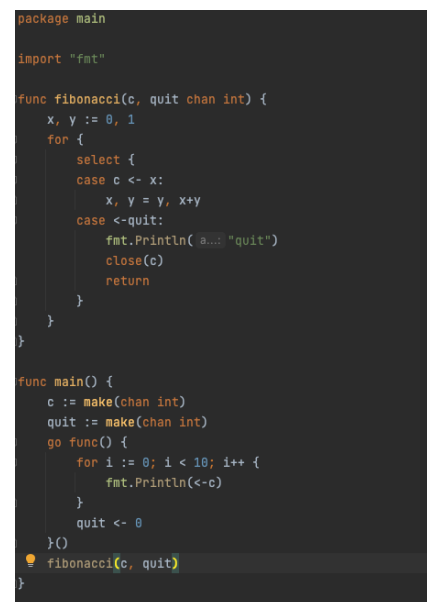
3 Sprachkonstrukte

3.1 Goroutines, Channels & Select

Goroutines Eine Goroutine ist ein leichtgewichtiger Thread, welcher von der Go runtime gemanaged wird. Goroutines nutzen dabei den selben Adressraum, daher muss der Zugriff auf geteilte Ressourcen synchronisiert werden. In Go gibt es Primitives, welche die Synchronisierung übernehmen. Jedoch werden diese nur selten genutzt, da in den meisten Fällen mit Channels gearbeitet wird.

Channels Ein Channel erlaubt einen einfachen Datenfluss. Per Default ist dabei das Senden und Empfangen blockierend, bis die andere Seite bereit ist. Goroutines lassen sich somit sehr leicht synchronisieren, ohne den Einsatz von Locks oder Variablen. Der Channel kann explizit vom Sender geschlossen werden. Damit wird dem Empfänger signalisiert, dass keine Werte mehr empfangen werden können. Das Schliessen des Channels sollte dabei exklusiv vom Sender ausgeführt werden. Senden auf einen geschlossenen Channel verursacht dabei "panic".

Select Bei einem Select wird bei mehrfacher Auswahl gewartet, bis eine Operation laufen kann.



```
package main

import "fmt"

func fibonacci(c, quit chan int) {
    x, y := 0, 1
    for {
        select {
            case c <- x:
                x, y = y, x+y
            case <-quit:
                fmt.Println("quit")
                close(c)
                return
        }
    }
}

func main() {
    c := make(chan int)
    quit := make(chan int)
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(<-c)
        }
        quit <- 0
    }()
    fibonacci(c, quit)
}
```

Abbildung 1: Beispiel Goroutine, Channel und Select

Beispiel In der Abbildung 1 werden alle oben genannten Sprachkonstrukte genutzt. Bei `func()` handelt es sich um eine Goroutine, welche 10 Werte von dem Channel `c` liest und anschliessend einen Wert auf den Quit-Channel schreibt. Das Select differenziert dabei zwischen den beiden Channels. Je nach Case wird entweder eine neue Fibonacci-Zahl berechnet oder der Channel geschlossen.

3.2 Maps & Slices

Slices Bei Slices handelt es sich um ein dynamisches Array welches speziell für Go implementiert wurde und als Datenstruktur in der Standardbibliothek mitenthalten ist. Im Untergrund besteht ein Slice aus einem Array weshalb es auch Funktionen eines Arrays mitliefert. Ein wichtiger Unterschied ist, dass ein Array ein Value-Typ ist, wohingegen ein Slice ein Referenz-Typ. Bei der Instanzierung eines Slices kann dessen Länge sowie Kapazität angegeben werden. Die Länge bezeichnet die Anzahl Elemente welches ein Slice enthält und die Kapazität die Anzahl Elemente welche sich im unterliegenden Array befinden.

```
s := []int{2, 3, 5, 7, 11, 13}
printSlice(s) // len=6 cap=6 [2 3 5 7 11 13]

// Slice the slice to give it zero length.
s = s[:0]
printSlice(s) // len=0 cap=6 []

// Extend its length.
s = s[:4]
printSlice(s) // len=4 cap=6 [2 3 5 7]

// Drop its first two values.
s = s[2:]
printSlice(s) // len=2 cap=4 [5 7]

s = s[:5] // panic: runtime error:
printSlice(s) // slice bounds out of range [:5] with capacity 4
```

Abbildung 2: Verhalten von Slices, Quelle Autor

```
foods := map[string]interface{}{
    "bacon": "delicious",
    "eggs": struct {
        source string
        price  float64
    }{ source: "chicken", price: 1.75},
    "steak": true,
    "waterBottles": 2,
    "myFile": os.Create( name: "myFile.txt"),
}
```

Abbildung 3: Beispiel einer Map, Quelle Autor

Maps Hierbei handelt es sich um eine Key-Value-Datenstruktur, welche in anderen Sprache unter HashMaps bekannt ist. Typen von Keys müssen comparable sein, weshalb Funktionen und Slices nicht genommen werden können. Eine spezielle Eigenschaft dieser Map ist der Value-Type **Interface**. Damit steht als Value dem Entwickler offen was für Typen dieser dort nutzen will. Es kann eine Mischung aus Structs, Strings, Arrays, Pointers etc. sein. Diese Eigenschaft ist nützlich, wenn bspw. das Schema vom erhaltenen JSON nicht bekannt ist und in einer Map gespeichert werden soll. `map interface!`

3.3 Structural & Nominal Typing

Erklärung und Codebeispiel

3.4 The Go Memory Model

```

package MemoryModel

var p *int

func main() {
    done := make(chan bool)
    // "done"-Variable wird in der Main-Methode
    // sowie in der Goroutine verwendet, somit
    // landet dieser im Heap

    go func() {
        x, y, z := 123, 456, 789
        _ = z // z kann auf dem Stack abgelegt werden
        p = &x // x und y landen auf dem Heap, weil sie
        p = &y // über das globale p referenziert werden

        // x wird nun nicht mehr referenziert
        // und kann durch den Garbage Collector
        // aufgesammelt werden

        p = nil
        // y wird nun auch nicht mehr referenziert
        // und kann durch den Garbage Collector
        // gesammelt werden

        done <- true
    }()

    <-done
}

```

Abbildung 4: Beispiel vom Memory Management, Quelle Autor

Die Sprache **Go** liefert ein automatisches Memory Management mit wie zum Beispiel automatische Memoryallozierung oder den Garbage Collector. Dieser arbeitet dabei mit Memory Blocks, welche Values beinhalten und unterschiedlich gross sein können. Memory Blocks entstehen bspw. durch den Aufruf von **new** (nur einen auf dem Heap oder Stack) oder **make** (mehrere auf dem Heap), Variablendeklarationen etc. Jede goroutine verwaltet einen Stack (Memory Segment). Dabei handelt es sich um einen Memory Pool von Memory Blocks welche alloziert werden können. Memory Blocks, welche durch eine Goroutine auf einen Stack alloziert werden, sind nur in dieser Goroutine intern ersichtlich. Darin müssen keine Synchronisationsarbeiten vorgenommen werden. Beim Heap handelt es sich um ein Singleton. Wenn ein Memory Block nicht in einer Goroutine Stack alloziert ist, landet er im Heap. Solche Memory Blöcke können durch mehrere Goroutine verwendet werden, weshalb die Synchronisierung beachtet werden muss. Wenn der

Compiler entdeckt, dass ein Memory Block über mehrere Goroutines hinweg verwendet wird oder sich nicht sicher ist ob es auf den Stack soll, wird dieser im Heap abgelegt. Die Vorteile Memory Blocks auf dem Stack abzulegen sind, dass die Allokierung schneller ist, diese nicht durch den Garbage Collector bereinigt werden müssen und sie CPU Cache-freundlicher sind.

3.5 Package Management

Erklärung und Codebeispiel

3.6 Defer

Erklärung und Codebeispiel

4 Fazit

4.1 Team-Fazit

4.2 Fazit Frederico Fischer

4.3 Fazit Oliver Werlen