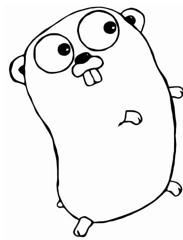
Arbeitsgruppe Programmiersprachen und Übersetzerkonstruktion
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Seminararbeit

Go



Autor: Finn Teegen
Datum: 22. Februar 2013
Betreuer: Björn Peemöller

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Geschichte . . . 1
1.2 Ziele . . . 1
1.3 Beeinflussende Sprachen . . . 2
1.4 Paradigmen . . . 2
1.5 Anwendungsgebiete . . . 2

2 Struktur und Konzepte **3**
2.1 Packages . . . 3
2.2 Datentypen . . . 4
2.3 Variablen und Konstanten . . . 5
2.4 Kontrollstrukturen . . . 6
2.4.1 Verzweigungen . . . 6
2.4.2 Schleifen . . . 7
2.5 Funktionen und Methoden . . . 8
2.6 Interfaces . . . 9
2.7 Zeiger . . . 10
2.8 Arrays und Slices . . . 11
2.9 Maps . . . 12
2.10 Goroutinen und Channels . . . 13
2.11 Reflexion . . . 15

3 Technische Unterstützung **16**
3.1 Compiler . . . 16
3.2 Werkzeuge und Entwicklungsumgebungen . . . 16
3.3 Bibliotheken . . . 16

4 Effizienz **17**
4.1 Geschwindigkeitsvergleich am Beispiel von Quicksort . . . 17
4.2 Diskussion der Ergebnisse . . . 18

5 Fazit **19**

Quellenverzeichnis **20**

Anhang **21**

1 Einleitung

1.1 Geschichte

Die Entwicklung von Go begann im September 2007 bei Google, wo Robert Griesemer, Rob Pike und Ken Thompson an einem Whiteboard ihre Ideen für eine neue Programmiersprache diskutierten.

Rob Pike ist für die Entwicklung der Programmiersprache Limbo bekannt, während Ken Thompson der Erfinder der Programmiersprache B, dem Vorgänger zu C, ist und zusammen mit Dennis Ritchie das Betriebssystem Unix entwickelt hat. Außerdem haben beide gemeinsam den Zeichenkodierungsstandard UTF-8 entwickelt und arbeiten an Plan 9, einem experimentellem Nachfolger des Betriebssystems Unix.

Ursprünglich als ein sogenanntes 20-Prozent-Projekt parallel zur Arbeit begonnen, avancierte Go bald zu einem Vollzeitprojekt, dem sich weitere Mitarbeiter anschlossen, die die Entwicklung maßgeblich vorantrieben. So existierte Anfang des Jahres 2008 bereits ein erster Prototyp eines Compilers zur Erprobung der Ideen und Mitte desselben Jahres begann die Entwicklung eines Front-Ends für die GNU Compiler Collection.

Öffentlich vorgestellt und gleichzeitig unter der BSD-Lizenz als Open-Source-Projekt veröffentlicht wurde Go schließlich im November 2009. Seitdem haben viele Personen zur Weiterentwicklung der Sprache beigetragen, sei es durch das Einbringen eigener Ideen, Beteiligung an Diskussionen oder Programmierarbeit.

Ein weiterer Meilenstein war dann im März 2012 das Erscheinen der Version 1 von Go. Für diese Version geschriebene Quelltexte sollen für lange Zeit auch mit zukünftigen Versionen kompatibel und ohne Änderungen zu kompilieren sein. Somit eignet sich Go 1 für den produktiven Einsatz und stellt eine gesicherte Grundlage für die weitere Entwicklung der Programmiersprache dar.

1.2 Ziele

Go wurde aus der Unzufriedenheit mit existierenden Sprachen heraus geboren, da es nach Ansicht der Erfinder keine Sprache zur Systemprogrammierung gab, die der Weiterentwicklung im Bereich der Computersysteme gerecht wurde. So sollte Go von Haus aus Unterstützung für Mehrkern- bzw. Mehrprozessor- und Netzwerkprogrammierung mitbringen, ein Aspekt, der in der heutigen Zeit signifikant an Bedeutung gewinnt.

Außerdem wurde versucht, die Einfachheit bei der Programmierung mit interpretierten, dynamisch-getypten Sprachen wie Python oder JavaScript mit der Effizienz und Sicherheit von kompilierten, statisch-typisierten Sprachen wie C zu vereinen. All dies schlug sich schließlich in den folgenden Zielen für Go nieder:

- Typ- und Speichersicherheit
- Unterstützung für Nebenläufigkeit und Kommunikation
- Hohe Effizienz beim Kompilieren und Ausführen
- Automatische Garbage Collection

1.3 Beeinflussende Sprachen

Go orientiert sich in vielen Punkten sehr stark an C, wovon in großen Teilen die Syntax, die Datentypen und Zeiger übernommen wurden. Außerdem lassen sich auch Elemente von Sprachen wie Pascal und Modula finden, zum Beispiel die Deklarationen und die Unterteilung in Packages.

Weiterhin beeinflussten Newsqueak und Limbo, beides Programmiersprachen, an denen ebenfalls Rob Pike beteiligt war, Go maßgeblich. So wurde in beiden Sprachen das von Tony Hoare entwickelte Konzept der *Communicating Sequential Processes*, das die Interaktion zwischen kommunizierenden Prozessen beschreibt, umgesetzt. In Go findet dieses durch Goroutinen und Channels Anwendung.

Mit der Garbage Collection wurde zudem ein Prinzip aufgenommen, das in vielen anderen Sprachen ebenfalls umgesetzt wurde, wobei Java sicherlich einen der prominentesten Vertreter darstellt.

Auch Python hatte einen gewissen Einfluss auf Go. So wurde das Konzept der Slices und Funktionen mit mehreren Rückgabewerten aus dieser Sprache übernommen.

1.4 Paradigmen

Go ist eine strukturierte, imperative, typsichere, nebenläufige Sprache, die darüber hinaus auch Aspekte der funktionalen (anonyme Funktionen und Funktionen höherer Ordnung) und objektorientierten (Interfaces und Methoden) Programmierung bietet.

1.5 Anwendungsgebiete

Wie bereits erwähnt, wurde Go als Sprache zur Systemprogrammierung konzipiert und eignet sich dementsprechend für systemnahe und effiziente Programmierung. Den Zielen (s. Abschnitt 1.2) entsprechend bietet sich insbesondere ein Einsatz auf verteilten Systemen an. Auch Serveranwendungen sind eines der ursprünglich angestrebten Anwendungsgebiete. So ist der Webserver der zu Go gehörigen Website golang.org in Go geschrieben. Zudem wird Go von der Google App Engine untersützt, einer Plattform zum Entwickeln und Hosten von Webanwendungen auf den Servern von Google.

Dennoch sind die Anwendungsmöglickeiten nicht nur darauf beschränkt, sodass Go letztendlich als eine *General Purpose Language* angesehen werden kann. Da Go noch eine recht junge Sprache ist, bleibt wohl abzuwarten, welche weiteren Anwendungsgebiete in der Praxis erschlossen werden.

2 Struktur und Konzepte

Quelltextdateien in Go sind grundsätzlich in UTF-8 kodiert, sodass Zeichen wie π oder Ω gültige Variablenbezeichner darstellen. Wie bereits in der Einleitung erläutert, lehnt sich die Syntax an die von C an, so auch der Kommentarstil. Dagegen sind Semikola am Ende von Zeilen optional und werden in der Regel weggelassen.

2.1 Packages

Go-Programme sind in Packages organisiert, die zumeist eine Menge von Funktionen, Typen, Konstanten und Variablen beinhalten. Diese können wiederum von anderen Packages genutzt werden. Zu welchem Package eine Datei gehört, wird am Beginn der Datei durch das Schlüsselwort **`package`** spezifiziert. Das bedeutet insbesondere, dass eventuell auch mehrere Dateien zu einem Package gehören können.

Für ausführbare Programme ist der Packagename mit `main` fest vorgegeben. Dies gilt ebenso für die Hauptfunktion, die ebenfalls den Namen `main` tragen muss. Sie darf nur einmal vorhanden sein und hat weder Argumente noch einen Rückgabewert.

Wird in einem Package die Funktionalität eines anderen Packages benötigt, so kann es über das Schlüsselwort **`import`**, gefolgt vom Namen des jeweiligen Packages in Anführungszeichen, eingebunden werden. Wenn das zu importierende Package nicht unter dem Standardpfad für Importe zu finden ist, so ist eine entsprechende Pfadangabe zu ergänzen. Durch die Anweisung `import "fmt"` wird beispielsweise das Package `fmt` importiert, welches verschiedene Funktionen für die formatierte Ein- und Ausgabe bereitstellt. Über den Packagenamen gefolgt von einem Punkt können darin enthaltene Funktionen etc. verwendet werden. So erzeugt `fmt.Println("Hello␣World")` zum Beispiel eine Ausgabe auf der Konsole, indem die Funktion `Println` der Packages `fmt` benutzt wird. Für eine bessere Übersicht können mehrere Importe auch auf die folgende Art und Weise zusammengefasst werden:

```
import (
    "fmt"
    "math"
)
```

Quelltext 2.1: Zusammenfassen mehrerer Importe

Im Gegensatz zu C gibt es in Go keine Trennung von Header- und Source-Dateien. Die Deklaration und Implementierung erfolgt an derselben Stelle und die Festlegung, was nach außen hin sichtbar ist, geschieht über die Groß- und Kleinschreibung: Beginnt die jeweilige Bezeichnung mit einem Großbuchstaben, so bedeutet dies, dass andere Packages sie sehen und verwenden können:

```
var pi = 3.14159    // wird nicht exportiert
var Pi = 3.14159    // wird exportiert
```

Quelltext 2.2: Bestimmung der Sichtbarkeit über Groß- und Kleinschreibung

2.2 Datentypen

Es gibt in Go *einfache* und *zusammengesetzte Datentypen*. Zu den einfachen Datentypen gehören Wahrheitswerte, numerische Werte und Zeichenketten, zu den zusammengesetzten zählen Structs, Funktionen, Interfaces, Zeiger, Arrays, Slices, Maps und Channels.

All diese Typen lassen sich außerdem in *Wert-* und *Referenztypen* unterteilen. Von ersteren wird bei einer Zuweisung an eine andere Variable oder Weitergabe als Funktionsargument immer eine Kopie erstellt, während bei zweiteren eine Referenz übergeben wird, die auf die ursprünglichen Daten verweist. Wenn eine Funktion also beispielsweise einen Referenztyp als Argument erhält, so ist jede Änderung an diesem auch für den Aufrufer sichtbar. Bei einem Werttyp dagegen wäre dies nicht der Fall, da hier auf einem Duplikat gearbeitet wird. Um dennoch Referenzen auf Werttypen übergeben zu können, gibt es Zeiger (s. Abschnitt 2.7). Im Folgenden werden nun die wichtigsten Datentypen im Detail vorgestellt.

Wahrheitswerte sind vom Typ **`bool`** und können die Werte `true` oder `false` annehmen.

Die numerischen Werte lassen sich in die natürlichen, ganzen, reellen und komplexen Zahlen einteilen. Für alle Zahlenräume gibt es Datentypen, die gleichzeitig die Größe in Bit und damit den Wertebereich bzw. die Genauigkeit spezifizieren (z.B. **`uint16`**, **`int32`**, **`float64`** oder **`complex128`**). Für die natürlichen und ganzen Zahlen gibt es darüber hinaus noch jeweils einen Typen ohne explizite Spezifikation der Größe (**`int`** und **`uint`**). Diese haben in Go 1 eine Größe von 32 Bit.

Zeichenketten sind vom Typ **`string`** und verhalten sich intern wie ein unveränderliches Array (s. Abschnitt 2.8) von Elementen des Typs **`byte`** (welcher wiederum ein Alias für **`uint8`** ist). Somit kann mit der bei Arrays üblichen Indizierung auf die einzelnen Bytes zugegriffen werden. Da Zeichenketten genau wie Quelltextdateien in UTF-8 kodiert sind, bedeutet dies allerdings, dass ein Byte nicht zwangsweise für genau ein Zeichen stehen muss (z.B. benötigt ein ä in der Kodierung von UTF-8 zwei Bytes). Genau aus demselben Grund liefert die integrierte Funktion `len` auch nicht die Anzahl der Zeichen einer Zeichenkette, sondern die Anzahl der für die Repräsentation benötigten Bytes. Um dennoch über die einzelnen Zeichen iterieren zu können, kann die **`range`**-Klausel (s. Abschnitt 2.4.2) verwendet werden.

Von den zusammengesetzten Datentypen werden an dieser Stelle nur die Structs (manchmal auch Records genannt) behandelt. Die restlichen Typen werden aufgrund ihrer speziellen Natur jeweils in einem eigenen Abschnitt erläutert.

Structs sind eine zusammenhängende Menge von eindeutig bezeichneten und typisierten Feldern. Sie müssen als neuer Typ deklariert werden, was über das Schlüsselwort **`type`** geschieht. Nach dem daraufhin folgenden Typnamen wird mit dem Schlüsselwort **`struct`** angegeben, dass ein Struct definiert wird. Die einzelnen Felder folgen in einem von geschweiften Klammern umschlossenen Block und werden zeilenweise untereinan-

der notiert, wobei Felder desselben Typs durch Kommata separiert auch in einer Zeile zusammengefasst werden können:

```
type DayMonth struct {
    day, month uint
}
```

Quelltext 2.3: Definition eines Structs

Structs können andere Structs auch als anonyme Felder, d.h. Felder ohne einen Bezeichner, einbinden. Die Felder des inneren Structs können dann wie Felder des äußeren Typs genutzt werden, sofern keine Namenskonflikte unter den Bezeichern auftreten. Das folgende Beispiel verdeutlicht dieses Prinzip:

```
type Date struct {
    DayMonth
    year uint
}
```

Quelltext 2.4: Einbinden eines Structs als anonymes Feld

Würde nun durch `var birthday Date` eine Variable von dem in Quelltext 2.4 definierten Typ deklariert werden (s. Abschnitt 2.3), so kann sowohl auf das neu definierte Feld `year` als auch auf die Felder `day` und `month` des inneren Structs zugegriffen werden.

2.3 Variablen und Konstanten

Die Deklaration von Variablen wird durch das Schlüsselwort **var** eingeleitet. Anschließend folgt der Name der Variablen sowie optional ein Typ und Initialwert. Wird kein Initialwert angegeben, so ist die Angabe des Typs zwingend erforderlich. Wird dagegen der Typ weggelassen, so ergibt sich der Typ der Variablen durch den Typ des zugewiesenen Wertes. So wird durch `var i = 42` beispielsweise eine Variable `i` vom Typ **int** deklariert, die mit dem Wert `42` initialisiert wird. Ähnlich wie schon bei dem Import von Packages (s. Quelltext 2.1) können mehrere Deklarationen zusammengefasst werden:

```
var (
    a, b, c float64
    x, y = "Hallo", true
)
```

Quelltext 2.5: Zusammenfassen mehrerer Deklarationen sowie Initialisieren mit verschiedenen Typen

Eine Besonderheit hierbei ist die Möglichkeit der gleichzeitigen Initialisierung mehrerer Variablen, sogar mit unterschiedlichen Typen. Im Fall von Beispiel 2.5 ergeben sich die Typen von `x` und `y` durch die Zuweisung der Werte `"Hallo"` und `true` implizit als **string** bzw. **bool**. Dieses Prinzip der Mehrfachzuweisung findet sich auch bei Funktionen wieder (s. Abschnitt 2.5).

Innerhalb von Funktionen kann die Deklaration von Variablen mit gleichzeitiger Initialisierung auch in einer verkürzten Form notiert werden. Auf das Schlüsselwort **var** sowie die Angabe eines Typs wird dabei verzichtet. Dafür findet für die Typbestimmung notwendigerweise eine initiale Zuweisung statt, wobei dem Gleichheitszeichen hierbei ein Doppelpunkt vorangestellt werden muss. Die beiden folgenden Anweisungen sind also

äquivalent:

```
var x, y = "Hallo", true
x, y := "Hallo", true
```

Quelltext 2.6: Verkürzte Deklaration von Variablen mit gleichzeitiger Initialisierung

Konstanten werden über das Schlüsselwort **const** definiert, gefolgt vom Namen der Konstante, optional einem Datentyp sowie einem konstanten Ausdruck hinter einem Gleichheitszeichen. Der Ausdruck `const Pi = 3.14159` definiert also eine Konstante namens `Pi` mit dem Wert `3.14159`. Wie bereits in Quelltext 2.1 oder 2.5 kann die Definition mehrerer Konstanten zusammengefasst werden:

```
const (
    Pi, E = 3.14159, 2.71828
    Language = "Go"
)
```

Quelltext 2.7: Zusammenfassen mehrerer Konstantendefinitionen

2.4 Kontrollstrukturen

2.4.1 Verzweigungen

Go bietet die üblichen Konstrukte, um Verzweigungen innerhalb des Programmflusses zu ermöglichen. So steht das allgemein bekannte **if**-Konstrukt mit optionalem **else**-Zweig zur Verfügung, wobei dieses in Go optional auch eine Initialisierungsanweisung enthalten kann, die vor dem Prüfen der Bedingung ausgeführt wird:

```
if Initialisierung; Bedingung {
    // Bedingung ist wahr gewesen
} else {
    // Bedingung ist falsch gewesen
}
```

Quelltext 2.8: Allgemeine Form der If-Else-Verzweigung

Mehrfachverzweigungen können mittels **switch** durchgeführt werden. Im Unterschied zu C ist der **switch**-Ausdruck in Go nicht auf Ganzzahlen beschränkt. Die allgemeine Form der Mehrfachverzweigung lautet:

```
switch Initialisierung; Ausdruck {
case Wert1:
    // Ausdruck == Wert1
case Wert2:
    // Ausdruck == Wert2
default:
    // Kein Fall ist eintreten
}
```

Quelltext 2.9: Allgemeine Form der Mehrfachverzweigung

Auch hier besteht wieder die Möglichkeit, optional eine Initialisierungsanweisung anzugeben. Trifft keiner der angegebenen Fälle zu, so wird der **default**-Zweig ausgeführt,

sofern vorhanden. Im Standardfall wird in Go immer nur ein Zweig, nämlich der erste zutreffende, ausgeführt. Über das Schlüsselwort **fallthrough** am Ende eines Zweiges kann jedoch erzwungen werden, dass auch die Anweisungen des nachfolgenden Zweiges abgearbeitet werden, unabhängig davon, ob die zugehörige Bedingung erfüllt ist.

Als Letztes bleibt die **goto**-Anweisung für Sprünge zu erwähnen, welche es auch in Go noch gibt. Folgendes Beispiel, welches einer Endlosschleife mit leerem Rumpf entspricht, demonstriert die Verwendung:

```
Label:
goto Label
```

Quelltext 2.10: Beispiel für die Verwendung von **goto**

2.4.2 Schleifen

Go kennt nur ein Schleifenkonstrukt: Die For-Schleife. Sie verhält sich grundsätzlich ähnlich zu der von C, es gibt aber einige Unterschiede in Hinblick auf die verwendete Syntax und die Einsatzzwecke. Die geschweiften Klammern, die den Schleifenrumpf umschließen, sind zwingend erforderlich, während im Gegensatz dazu die runden Klammern, die in C den Schleifenkopf umschließen, in Go nicht erlaubt sind. Die Basisform der For-Schleife ist die folgende:

```
for Initialisierung; Bedingung; Fortsetzung {
    // Rumpf
}
```

Quelltext 2.11: Basisform der For-Schleife

Die Initialisierungs- und die Fortsetzungsanweisung kann optional weggelassen werden. Man erhält dann das Äquivalent zu einer While-Schleife. Wird zusätzlich auch noch die Bedingung weggelassen, so entspricht dies einer Endlosschleife.

Ein besonderer Einsatzzweck ist die Bereichsschleife zusammen mit dem Schlüsselwort **range**. Auf diese Art und Weise kann durch die Elemente von Strings, Arrays, Slices oder Maps iteriert werden. Auch für Channels besteht diese Möglichkeit, allerdings verhält sich die **range**-Klausel in diesem Fall anders (s. Abschnitt 2.10). Die Funktionsweise der Bereichsschleife soll am Beispiel eines Arrays verdeutlicht werden:

```
for i, elem := range array {
    fmt.Println(i, elem)
}
```

Quelltext 2.12: Beispiel einer Bereichsschleife

Im obigen Beispiel 2.12 wird davon ausgangen, dass eine Arrayvariable mit Namen `array` existiert. In jedem Schleifendurchgang wird nun in `i` und `elem` der aktuelle Index und eine Kopie des Elements mit diesem Index gespeichert, die dann über `fmt.Println` auf der Konsole ausgegeben wird. Die Schleife wird automatisch verlassen, sobald alle Elemente durchlaufen sind.

2.5 Funktionen und Methoden

Funktionen werden durch das Schlüsselwort **func** eingeleitet. Ihre allgemeine Form lautet wie folgt, wobei die Angabe von Argumenten und Rückgabewerten optional ist:

```
func Name(Argumente) Rückgabe {
    // Rumpf
}
```

Quelltext 2.13: Allgemeine Form von Funktionen

Ähnlich wie in Python kann eine Funktion in Go mehrere Rückgabewerte haben. Im Beispiel 2.14 gibt die Funktion `successors` die beiden nächsten Nachfolger der als Argument übergebenen Zahl zurück.

```
func successors(n uint) (uint, uint) {
    return n + 1, n + 2
}
```

Quelltext 2.14: Beispiel einer Funktion mit mehreren Rückgabewerten

Mitunter kann es vorkommen, dass eine Funktion zwar mehrere Rückgabewerte liefert, aber nicht alle für die weitere Verarbeitung relevant sind. In diesem Fall können zu ignorierende Rückgabewerte dem sogenannten *Blank Identifier* zugewiesen werden. Dieser wird mit einem Unterstrich (_) notiert:

```
_, i := successors(2)
```

Quelltext 2.15: Verwendung des Black Identifiers

Funktionen können auch auf eigenen Datentypen definiert werden, es handelt sich dann um Methoden des jeweiligen Typs. Der Typ, auf dem die Funktion definiert ist, wird zusammen mit einem Bezeichner in Klammern vor den Namen der Funktion geschrieben. Über den Bezeichner erhält man innerhalb der Funktion Zugriff auf die Variable, auf der die Methode aufgerufen wurde.

```
type Complex struct {
    re, im float64
}

func (c Complex) Magnitude() float64 {
    return math.Sqrt(c.re * c.re + c.im * c.im)
}
```

Quelltext 2.16: Definition von Methoden auf eigenen Datentypen

Im Beispiel 2.16 wird zunächst ein eigener Datentyp namens `Complex` definiert, der eine komplexe Zahl repräsentieren soll. Es folgt die Funktion `Magnitude`, um den Betrag einer komplexen Zahl zu berechnen. `Magnitude` ist dabei eine Methode des Datentyps `Complex`. Wenn `number` nun eine Variable vom Typ `Complex` wäre, erhielte man mithilfe der Anweisung `number.Magnitude()` den Betrag von `number`.

Wie eingangs erwähnt, unterstützt Go auch anonyme Funktionen und Funktionen höherer Ordnung. Am Beispiel einer Map-Funktion für Ganzzahl-Slices (s. Abschnitt 2.8), die eine übergebene Funktion auf jedes Element des übergebenen Slices anwendet, soll dies demonstiert werden:

```
func mapInts(s []int, f func(int) int) {
    for i, elem := range s {
        s[i] = f(elem)
    }
}
```

Quelltext 2.17: Map-Funktion für Ganzzahl-Slices unter Verwendung von Funktionen höherer Ordnung

Die Funktion `mapInts` erwartet einen Slice `s` und eine Funktion `f`. Die Funktion `f` muss dabei nach der Typspezifikation ein Argument vom Typ **`int`** erwarten und auch wieder eine Ganzzahl zurückgeben. Ein Aufruf der Funktion `mapInts` auf einem Slice `slice` und unter Verwendung einer anonymen Funktion könnte dann wie folgt aussehen:

```
mapInts(slice, func(i int) int { return i + 1 })
```

Quelltext 2.18: Verwendung anonymer Funktionen

2.6 Interfaces

Durch Interfaces halten Aspekte der objektorientierten Programmierung in Go Einzug. Das folgende Beispiel demonstriert die Verwendung von Interfaces in Go. Zunächst werden zwei Structs `Square` und `Circle` für ein Quadrat bzw. einen Kreis definiert.

```
type Square struct {
    length float64
}

type Circle struct {
    radius float64
}
```

Quelltext 2.19: Typdefinition für ein Quadrat und einen Kreis

Anschließend definieren wir ein Interface namens `Form`, das geometrische Figuren repräsentieren soll und die Methode `Area` beinhaltet.

```
type Form interface {
    Area() float64
}
```

Quelltext 2.20: Interfacedefinition für geometrische Figur

Wie in Abschnitt 2.5 erwähnt, können auf den eigenen Typen Funktionen definiert werden. In diesem Fall definieren wir auf den beiden Typen `Square` und `Circle` die Funktion `Area`.

Mit all diesen Definition ist bereits erreicht, dass die Datentypen `Square` und `Circle` das Interface `Form` implementieren. Anders als beispielsweise in Java bedarf es dazu keiner Angabe, dass ein Typ ein bestimmtes Interface implementiert. Dies ist automatisch der Fall, sobald auf einem Typ alle Methoden eines Interfaces definiert sind. Dieses Konzept ist allgemein unter der Bezeichnung *Duck Typing* bekannt.

```
func (s Square) Area() float64 {
    return s.length * s.length
}

func (c Circle) Area() float64 {
    return math.Pi * c.radius * c.radius
}
```

Quelltext 2.21: Implementierung der Flächenberechnung für Quadrate und Kreise

Eine besondere Rolle bei den Interfaces spielt das *leere Interface* `interface{}`. Es beinhaltet keine Methoden und wird somit von jedem Typen implementiert. Um den tätsächlichen Typ einer Variable vom Typ `interface{}` in Erfahrung zu bringen, lässt sich neben der Reflexion (s. Abschnitt 2.11) auch ein *Type Switch* wie in Quelltext 2.22 verwenden.

```
func testType(arg interface{}) {
    switch a := arg.(type) {
    case int:
        // arg war vom Typ int
    default:
        // arg hatte einen anderen Typ
    }
}
```

Quelltext 2.22: Type Switching bei leerem Interface

Innerhalb eines zutreffenden **case**-Zweiges hat die Variable a dann den ermittelten Typ. Im obigen Beispiel 2.22 bedeutet dies, dass die Variable a vom Typ **int** wäre und direkt als solche verwendet werden könnte, wenn der erste **case**-Zweig der zutreffende Zweig und somit **int** der ermittelte Typ von `arg` gewesen wäre. Die Nutzung von **fallthrough** ist bei einem Type Switch nicht gestattet, da der Typ von a dann nicht mehr eindeutig bestimmt wäre.

2.7 Zeiger

Um einen Zeigertypen zu definieren, wird dem eigentlichen Typ ein Stern (*) vorangestellt, zum Beispiel ***int** für einen Zeiger auf einen Wert vom Typ **int**. Um dagegen die Adresse von einer bereits vorhanden Variable zu erhalten, wird das Kaufmanns-Und (&) vor die entsprechende Variable geschrieben.

Zeiger in Go sind also nahezu identisch zu denen aus C. Allerdings ist der Nullwert eines Zeigers in Go **nil**. Ein weiterer gravierender Unterschied ist das Fehlen der aus C bekannten Zeigerarithmetik, d.h. das Rechnen mit Zeigern ist in Go nicht möglich. Dies ist eine direkte Konsequenz des gesetzten Ziels der Speichersicherheit (s. Abschnitt 1.2).

Um Speicher zu allozieren und die Adresse des reservierten Speicherbereichs zu erhalten, gibt es die eingebaute Funktion new, die als Argument einen Typ erwartet und einen Zeiger auf eben diesen Typ zurückgibt. Der allozierte Speicherbereich wird dabei

grundsätzlich mit Nullen initialisiert.

Mit `new` allozierter Speicher muss nicht wieder freigegeben werden, da dies aufgrund der vorhandenen Garbage Collection automatisch erfolgt.

2.8 Arrays und Slices

Arrays werden durch eine Größe und den Typ der darin enthaltenen Elemente spezifiziert. Beispielsweise repräsentiert `[4]int` einen Array von vier ganzen Zahlen. Die Größe eines Arrays ist dabei unveränderlich und Teil des Datentyps. Das hat zur Konsequenz, dass `[4]int` und `[5]int` zwei verschiedene, inkompatible Typen sind.

In Go zählen Arrays zu den Werttypen, seine Elemente werden also bei einer Zuweisung oder Übergabe als Funktionsargument stets kopiert. Damit unterscheiden sie sich von denen in C, wo sie einen Zeiger auf ihr erstes Element darstellen.

Der folgende Quelltextausschnitt zeigt, wie ein Array bei der Deklaration gleichzeitig mit Werten initialisiert werden kann. Die drei Punkte innerhalb der eckigen Klammern signalisieren dabei, dass der Compiler das Zählen der Elemente übernehmen soll:

```
array := [...]int{1,2,3,4}
```

Quelltext 2.23: Deklaration eines Arrays und Initialisierung mit den Zahlen 1-4

Die Elemente eines Arrays werden wie üblich indiziert: `array[i]` greift somit auf das Element an der Stelle `i` zu, wobei die Zählung bei null beginnt. Über die spracheigene Funktion `len` kann die Größe eines Arrays in Erfahrung gebracht werden. Im obigen Beispiel 2.23 würde `len(array)` also `4` zurückgeben. Außerdem ist, wie bereits in Abschnitt 2.4.2 erwähnt, die **`range`**-Klausel für Arrays verwendbar, um über die enthaltenen Elemente zu iterieren.

Slices bezeichnen ein Segment eines Arrays und besitzen zusätzlich zu ihrer Größe eine Kapazität. Letztere beschreibt die obere Schranke für die Größe des Slices innerhalb des darunterliegenden Arrays. Die Typspezifikation eines Slices lautet `[]T`, wobei `T` für den Typ der enthaltenen Elemente steht. Im Unterschied zu Arrays sind Slices Referenztypen.

Neben der wie bei Arrays vorhandenen Funktion `len` gibt es für Slices noch die Funktion `cap`, um die Kapazität auszulesen. Auch bei Slices können analog zu Arrays mit **`range`** alle Elemente durchlaufen werden. Eine Möglichkeit, Slices zu erstellen, ist analog zu Quelltext 2.23 zu Arrays die folgende:

```
slice := []int{1,2,3,4}
```

Quelltext 2.24: Deklaration eines Slices und Initialisierung mit den Zahlen 1-4

Im obigen Beispiel 2.24 hätte der Slice eine Länge und Kapazität von vier. Alternativ können Slices auch über die Anweisung `make` angelegt werden. Hierbei kann optional auch eine bestimmte Kapazität angegeben werden:

```
slice := make([]int, 4, 8)
```

Quelltext 2.25: Anlegen eines Slices mithilfe von `make`

Ein besonderes Vorgehen im Zusammenhang mit Slices ist das sogenannte *Slicing*. Es bezeichnet das „Herausschneiden“ eines Bereichs von einem Slices oder Arrays. Dabei entsteht ein neuer Slice mit veränderter Größe und möglicherweise veränderter Kapazität, dem nach wie vor das ursprüngliche Array zugrundeliegt. Die Kapazität des neuen Slices

ergibt sich, indem von der Gesamtlänge des Arrays der Index des ersten Elements des neuen Slices im Array abgezogen wird. Durchgeführt wird das Slicing, indem hinter einem existierenden Slice oder Array in eckigen Klammern zwei Indizes geschrieben werden, getrennt durch einen Doppelpunkt. So erstellt die Anweisung

```
newSlice := slice[i:j]
```

Quelltext 2.26: Erstellen eines neuen Slices durch Slicen

einen neuen Slice, der alle Elemente von Index `i` bis Index `j - 1` des Ausgangsslices `slice` enthält. Die Elemente des neuen Slices werden wieder beginnend bei null indiziert. Auf die Angabe der Indizes `i` und `j` kann optional verzichtet werden, es werden dann standardmäßig die Werte `0` respektive `len(slice)` gewählt. Der Ausdruck `slice[:]` würde also den ursprünglichen Slice zurückgeben.

Zusätzlich zu `len` und `cap` sind für Slices auch noch die Funktionen `copy` und `append` definiert. Erstere kopiert die Inhalte eines Slice in einen anderen, zweitere ermöglicht es, einem bestehenden Slice mehrere Elemente oder ein Slice hinzuzufügen, wobei generell ein neuer Slice zuückgegeben wird, da sich das darunterliegende Array unter Umständen geändert haben kann.

2.9 Maps

Maps gehören in Go wie in Python zu den fest eingebauten Datentypen. Sie dienen der Abbildung von Schlüsseln eines bestimmten Typs auf Elemente eines eventuell davon verschiedenen Typs. Der Schlüsseltyp kann dabei beliebig sein, solange die Operatoren == und != auf diesem definiert sind. Die Einträge werden ungeordnet abgespeichert, weswegen der Zugriff auf ein Element in linearer Zeit, das Einfügen eines Eintrags dagegen in konstanter Zeit erfolgt. Der Typ einer Map ist **`map`**`[T1] T2`, wobei `T1` dem Schlüsseltyp und `T2` dem Wertetyp entspricht. Maps sind genau wie Slices Referenztypen.

Erstellt werden Maps mithilfe von `make`. Durch die folgende Anweisung wird also eine Map mit Namen `numbers` erstellt, die Schlüssel vom Typ **`string`** und Werte vom Typ **`int`** besitzt:

```
numbers := make(map[string] int)
```

Quelltext 2.27: Erstellen einer Map

Die Verwendung gestaltet sich analog zu der von Arrays oder Slices, nur dass auf die Elemente nicht über einen Index zugegriffen wird, sondern über ihren Schlüssel. Der Ausdruck

```
numbers["eins"]
```

Quelltext 2.28: Zugriff ein einen Eintrag einer Map

würde also auf den Eintrag mit dem Schlüssel `"eins"` zugreifen. Gibt es zu diesem Schlüssel keinen Eintrag, so liefert der Ausdruck den Initialwert des Wertetypen zurück. In diesem Beispiel wäre das `0`, da der Wertetyp **`int`** ist. Um einen neuen Eintrag hinzuzufügen oder einen bereits vorhandenen zu überschreiben, weißt man dem Ausdruck aus Quelltext 2.28 einen Wert des Wertetyps zu. Das Löschen eines Eintrags wird mithilfe der eingebauten Funktion `delete` durchgeführt, die als erstes Argument eine Map und als zweites den Schlüssel des Eintrags erwartet. Genau wie bei Arrays und Slices kann

erneut das Schlüsselwort **`range`** verwendet werden, um über alle Einträge zu iterieren.

2.10 Goroutinen und Channels

Ein zentrales Sprachelement von Go sind die Goroutinen, nebenläufig ausgeführte Funktionen, d.h. der eigentliche Programmfluss läuft ungehindert weiter. Goroutinen werden mit dem Schlüsselwort **`go`** gefolgt von einem Funktionsaufruf gestartet. Dabei kann jede beliebige Funktion als Goroutine ausgeführt werden.

Goroutinen sind leichtgewichtiger als Threads, kosten also weniger Ressourcen als diese. Sie werden auf einem Thread Pool verteilt, sodass ohne Probleme mehrere Tausend Goroutinen gleichzeitig ausgeführt werden können. Blockiert eine Goroutine einen Thread, so können andere Goroutinen auf die übrigen Threads verteilt werden. Dies wird vom Laufzeitsystem automatisch gehandhabt. Mit dem Beenden des eigentlichen Programms werden auch sämtliche laufende Goroutinen beendet.

Ist die Ausführung einer Goroutine beendet, so erfolgt ohne Weiteres keine Rückmeldung darüber. Auch eventuelle Rückgabewerte der als Goroutine gestarteten Funktion werden verworfen. Um dies zu umgehen, können Channels verwendet werden.

Channels sind ebenfalls ein zentrales Konzept in Go und dienen der Kommunikation mehrerer Goroutinen und des Hauptprogramms untereinander. Häufig findet die Kommunikation bei nebenläufiger Programmierung über gemeinsam genutzten Speicher statt, wobei der Zugriff auf diesen beispielsweise über Semaphoren reguliert werden kann. Diese Möglichkeit besteht nach wie vor in Go (über das Package `sync`), der empfohlene Weg ist allerdings die Verwendung von Channels.

Erstellt werden sie mit der Funktion `make`, die als Argument die Typdefinition des Channels erwartet. Diese lautet **`chan`** `T`, wobei `T` dabei ein beliebiger Datentyp sein kann, der über den Channel ausgetauscht werden soll. Als optionales zweites Argument kann bei `make` eine Kapazität für den Channel angegeben werden. Diese bestimmt, wieviele Elemente vom Typ `T` zwischengespeichert werden können, bevor alle weiteren Schreibzugriffe auf den Channel blockieren. Wird auf die Angabe der Kapazität verzichtet, so wird diese implizit auf 1 gesetzt.

Bei Channels, die nur zur Ausgabe bestimmt sind, kann zusätzlich `<-` hinter das Schlüsselwort **`chan`** geschrieben werden. Ebenso kann man mit `<-`**`chan`** einen Channel definieren, aus dem nur gelesen werden kann.

Das Lesen und Schreiben aus bzw. in einen Channel erfolgt über den Operator `<-`. Die Verwendung ist in Quelltext 2.30 und 2.31 zu sehen. Wenn ein Channel vom Typ **`chan`** `T` ist, können nur Daten von diesem Typ `T` geschrieben oder gelesen werden. Es können beliebig viele Goroutinen lesend und schreibend auf einen Channel zugreifen, sofern dieser die jeweilige Operation zulässt. Beiden Operationen werden atomar ausgeführt.

Bei Channels kann außerdem die **`range`**-Klausel in einer For-Schleife verwendet werden (s. Quelltext 2.31). Die Schleife wird dann solange ausgeführt, wie der Channel nicht geschlossen ist. Sollte der Channel in einem Durchlauf leer sein, wird die Ausführung der Schleife blockiert und erst dann wieder fortgesetzt, wenn wieder etwas im Channel zum Auslesen vorhanden ist.

Für den Fall, dass bei der Möglichkeit von Eingaben aus mehreren Channels verzweigt werden soll, gibt es in Go das **select**-Konstrukt. Es ähnelt stark der bereits in Abschnitt 2.4.1 vorgestellten **switch**-Anweisung.

```
select {
    case fromOne <- channelOne:
        // Signal aus channelOne in fromOne geschrieben
    case <- channelTwo:
        // Signal aus channelTwo empfangen
    default:
        // Es gab kein Signal
}
```

Quelltext 2.29: Bei möglichen Eingaben aus mehreren Channels verzweigen

Die einzelnen **case**-Ausdrücke werden immer von oben nach unten ausgewertet. Wird der **default**-Zweig weggelassen, so blockiert die **select**-Anweisung, bis einer der Channels ein Signal empfängt.

Die folgende Variante des Consumer-Producer-Problems soll die Verwendung von Channels im Zusammenhang mit Goroutinen verdeutlichen. Dabei wird ein simples Warenlager modelliert, bei dem Produzenten Waren einlagern und Konsumenten Waren erwerben können. Dabei wird davon ausgegangen, dass es einen Typ `Item` gibt, der eine Ware repräsentiert, sowie die Funktionen `makeItem` und `buyItem`. Erstere generiert eine Ware, liefert also etwas vom Typ `Item` zurück, letztere erwartet als Argument eine Ware vom Typ `Item` und verarbeitet diese auf nicht näher bestimmte Art und Weise.

```
func producer(depot chan<- Item) {
    for {
        depot <- makeItem()
    }
}
```

Quelltext 2.30: Funktion zur Simulation des Produzenten

Quelltext 2.30 zeigt den Producer, also die Funktion, die im gewählten Beispiel Waren produziert und einlagert. Sie erhält als Argument einen Channel `depot`, auf den innerhalb der Funktion nur geschrieben werden kann, was durch **chan**<- signalisiert wird. In einer Endlosschleife werden dann mit der Funktion `makeItem` Elemente vom Typ `Item` generiert und über den Operator <- in den Channel geschrieben. Die Schreiboperation blockiert dabei automatisch, solange der Channel voll ist.

```
func consumer(depot <-chan Item) {
    for good := range depot {
        buyItem(good)
    }
}
```

Quelltext 2.31: Funktion zur Simulation des Konsumenten

Im obigen Quelltext 2.31 ist der Consumer zu sehen. Die Funktion erhält ebenfalls einen Channel `depot` als Argument, der hierbei innerhalb der Funktion nur lesend gebraucht werden kann. Mithilfe der Bereichsschleife werden vorhandene Waren aus dem Channel

in die Variable `good` gelesen und mit der Funktion `buyItem` weiterverarbeitet. Wie oben beschrieben, pausiert die Schleife bei einem leeren Channel, bis wieder Elemente zum Auslesen vorhanden sind.

```
depot := make(chan Item, 10)
```

Quelltext 2.32: Erstellen eines Channels für das Warenlager

Wird nun durch eine Anweisung wie in Quelltext 2.32 ein Warenlager erstellt, können über die Anweisungen **go** `producer(depot)` und **go** `consumer(depot)` beliebig viele Producer und Consumer parallel gestartet werden, die allesamt auf demselben Warenlager arbeiten. Die Laufzeitumgebung von Go gewährleistet dabei den korrekten Zugriff auf dieses. Wie man somit sehen kann, lässt sich das Consumer-Producer-Problem in Go mithilfe von Goroutinen und Channels recht elegant lösen.

2.11 Reflexion

Reflexion bezeichnet die Fähigkeit eines Programms, seine Struktur zur Laufzeit zu untersuchen und eventuell sogar zu verändern. Go ermöglicht einem dies mithilfe des Packages `reflect`.

Da die Möglichkeiten dieses Packages sehr umfangreich sind, wird im Folgenden nur ein kleines Beispiel gegeben, in dem demonstriert wird, wie man ähnlich zum Type Switching (vgl. Quelltext 2.22) mithilfe des `reflect`-Packages den Typ einer Variable ermitteln kann.

```
func getType(arg interface{}) string {
    return reflect.TypeOf(arg).Name()
}
```

Quelltext 2.33: Typermittlung einer Variable

Die Funktion `getType` erhält dabei ein Argument vom Typ **interface**`{}`. Somit ist der tatsächliche Typ der Variable `arg` innerhalb der Funktion nicht bekannt ist. Über die Funktion `TypeOf` des Packages `reflect` erhält man dann zunächst eine Struktur, die vom Typ `Type` ist, der ebenfalls Teil des `reflect`-Packages ist, und zahlreiche Informationen über den Datentyp von `arg` enthält. Über die Methode `Name` des Datentyps `Type` wird schließlich der eigentliche Namen des Datentyps ausgelesen, der von der Funktion `getType` abschließend zurückgegeben wird.

3 Technische Unterstützung

3.1 Compiler

Es gibt mit gc und gccgo derzeit zwei Compiler für Go. gc ist dabei eigentlich eine Sammlung von mehreren Compilern für unterschiedliche Prozessorarchitekturen. Dies sind namentlich 5g für ARM, 6g für AMD64 und 8g für x86. gc ist fester Bestandteil des auf der offiziellen Website angebotenen Softwarepakets, welches für Linux, Windows, Mac OS X und FreeBSD verfügbar ist.

gccgo ist ein Front-End für die GNU Compiler Collection, auch bekannt als gcc. Es unterstützt daher viele Optionen, die auch gcc beherrscht, u.a. die Optimierungsstufe O3. Im Regelfall benötigt das Kompilieren eines Goprogramms mit gccgo minimal mehr Zeit als mit gc, dafür ist der erzeugte Code aber auch minimal effizienter (s. Kapitel 4). gccgo ist im Gegensatz zu gc nur unter Unix-artigen Systemen verfügbar.

3.2 Werkzeuge und Entwicklungsumgebungen

Go liefert von Haus aus mehrere Werkzeuge mit, die die Entwicklung unterstützen können. Im Folgenden werden die drei wichtigsten kurz vorgestellt: gofmt dient der automatischen Formatierung von Quelltextdateien, godoc kann ähnlich wie javadoc aus speziell annotierten Kommentaren eine Dokumentation erstellen und gotest testet automatisiert Packages anhand zuvor definierter Testfälle.

Als Entwicklungsumgebung lässt sich prinzipiell jeder Texteditor verwenden, der mit UTF-8-Kodierung umgehen kann. Darüber hinaus bieten fast alle verbreiteten Editoren mit Syntaxhighlighting auch Unterstützung für die Syntax von Go. Unter Umständen kann es aber komfortabler sein, innerhalb einer IDE zu programmieren. Für diesen Fall gibt es dann beispielsweise mit GoClipse ein passendes Plugin für Eclipse.

3.3 Bibliotheken

Go bringt mit der Standardbibliothek bereits eine Vielzahl von Bibliotheken mit, die für alle offiziell unterstützten Plattformen verfügbar sind. Eine vollständige Übersicht aller zur Standardbibliothek gehörigen Packages findet sich auf der Website http://golang.org/pkg/.

Sollte der Umfang der sehr umfangreichen Standardbibliothek mal nicht ausreichen, gibt es zahlreiche Community-Projekte, die eigene Bibliotheken anbieten, u.a. für die Umsetzung grafischer Benutzeroberflächen. Unter der Adresse http://go-lang.cat-v.org/pure-go-libs ist eine Auswahl verschiedener angebotener Packages zu finden.

4 Effizienz

4.1 Geschwindigkeitsvergleich am Beispiel von Quicksort

Um die Ausführungsgeschwindigkeit von kompilierten Go-Programmen zu evaluieren und mit anderen Programmiersprachen wie C oder Java in Relation zu setzen, wurde auf Basis des Sortieralgorithmus Quicksort ein einfacher Benchmark entwickelt.

Dabei wurde das Sortierverfahren in mehreren Sprachen umgesetzt, wobei bei jeder Sprache darauf geachtet wurde, den Algorithmus möglichst effizient umzusetzen und eventuell nutzbringende Sprachfeatures für die Implementierung zu verwenden.

Für den eigentlichen Benchmark wurde dann ein mit Zufallswerten gefülltes Array zehnmal sortiert. Aus der bei jedem Sortiervorgang durchgeführten Zeitmessung wurde anschließend die durchschnittlich benötigte Zeit errechnet. Dieses Vorgehen wurde insgesamt für Arrays verschiedener Größen angewandt und auf alle Sprachen übertragen.

Das folgende Diagramm 4.1 veranschaulicht die Ergebnisse des auf diese Art und Weise durchgeführten Geschwindigkeitsvergleiches:

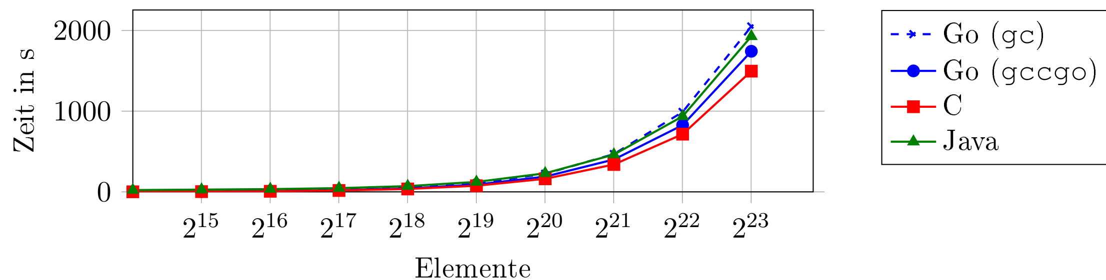


Diagramm 4.1: Quicksort

In allen Programmiersprachen wurde zudem eine Variante implementiert, die das Sortieren nebenläufig ausführt, indem die Ausführung auf mehrere Threads verteilt wird. Das folgende Diagramm 4.2 zeigt die Benchmarkergebnisse für diese Variante:

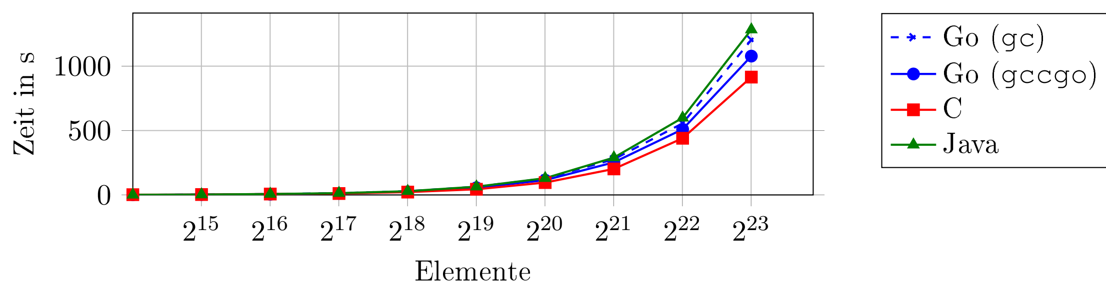


Diagramm 4.2: Parallelisiertes Quicksort

4.2 Diskussion der Ergebnisse

Im Gesamtbild ergibt sich aus den Diagrammen 4.1 und 4.2, dass die Ausführungsgeschwindigkeit von Go-Programmen genauso hoch oder sogar höher als die von Java ist und durchaus an die von C herankommt, insbesondere bei der Verwendung des gccgo-Compilers.

Allerdings sollte erwähnt werden, dass die hier durchgeführte Effizienzbetrachtung selbstverständlich nur bedingt aussagekräftig ist, weil nur ein spezieller Algorithmus untersucht wurde und die Benchmarks auf einer einzigen Rechnerkonfiguration[1] durchgeführt wurden. Abhängig vom Testszenario und der Testumgebung können sich eventuell unterschiedliche Resultate ergeben.

Dennoch lässt sich in diesem konkreten Fall festhalten, dass die hohe Ausführungsgeschwindigkeit in Hinblick auf die Einfachheit und den Umfang des Quelltextes durchaus bemerkenswert ist. Dies ist vor allem bei der Variante, die das Sortieren auf mehrere Threads verteilt, der Fall, da der Quelltext unter Go durch die Verwendung der Go-Routinen deutlich kürzer als in allen anderen Sprachen ausfällt. Zumindest in diesem Szenario bietet Go somit bei erheblich weniger Quelltextumfang nahezu dieselbe Effizienz wie C, die sicherlich das Optimum darstellt.

[1] Prozessor: Intel Core 2 Duo T6400 2x2.0GHz, Betriebssystem: Ubuntu 12.04 LTS, gcc-Version: 4.7.2, gccgo-Version: 4.7.2, Java-Version: 1.6.0.24 (OpenJDK Runtime Environment), Go-Version: 1.0.3

5 Fazit

Für einige mag Go gewissermaßen einen Rückschritt darstellen. So wurde bewusst auf viele Features anderer populärer Sprachen wie C/C++ oder Java verzichtet, zum Beispiel Exceptions, Assertions, Generics oder die Überladung von Methoden und Operatoren. Dabei sei allerdings gesagt, dass es nicht ausgeschlossen ist, dass einige dieser Funktionen noch in späteren Versionen umgesetzt werden. Der Verzicht auf implizite Type Conversions, wie es sie in C gibt (hier heißt dieses Vorgehen Type Casting), ist auch ein viel diskutierter Kritikpunkt. Diese Entscheidung wurde aber bewusst getroffen, um der Entstehung von Programmierfehlern entgegen zu wirken.

Für andere wiederum ist Go ein gelungener Versuch der Umsetzung einer modernen Sprache zur Systemprogrammierung. Die Unterstützung nebenläufiger Programmierung durch native Sprachelemente ist mit Sicherheit eine der größten Stärken von Go. Die oben genannten Kürzungen am Sprachumfang im Vergleich zu anderen Sprachen können auch als Vereinfachungen gesehen werden, die einem das Programmieren erleichtern.

In Hinblick auf die ursprünglich gesetzten Ziele (s. Abschnitt 1.2) kann man sagen, dass Go diese durchaus erreicht hat. Auch lässt Go mit impliziten Typzuweisungen bei der Variablendeklaration und den Interfaces, die allein durch das Definieren von Methoden implementiert werden, in der Tat ein wenig das Gefühl aufkommen, dass man in einer dynamischen Sprache programmiert. Dennoch muss man nicht auf die Vorteile einer kompilierten Programmiersprache verzichten.

Da Go sich stark an der weit verbreiteten Sprache C orientiert, gleichzeitig aber eine deutliche Vereinfachung gegenüber dieser darstellt, und es eine umfangreiche Dokumentation gibt, ist der Einstieg recht einfach und viel auch mir nicht schwer.

Ich persönlich fand es interessant und sehr lehrreich, eine noch so junge Sprache wie Go kennenzulernen. Man darf gespannt sein, was die Zukunft bringen wird, wo doch bereits jetzt eine aktive Community um diese Sprache besteht. Ein Großteil seiner Bekanntheit hat Go wohl seinem prominenten Schirmherren Google zu verdanken, aber wie am Beispiel von Java zu sehen ist, ist dies nicht unbedingt von Nachteil. In jedem Fall werde ich die weitere Entwicklung mit Interesse verfolgen.

Quellenverzeichnis

[1] Müller, F.: *Systemprogrammierung in Google Go*, dpunkt.verlag, 2011

[2] Feike R., Blass S.: *Programmierung in Google Go: Einstieg, Beispiele und professionelle Anwendung*, Addison-Wesley, 2010

[3] Chisnall, D.: *The Go Programming Language Phrasebook*, Addison-Wesley, 2012

[4] *FAQ - The Go Programming Language*,
http://golang.org/doc/faq, 30.01.2013

[5] *Effective Go - The Go Programming Language*,
http://golang.org/doc/effective_go.html, 30.01.2013

[6] *The Go Programming Language Specification - The Go Programming Language*,
http://golang.org/ref/spec, 04.02.2013

[7] *go-wiki - Go Language Community Wiki*,
http://code.google.com/p/go-wiki/, 09.02.2013

[8] *Google programming Frankenstein is a Go*,
http://www.theregister.co.uk/2010/05/20/go_in_production_at_google/,
13.02.2013

[9] *App Engine Go Overview - Google App Engine - Google Developers*,
https://developers.google.com/appengine/docs/go/overview, 19.02.2013

[10] *goclipse - Eclipse-based IDE for Google's Go Programming Language*,
http://code.google.com/p/goclipse/, 20.02.2013

Anhang

benchmark.sh

```
#!/bin/bash

MAX_SIZE=8388608
MAX_RUNS=10
COUNT=4
DEPTH=4
FOLDER="./"
SUFFIX="_threads"
ENDING=".csv"

DESC[0]="Go␣(gc)"
DESC[1]="Go␣(gccgo)"
DESC[2]="C"
DESC[3]="Java"
DESC[4]="Python"

BUILD[0]="go␣build␣quicksort.go"
BUILD[1]="gccgo␣-O3␣quicksort.go␣-o␣quicksort"
BUILD[2]="gcc␣-std=c99␣-pthread␣-O3␣quicksort.c␣-o␣quicksort"
BUILD[3]="javac␣QuickSort.java"
BUILD[4]=""

RUN[0]="./quicksort"
RUN[1]="./quicksort"
RUN[2]="./quicksort"
RUN[3]="java␣-Xss16m␣QuickSort"
RUN[4]="./quicksort.py"

CLEAN[0]="rm␣-f␣quicksort"
CLEAN[1]="rm␣-f␣quicksort"
CLEAN[2]="rm␣-f␣quicksort"
CLEAN[3]="rm␣-f␣QuickSort.class␣QuickSort\$Wrapper.class"
CLEAN[4]=""
```

```
OUTPUT[0]="gc"
OUTPUT[1]="gccgo"
OUTPUT[2]="c"
OUTPUT[3]="java"
OUTPUT[4]="python"

if [[ $# -gt 0 ]]; then
    if [[ $1 != *[!0-9]* ]]; then
        if [[ $1 -gt 0 ]]; then
            MAX_SIZE=$1
        fi
    fi
fi

for ((i=0; i<$COUNT; i++)); do
    echo "Testing ${DESC[i]}..."
    mkdir -p ${FOLDER}
    echo -n "" > ${FOLDER}${OUTPUT[i]}${ENDING}
    echo -n "" > ${FOLDER}${OUTPUT[i]}${SUFFIX}${ENDING}

    echo "Building executable..."
    ${BUILD[i]}

    echo "Running benchmark..."
    for ((size=1; size<=$MAX_SIZE; size=size*2)); do
        sum[0]=0
        sum[1]=0

        echo -n "Sorting $size element"
        if [[ $size -gt 1 ]]; then
            echo -n "s"
        fi
        for ((run=1; run<=$MAX_RUNS; run++)); do
            result=$(${RUN[i]} $size $DEPTH | sed 's/failed/0/g
                ' | grep -P -o "[0-9]+")
            sum[0]=$((${sum[0]}+$(($(echo "$result" | head -1))
                )))
            sum[1]=$((${sum[1]}+$(($(echo "$result" | tail -1))
                )))
            echo -n "."
        done
        echo
```

```
        average[0]=$((${sum[0]}/$MAX_RUNS))
        average[1]=$((${sum[1]}/$MAX_RUNS))

        echo "${size};${average[0]}" >> ${FOLDER}${OUTPUT[i]}${
           ENDING}
        echo "${size};${average[1]}" >> ${FOLDER}${OUTPUT[i]}${
           SUFFIX}${ENDING}
    done

    ${CLEAN[i]}
done
```

quicksort.go

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "runtime"
    "strconv"
    "time"
)

func createArray(size int) []int {
    array := make([]int, size)

    rand.Seed(time.Now().UnixNano())

    for i, _ := range array {
        array[i] = rand.Int()
    }

    return array
}

func partition(data []int) int {
    i, j, x := 1, len(data) - 1, data[0]

    for i <= j {
        for i <= j && data[i] <= x {
```

```
            i++
        }

        for j >= i && data[j] >= x {
            j--
        }

        if i < j {
            data[i], data[j] = data[j], data[i]
        }
    }

    data[0], data[j] = data[j], data[0]

    return j
}

func quickSort(data []int) {
    if len(data) > 1 {
        index := partition(data)

        quickSort(data[:index])
        quickSort(data[(index + 1):])
    }
}

func wrapper(data []int, depth int, done chan<- bool) {
    if depth < 1 {
        quickSort(data)
        done <- true
    } else {
        if len(data) > 1 {
            index := partition(data)

            wait := make(chan bool)

            go wrapper(data[:index], depth - 1, wait)
            go wrapper(data[(index + 1):], depth - 1, wait)

            done <- (<- wait && <- wait)
        } else {
            done <- true
        }
```

```
    }
}

func isSorted(data []int) bool {
    for i := 0; i < len(data) - 1; i++ {
        if data[i] > data[i + 1] {
            return false
        }
    }

    return true
}

func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())

    if len(os.Args) < 2 {
        fmt.Println("Specify␣array␣size␣as␣parameter!")
        os.Exit(1)
    }

    size, _ := strconv.Atoi(os.Args[1])

    if size <= 0 {
        fmt.Println("Parameter␣must␣be␣a␣positive␣integer!")
        os.Exit(1)
    }

    var depth int

    if len(os.Args) > 2 {
        depth, _ = strconv.Atoi(os.Args[2])

        if depth < 0 {
            fmt.Println("Depth␣must␣be␣zero␣or␣a␣positive␣
               integer!")
            os.Exit(1)
        }
    } else {
        depth = 4
    }

    var t0, t1 time.Time
```

```
    array := createArray(size)

    t0 = time.Now()
    quickSort(array)
    t1 = time.Now()

    if isSorted(array) {
        fmt.Println("Sorted␣in", t1.Sub(t0).Nanoseconds() /
           1000000, "milliseconds.")
    } else {
        fmt.Println("Sorting␣failed!")
    }

    array = createArray(size)

    t0 = time.Now()
    done := make(chan bool)
    go wrapper(array, depth, done)
    <- done
    t1 = time.Now()

    if isSorted(array) {
        fmt.Println("Sorted␣with␣go-routines␣in", t1.Sub(t0).
           Nanoseconds() / 1000000, "milliseconds.")
    } else {
        fmt.Println("Sorting␣with␣go-routines␣failed!")
    }
}
```

quicksort.c

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/time.h>

struct qdata {
    int *data;
    int size;
    int depth;
```

```
};

int is_sorted(int *data, int size)
{
    for (int i = 0; i < size - 1; i++) {
        if (data[i] > data[i + 1]) {
            return 0;
        }
    }

    return 1;
}

static void start_timer(struct timeval *start)
{
    gettimeofday(start, NULL);
}

static void end_timer(struct timeval *start, int *data, int
   size, int threads)
{
    struct timeval end;

    gettimeofday(&end, NULL);

    if (is_sorted(data, size)) {
        if (threads) {
            printf("Sorted␣with␣threads␣in␣");
        } else {
            printf("Sorted␣in␣");
        }
        printf("%li", (end.tv_sec - start->tv_sec) * 1000 + (
           end.tv_usec - start->tv_usec) / 1000);
        printf("␣milliseconds.\n");
    } else {
        if (threads) {
            printf("Sorting␣with␣threads␣failed!\n");
        } else {
            printf("Sorting␣failed!\n");
        }
    }
}
```

```
void init(int *data, int size)
{
    struct timeval t;

    gettimeofday(&t, NULL);

    srand((t.tv_sec * 1000) + (t.tv_usec / 1000));

    for (int i = 0; i < size; i++)
        data[i] = rand();
}

inline static void swap(int *a, int *b)
{
    register int tmp = *a;

    *a = *b;
    *b = tmp;
}

inline static int partition(int *data, int size)
{
    register int pivot = data[0];
    register int i = 1;
    register int j = size - 1;

    while (i <= j) {
        while (i <= j && data[i] <= pivot)
            i++;

        while (j >= i && data[j] >= pivot)
            j--;

        if (i < j)
            swap(&data[i], &data[j]);
    }

    swap(&data[0], &data[j]);

    return j;
}

void quick_sort(int *data, int size)
```

```
{
    if (size > 1) {
        int index = partition(data, size);

        quick_sort(data, index);
        quick_sort(&data[index + 1], size - index - 1);
    }
}

void *wrapper(void *arg)
{
    struct qdata *data = (struct qdata *)arg;

    if (data->depth < 1) {
        quick_sort(data->data, data->size);
    } else {
        if (data->size > 1) {
            int index = partition(data->data, data->size);

            pthread_t t1;
            pthread_t t2;

            struct qdata next;

            next.data = &data->data[index + 1];
            next.size = data->size - index - 1;
            next.depth = data->depth - 1;

            data->size = index;
            data->depth = data->depth - 1;

            int r1 = pthread_create(&t1, NULL, wrapper, data);
            int r2 = pthread_create(&t2, NULL, wrapper, &next);

            if (r1) {
                quick_sort(data->data, data->size);
            } else {
                pthread_join(t1, NULL);
            }

            if (r2) {
                quick_sort(next.data, next.size);
            } else {
```

```
                pthread_join(t2, NULL);
            }
        }
    }
}

int main(int argc, char *argv[])
{
    int size;
    int depth;
    struct timeval t;

    if (argc < 2) {
        printf("Specify␣array␣size␣as␣parameter!\n");

        return -1;
    }

    size = atoi(argv[1]);

    if (size <= 0) {
        printf("Size␣must␣be␣a␣positive␣integer.\n");

        return -1;
    }

    if (argc > 2) {
        depth = atoi(argv[2]);

        if (depth < 0) {
            printf("Depth␣must␣be␣zero␣or␣a␣positive␣integer!\n
               ");

            return -1;
        }
    } else {
        depth = 4;
    }

    int *array = malloc(size * sizeof(int));

    init(array, size);
```

```
    start_timer(&t);
    quick_sort(array, size);
    end_timer(&t, array, size, 0);

    init(array, size);

    struct qdata data;

    data.data = array;
    data.size = size;
    data.depth = depth;

    pthread_t thread;

    start_timer(&t);

    int rval = pthread_create(&thread, NULL, wrapper, &data);

    if (rval) {
        quick_sort(data.data, data.size);
    } else {
        pthread_join(thread, NULL);
    }

    end_timer(&t, array, size, 1);

    free(array);
}
```

QuickSort.java

```
import java.util.Random;

public class QuickSort {

    private static int[] createArray(int size) {
        int[] array = new int[size];
        Random random = new Random(System.nanoTime());

        for (int i = 0; i < array.length; i++) {
            array[i] = random.nextInt(Integer.MAX_VALUE);
        }
```

```
    return array;
}

private static int partition(int[] data, int low, int high)
    {
    int i = low + 1;
    int j = high;
    int pivot = data[low];
    int tmp;

    while (i <= j) {
        while (i <= j && data[i] <= pivot) {
            i++;
        }

        while (j >= i && data[j] >= pivot) {
            j--;
        }

        if (i < j) {
            tmp = data[i];
            data[i] = data[j];
            data[j] = tmp;
        }
    }

    tmp = data[low];
    data[low] = data[j];
    data[j] = tmp;

    return j;
}

private static void quickSort(int[] data, int low, int high
   ) {
    if ((high - low + 1) > 1) {
        int index = partition(data, low, high);

        quickSort(data, low, index - 1);
        quickSort(data, index + 1, high);
    }
}
```

```
private static class Wrapper implements Runnable {

    private int[] data;
    private int low, high, depth;

    public Wrapper(int[] data, int low, int high, int depth
       ) {
        this.data = data;
        this.low = low;
        this.high = high;
        this.depth = depth;
    }

    public void run() {
        if (depth < 1) {
            quickSort(data, low, high);
        } else {
            if ((high - low + 1) > 1 ){
                int index = partition(data, low, high);

                Thread t1 = new Thread(new Wrapper(data,
                   low, index - 1, depth - 1));
                Thread t2 = new Thread(new Wrapper(data,
                   index + 1, high, depth - 1));

                t1.start();
                t2.start();
                try {
                    t1.join();
                    t2.join();
                } catch (InterruptedException e) {
                    // ignore
                }
            }
        }
    }

}

private static boolean isSorted(int[] data, int size) {
    for (int i = 0; i < size - 1; i++) {
        if (data[i] > data[i + 1]) {
```

```
            return false;
        }
    }

    return true;
}

public static void main(String[] args) throws
   InterruptedException {
    int size = 0;

    if (args.length < 1) {
        System.err.println("Specify␣array␣size␣as␣parameter
           !");
        System.exit(1);
    }

    try {
        size = Integer.parseInt(args[0]);
    } catch (NumberFormatException e) {
        System.err.println("Parameter␣must␣be␣a␣positive␣
           integer!");
        System.exit(1);
    }

    if (size <= 0) {
        System.err.println("Parameter␣must␣be␣a␣positive␣
           integer!");
        System.exit(1);
    }

    int depth = 4;

    if (args.length > 1) {
        try {
            depth = Integer.parseInt(args[1]);
        } catch (NumberFormatException e) {
            System.err.println("Depth␣must␣be␣zero␣or␣a␣
               positive␣integer!");
            System.exit(1);
        }

        if (depth < 0) {
```

```
                System.err.println("Depth␣must␣be␣zero␣or␣a␣
                    positive␣integer!");
                System.exit(1);
            }
        }

        int[] array;
        long start;
        long end;

        array = createArray(size);
        start = System.currentTimeMillis();

        quickSort(array, 0, array.length - 1);

        end = System.currentTimeMillis();

        if (isSorted(array, array.length)) {
            System.out.println("Sorted␣in␣" + (end - start) + "
                ␣milliseconds.");
        } else {
            System.out.println("Sorting␣failed!");
        }

        array = createArray(size);
        start = System.currentTimeMillis();

        Thread thread = new Thread(new Wrapper(array, 0 , array
            .length - 1, depth));
        thread.start();
        thread.join();

        end = System.currentTimeMillis();

        if (isSorted(array, array.length)) {
            System.out.println("Sorted␣with␣threads␣in␣" + (end
                 - start) + "␣milliseconds.");
        } else {
            System.out.println("Sorting␣with␣threads␣failed!");
        }
    }

}
```