Die Programmiersprache Go

Freier Software Abend
Köln
4. November 2019

Harald Weidner
hweidner@gmx.net

Die Anfänge von Go

- Entwickelt seit 2007 bei Google
  - Erste Ideen in 45-minütigen Compilier-Kaffeepausen
  - Unzufriedenheit mit C/C++, Java und Skriptsprachen
- Ansatz: C
  - – alles, was unsichere Programmierung fördert
  - – alles, was den Compiler langsam macht
  - + moderne Datentypen und Objektsystem
  - + Nebenläufigkeit / Parallelität
  - + umfangreiche Standardbibliothek
  - + moderne Toolchain

Die Erfinder von Go

Robert Grisemer

- Java Hotspot VM

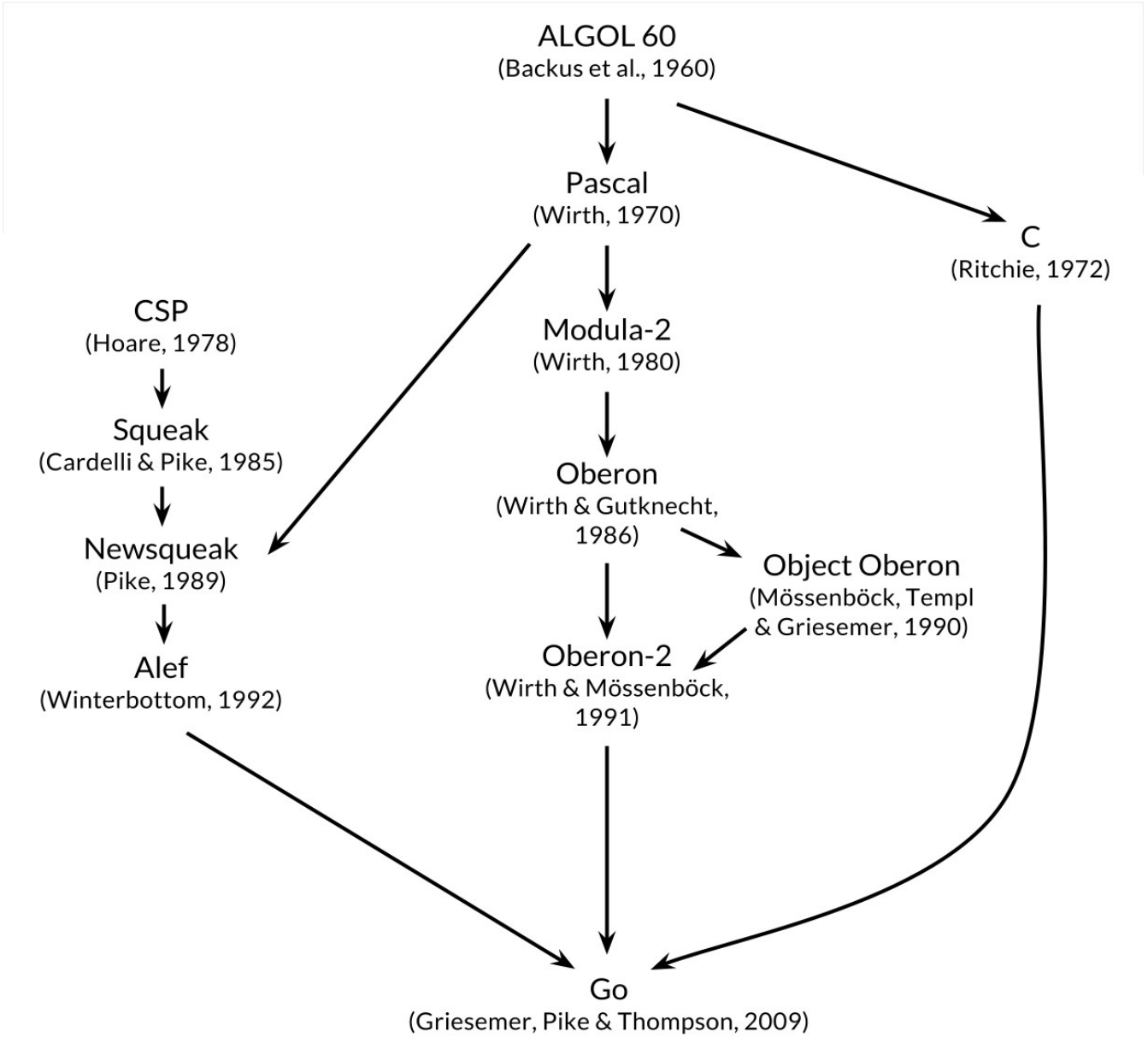
Rob Pike

- Plan 9, Newsqueak, UTF-8

Ken Thomson

- B, Multics, Plan 9, UTF-8

Go Stammbaum



Quelle:

A. Donovan,
B. Kernighan:

The Go Programming Language

Addison-Wesley, 2016

Go Timeline

- Erste Entwicklung seit 2007
- Erste Veröffentlichung im Nov. 2009
  - BSD-Style Lizenz
- Go 1.0 im März 2012
  - Go 1 Kompatibilitätsversprechen
- Alle 6 Monate ein Minor Release (Feb. / Aug.)
  - Jeweils 12 Monate Support für ein Release
  - Patch Releases bei Bedarf
- Aktuell: Go 1.13.4 (Go 1.13 seit 4.9.2019)

Go Compiler

Go Frontend für GCC

- Debian-Paket: gccgo
- Dynamisch gelinkte Binaries
- Viele Plattformen und Betriebssysteme
- Teilweise bessere Performance der Programme

Go Compiler (Gc)

- Debian-Paket: golang
- Statisch gelinkte Binaries
- Verfügbar für i386, amd64, arm, ppc, mips, sparc, s390, wasm
- Linux, *BSD, AIX, Plan 9, Windows, MacOSX, Android, Solaris, NaCl
- Der schnellere Compiler

Bekannte Anwendungen in Go

Cloud und Infrastruktur

- Docker, RKT, Kubernetes, Juju, uRoot, LXD, Terraform, etcd, Consul, KataContainers, gVisor, CloudFoundry, ...

Datenbanken

- Prometheus, Grafana, InfluxDB, TiDB, Dgraph, Vitess, ...

WWW und webbasierte Anwendungen

- Caddy, Traefik, Hugo, Perkeep, Gitea, Gogs, Mattermost, Keybase, ...

Beispiel: hello.go

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, Gophers!")
}
```

Modularisierung durch Packages

- Programme bestehen aus Packages
  - Ein oder mehrere Sourcecode-Dateien pro Package
  - I.d.R.: ein Package = ein Verzeichnis
- Exportierte Bezeichner beginnen mit Großbuchstaben
  - Gilt für **alles**: Variablen, Konstanten, Typen, Interfaces, Funktionen, Methoden, struct-Elemente
  - Alles andere ist nicht außerhalb des Package sichtbar
- Import ungenutzter Packages ist ein Fehler!
- Keine zirkulären Abhängigkeiten zwischen Packages
  - Schnelle Builds durch Caching, parallele Compilierung
- Pseudo-Packages: builtin, C, unsafe
- Neu ab Go 1.12: Modules (Sammlung von Packages)

Go Syntax – Schlüsselwörter

Nur 25 Schlüsselwörter

- dürfen nicht anderweitig verwendet werden
- garantiert keine Änderungen in Go 1
  (Grund: Go 1 Kompatibilitätsversprechen)

`break`	`default`	**`func`**	`interface`	`select`
`case`	`defer`	`go`	`map`	`struct`
`chan`	`else`	`goto`	**`package`**	`switch`
`const`	`fallthrough`	`if`	`range`	**`type`**
`continue`	`for`	**`import`**	`return`	**`var`**

Go Syntax – vordefinierte Namen

20 vordefinierte Typen

```
bool         error     int8    rune     uint16
byte         float32   int16   string   uint32
complex64    float64   int32   uint     uint64
complex128   int       int64   uint8    uintptr
```

4 vordefinierte Konstanten

```
nil   false   true   iota
```

15 vordefinierte Funktionen

```
append   complex   imag   new     println
cap      copy      len    panic   real
close    delete    make   print   recover
```

Go Syntax – Datentypen

Zusammengesetzte Datentypen

```
var a [32]byte                  // Array
var s []string                  // Slice
var m map[string]int            // Map
var p *int                      // Pointer
var f func(int32) int64         // 1st Class Funktion

type IPv6 [16]uint8             // Typdefinition

type Person struct {            // Struct
    Name, Vorname string
    Alter         uint
}
```

Go Syntax – Kontrollstrukturen

```
// for-Schleife
for i:=0; i<10; i++ {
  fmt.Println(i)
}

// while-Schleife
for i<=10 { … }

// Endlosschleife
for { … }
```

```
if a == 0 {
  return 0
} else if a > 0 {
  return 1
} else {
  return -1
}
```

```
switch m {
default:
  foo()
case 0, 2, 4, 6:
  bar()
case 1, 3, 5, 7:
  baz()
}
```

Go Syntax – Funktionen

```
func mult(a, b int64) int64 {
  return a * b
}

func add(a, b int64) (c int64) {
  c = a + b
  return
}

func div_mit_rest(a, b int64) (q, r int64) {
  q = a / b
  r = a % b
  return
}
```

Sichere Software mit Go (1)

Verzicht auf gefährliche Konstrukte

- Garbage Collection statt manueller Speicherverwaltung
- Starke, statische Typisierung
- Keine automatische Typumwandung
- Vorinitialisierung aller Typen mit Standardwerten
- Keine Compilerwarnungen (aber `go vet`)
- Indexprüfungen bei Array-Zugriffen
- Pointer, aber keine Pointer-Arithmetik
- (Fast) kein undefiniertes Verhalten
- Increment (x++) und Decrement (x--) sind Anweisungen

Sichere Software mit Go (2)

Garbage Collection

- Es ist erlaubt (und guter Stil), Referenzen auf lokale Objekte zu publizieren

```
func answer() *int {
    var i int = 42
    return &i
}
```

- Escape Analysis: Objekt wird automatisch auf dem Heap erzeugt
- Garbage Collector löscht Objekt, wenn keine Referenz darauf mehr existiert

Sichere Software mit Go (3)

Statische Typisierung

- Keine automatische Typumwandlung
- Benamte Typen sind unterschiedlich
  Der Entwickler hat ihnen absichtlich verschiedene Namen gegeben

```
type Celsius float32
type Fahrenheit float32

var a Celsius
var b Fahrenheit

a = b    // Fehler: a und b haben
b = a    // unterschiedliche Typen
```

Sichere Software mit Go (4)

Indexprüfung bei jedem Zugriff auf Array/Slice

```
var a, b [100]int

for i := 0; i < len(a); i++ {
    b[i] = a[i] + a[i+1]
}
// Runtime Error bei i=99
```

- Beeinträchtigung der Performance (Benchmarks)
- In anderen Sprachen schwer zu findende Laufzeitfehler

Objektorientierung in Go

- Methoden
  - (Fast) alle selbstdefinierten Typen können Methoden haben
- Interfaces
  - Trennung von Spezifikation und Implementierung
- Polymorphie
- Komposition
- Keine Konstruktoren
  - Vermeidet langwierige Erzeugung von Objekten
- Keine Destruktoren
  - Aber Finalizer
- Keine Vererbung
  - Vermeidet schwerfällige Klassenhierarchien
- Sichtbarkeitsregeln auf Package-Ebene
- (Noch) keine Generics

Objektorientierung - Methoden

```
type Celsius float32
type Fahrenheit float32

func (c Celsius) Print() {
  fmt.Printf("%.1f°C", float32(c))
}

func (f Fahrenheit) Print() {
  fmt.Printf("%.1f°F", float32(f))
}

func main() {
  x := Celsius(20)
  y := Fahrenheit(65)
  x.Print()      // 20.0°C
  y.Print()      // 65.0°F
}
```

Objektorientierung - Polymorphie

- Polymorphie durch Interfaces

```
type Printer interface {
    Print()
}

func main() {
  var p Printer
  p = Celsius(20)
  p.Print()                 // 20.0°C
  p = Fahrenheit(65)
  p.Print()                 // 65.0°F
}
```

Objektorientierung - Komposition

```
type LockableTime struct {
    time.Time     // Methoden Hour(),
                  // Minute(), Second()
    sync.Mutex    // Methoden Lock(),
                  // Unlock
}

var lt LockableTime

lt.Lock()
fmt.Println(lt.Hour(), lt.Minute(), lt.Second())
lt.Unlock()
```

- „Flache“ Nutzung der Methoden der eingebetteten Typen
- Als Ersatz für (Mehrfach-)Vererbung oft ausreichend

Nebenläufigkeit in Go (1)

- **Communicating Sequential Processes (CSP)**
  - Tony Hoare (University of Oxford), 1978
  - Prozessalgebra zur Beschreibung von Interaktionen zwischen unabhängigen Prozessen
- Goroutine
  - Funktion, die nebenläufig ausgeführt wird
  - Syntax: `go f()` oder als Closure: `go func() {…}()`
  - Implizite Verwendung, z.B. durch Bibliothek net/http
- Channel
  - Typisierter Kanal (Kommunikation, Synchronisation)
  - Buffered oder unbuffered

Nebenläufigkeit (Beispiel)

```
func fib(c chan int, n int) {
   x, y := 0, 1
   for x <= n {
      c <- x
      x, y = y, x+y
   }
   close(c)
}

func main() {
   c := make(chan int)
   go fib(c, 1_000_000)

   for f := range c {
      fmt.Println(f)
   }
}
```

Nebenläufigkeit in Go (2)

- Goroutinen sind wesentlich leichtgewichtiger als Betriebssystem-Threads
  - Initiale Stackgröße 2 kB (1 Mio. Goroutinen = 2 GB)
  - Scheduler in Go Runtime verteilt Goroutinen auf Threads des Betriebssystems
- Parallelität durch OS-Threads
  - Gesteuert durch Env.-variable GOMAXPROCS
  - Default (seit Go 1.5): Anzahl CPUs
  - Oder explizit im Programm:
    `runtime.GOMAXPROCS(16)`

Go Toolchain

`go build`	Package compilieren
`go clean`	Compilatdateien löschen
`go doc`	Dokumentation aus Quelltext extrahieren
`go env`	Für Go relevantes Environment anzeigen
`go fix`	Quelltext-Reparaturen ausführen
`go fmt`	Quelltext formatieren
`go generate`	Codegenerierung anstoßen
`go get`	Package herunterladen
`go install`	Package installieren
`go list`	Package anzeigen
`go run`	Programm compilieren und ausführen
`go test`	Unit Tests / Benchmarks ausführen
`go tool`	Tool aus der Go Suite ausführen
`go version`	Version anzeigen
`go vet`	Probleme im Quelltext suchen

Weiterführende Informationen

- Go Homepage http://golang.org/
- Tutorial http://tour.golang.org/
- Playground http://play.golang.org/
- Golang Book http://golang-book.com/
- Language Design in the Service of Software Engineering
  http://talks.golang.org/2012/splash.article
- Less is exponentially more
  http://commandcenter.blogspot.de/2012/06/less-is-exponentially-more.html
- Another Go at Language Design
  http://web.stanford.edu/class/ee380/Abstracts/100428-pike-stanford.pdf
- Building Large-Scale Distributed Systems
  http://static.googleusercontent.com/media/research.google.com/de//people/jeff/stanford-295-talk.pdf