Die Programmiersprache Go

OpenRheinRuhr 2015, Oberhausen
7. November 2015

Harald Weidner
hweidner@gmx.net

Die Programmiersprache Go

„Go, otherwise known as Golang, is an open source, compiled, garbage-collected, concurrent system programming language.“
[http://en.wikipedia.org/wiki/Go_(programming_language)]

„It’s a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language.“
[http://golang.org/doc]

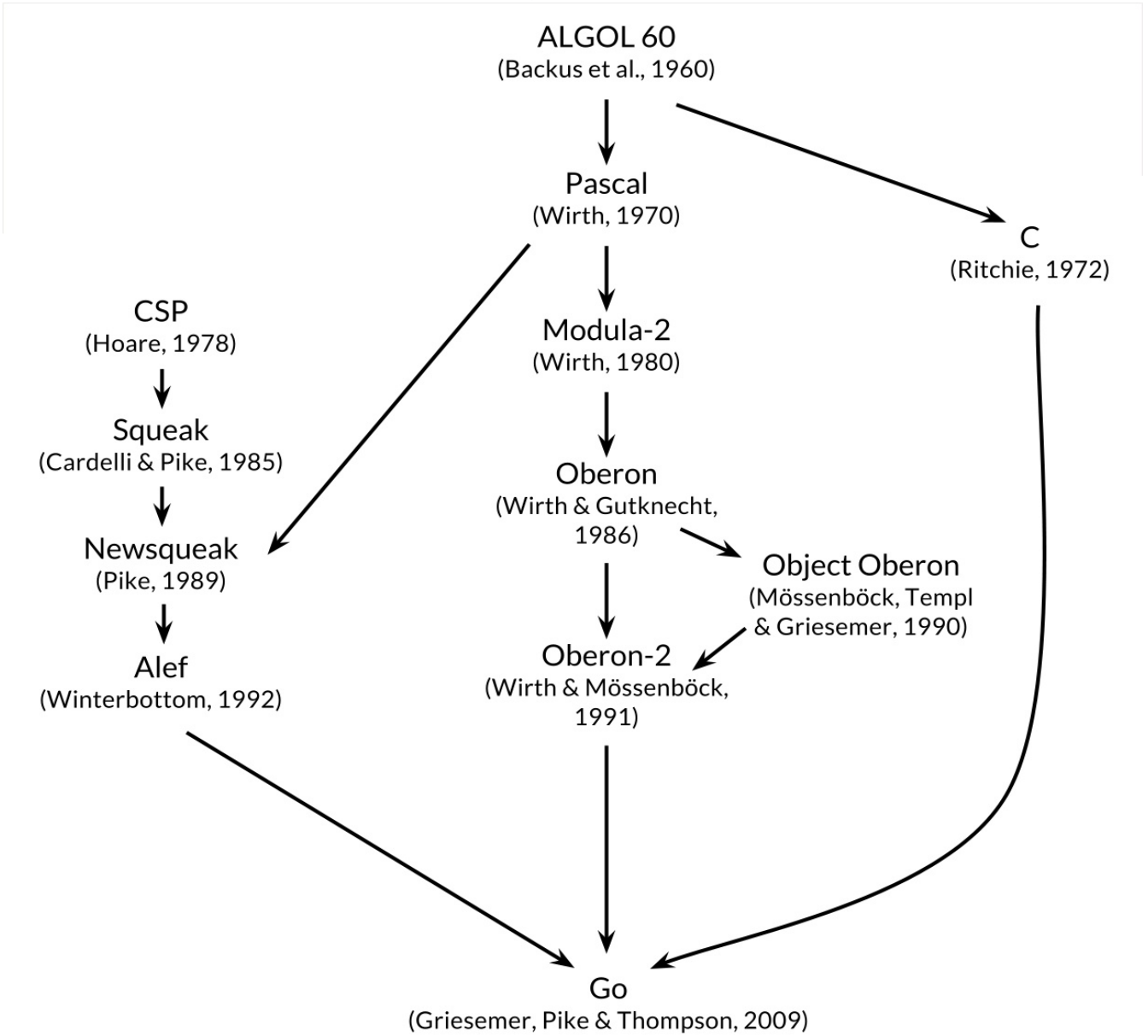
Die Anfänge

- Entwickelt seit 2007
  - 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
  - + Nebenläufigkeit / Parallelismus
  - + moderne Datentypen und Objektsystem
  - + umfangreiche Standardbibliothek
  - + moderne Toolchain

Autoren

- Ursprüngliche Autoren
  - Ken Thompson (B, Multics, Unix, Plan 9, UTF-8)
  - Rob Pike (Plan 9, Newsqueak, UTF-8)
  - Robert Griesemer (Java HotSpot VM)
- Weitere Autoren (u.a.)
  - Russ Cox (Compiler GC)
  - Ian Lance Taylor (Compiler GccGo)
  - Brad Fitzpatrick (Teile der Standardbibliothek)
  - Andrew Gerrand (Release Manager)

Abstammung



Quelle:

A. Donovan,
B. Kernighan:

The Go Programming Language

Addison-Wesley, 2016

Versionsgeschichte

2009: erste Veröffentlichung

2012: Go 1.0

2013: Go 1.1, Go 1.2

- Precise Garbage Collection, Three Index Slice

2014: Go 1.3, Go 1.4

- Garbage Collection in Go, Umzug nach GitHub

2015: Go 1.5

- Compiler und Runtime in Go (+ Assembler)

Go Compiler

Go Frontend für GCC

- Debian-Paket: gccgo
- Dynamisch gelinkte Binaries
- Viele Plattformen, u.a. i386, amd64, arm, mips, ia64, s390, ppc, ...
- Derzeit oft bessere Performance der Programme

Gc von Google

- Debian-Paket: golang
- Statisch gelinkte Binaries
- Verfügbar für i386, amd64, arm, arm64, sparc
- Linux, FreeBSD, Windows, Apple, Plan 9
- Derzeit der schnellere Compiler

Designkriterien

- Einfache Sprache
- Große Projekte mit vielen Entwicklern
- Sichere Software
- Skalierbare Software
- Große, verteilte Umgebungen
- Einfaches Rollout

Einfache Sprache

- Schnell zu erlernen
  - Für erfahrene Programmierer in drei Tagen
- (Vergleichsweise) wenige Features
  - die sich kombinieren lassen
  - die in vorhersagbarer Weise interagieren
- Lesbare Programme
  - Optimierung auf Lesen, nicht Schreiben
- Go 1 Kompatibilitätsgarantie
  - https://golang.org/doc/go1compat

Einfache Sprache

- Kein Forschungsbeitrag zur Theorie der Programmiersprachen
- Sondern eine Sprache im Dienste des Software Engineerings
- „Less is more“

Beispiel: hello.go

```
package main

import "fmt"

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

Einfache Sprache

Nur 25 Schlüsselwörter

- dürfen nicht anderweitig verwendet werden
- garantiert keine Änderungen in Go 1

`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`**

Einfache Sprache

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
```

Einfache Sprache

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 ip6 [16]uint8              // Typdefinition

type person struct {            // Struct
    name, vorname string
    alter uint8
}
```

Große Projekte

Große Projekte mit vielen Entwicklern

- Schneller Compiler
- Strenge Modularität, einfache Exportregeln
- Objektorientierung, Polymorphie, Komposition
- Kanonische Codeformatierung mit `go fmt`
- Toolunterstütztes Refactoring mit `go fix`
- Dokumentation im Quelltext
- Unit Testing, Benchmarking, Profiling

Große Projekte

- C/C++ Compiler sind langsam
  - Ältere, historisch gewachsene Compiler
  - Umfangreiche Sprachen, komplexe Parser
  - Zirkuläre Abhängigkeiten der Header
- Go: keine zirkulären Abhängigkeiten
  - Wenn Modul A von B und dieses von C abhängt:
  - Compiliere zuerst C, dann B, dann A
  - Compiler für A liest nur Schnittstellen von B; diese enthalten bei Bedarf Informationen über C

Große Projekte

- Auf einem Mac (OS X 10.5.7, GCC 4.0.1):
  - C: `#include <stdio.h>`
    liest 360 Zeilen aus 9 Dateien
  - C++: `#include <iostream>`
    liest 25.326 Zeilen aus 131 Dateien
  - Objective-C: `#include <Cocoa/Cocoa.h>`
    liest 112.047 Zeilen aus 689 Dateien
- Go: `import "fmt"`
  - Liest 195 Zeilen aus **einer** Datei
    (enthält Infos über 6 abhängige Packages)

[Quelle: http://web.stanford.edu/class/ee380/Abstracts/100428-pike-stanford.pdf]

Große Projekte

Go ist modular

- Programme bestehen aus Modulen (Packages)
  - Ein oder mehrere Sourcefiles 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 Module ist ein Fehler!
- Pseudo-Packages: builtin, C, unsafe

Große Projekte

Go ist objektorientiert

- Nicht zwingend, nur wenn es sinnvoll ist
- Keine Klassen
  - (Fast) jeder selbstdefinierte Typ kann Methoden haben
- Keine Vererbung
  - Vererbung führt zu schwerfälligen Klassen-hierarchien, Basisklassen schwer änderbar
  - Vererbung verwässert Trennung von Schnittstelle und Implementierung

Große Projekte

- Polymorphie durch Interfaces

```
type Printer interface {
    Print()
}

type A int
type B int
func (a A) Print() { fmt.Printf("<%d>\n", int(a)) }
func (b B) Print() { fmt.Printf("[%d]\n", int(b)) }

a := A(1); b := B(2)
var p Printer
p = a; p.Print()     // ergibt <1>
p = b; p.Print()     // ergibt [2]
```

Große Projekte

- Komposition durch Interfaces

```
// aus dem Package io
type Reader interface { Read(p []byte) (n int, err error) }
type Writer interface { Write(p []byte) (n int, err error) }

f1, _ := os.Open("/etc/passwd")
f2, _ := os.Open("/etc/group")
f3, _ := os.Create("/tmp/pwdgrp")

var r io.Reader
r = io.MultiReader(f1, f2)
r = io.LimitReader(r, 4096)
r = io.TeeReader(r, f3)

io.Copy(os.Stdout, r)
```

[Beispiel – es fehlen das Schließen der Dateien und Fehlerbehandlung!]

Große Projekte

- Komposition durch Einbettung

```
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()
```

Große Projekte

Umfangreiche Toolchain

- Code-Formatierung vorgegeben durch `go fmt`
- Dokumentation aus Kommentaren im Sourcecode
- Unit Testing
- Benchmarking
- Profiling
- Unterstützung für Refactoring

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 ausführen
`go tool`	Tool aus der Go Suite ausführen
`go version`	Version anzeigen
`go vet`	Probleme im Quelltext suchen

Sichere Software

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
- Kein undefiniertes Verhalten
- Increment (x++) und Decrement (x--) sind Anweisungen

Sichere Software

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

Statische Typisierung

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

```
type A int
type B int

var a A
var b B

a = b   // Fehler: A und B sind
b = a   // unterschiedliche Typen
```

Sichere Software

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

Skalierbare Software

Moderne Computer haben mehrere CPUs, Multicore, Hyperthreading

- Ältere Sprachen haben allenfalls nachträgliche Unterstützung zur Nutzung von Parallelität

Goroutinen

```
foo(x)      // Funktionsaufruf
go bar(y)   // Goroutine
```

- Nebenläufige Ausführung von Programmcode
- Runtime verteilt Goroutinen auf Threads

Skalierbare Software

- Goroutinen sind wesentlich leichtgewichtiger als Betriebssystem-Threads
  - Initiale Stackgröße 2 kB (1 Mio. Goroutinen = 2 GB)
  - Task-Switching (Go-Compiler kennt die benutzen CPU-Register)
- Goroutinen können benutzt werden, um unabhängige Programmteile nebenläufig ablaufen zu lassen
  - Beispiel HTTP Server aus Go Standardbibliothek: eine Goroutine für jeden HTTP-Request

Skalierbare Software

Kommunikation der Goroutinen über Channel

```
c := make(chan int, 1)   // Channel

c <- 42                  // Schreiben
x = <- c                 // Lesen
```

- first class, thread-safe
- gepuffert oder ungepuffert

„Don't communicate by sharing memory – share memory by communicating“

Große verteilte Umgebungen

- Explizite Fehlerbehandlung
- Netzwerkfunktionen in Standardbibliothek
- Explizite Timeouts
- Netzwerkkommunikation, RPC, REST
- Serialisierung, JSON, Protocol Buffers

Große verteilte Umgebungen

Fehlerbehandlung

- In großen verteilten Systemen sind Fehler normal
- Jeder Programmteil muss stets mit Fehlern rechnen und geeignete Maßnahmen treffen
- Fehlerbehandlung ist expliziter Teil des Ablaufs
- Keine Exceptions (aber `panic/recover`)

Große verteilte Umgebungen

Fehlerbehandlung

- Eingebauter Typ error als Interface

```
type error interface {
    Error() string
}
```

- Funktionen, die scheitern können, geben Fehler als Rückgabewert zurück

```
func try() error {
    …
}
```

- Im Erfolgsfall gilt `error == nil`

Einfaches Deployment

- Von GC erzeugte Binaries
  - Statisch gelinkt (Default)
  - Nur gegen glibc gelinkt (CGO)
- Von GccGo erzeugte Binaries
  - Dynamisch gegen eine libgo gelinkt
  - libgo enthält Standardbibliothek und Runtime

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