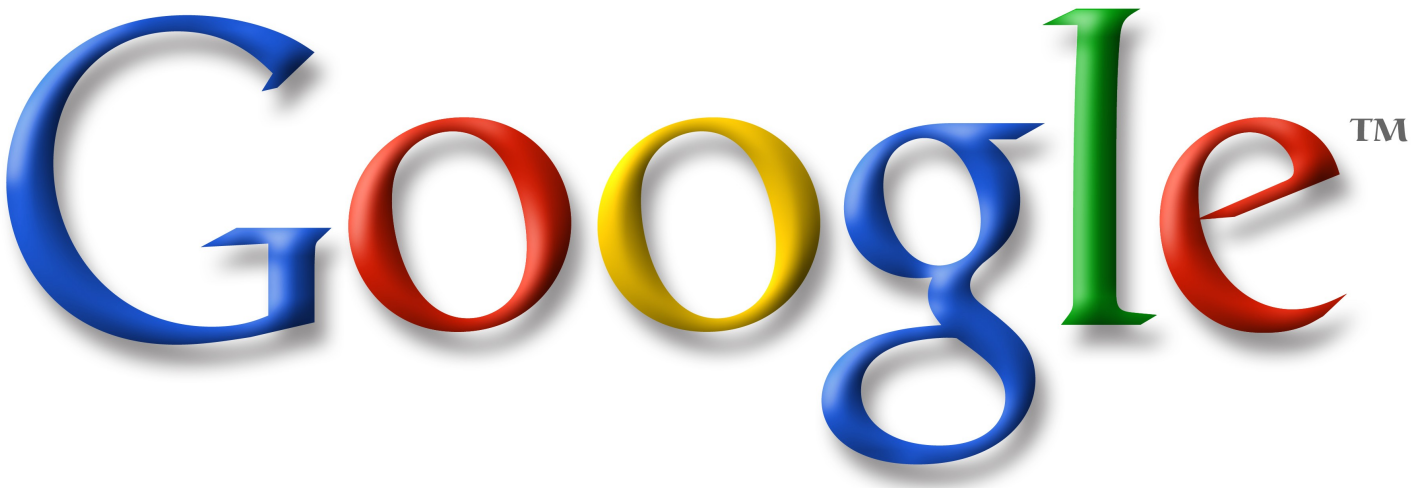
The Programming Language Go

Jessica Pavlin
Department of Computing and Software
McMaster University

November 16, 2010

Outline

Who Designed and Implemented Go?



A Very Brief History

Sept. 2007	Robert Griesemer, Rob Pike and Ken Thompson started sketching the goals for a new language on a white board
Sept. 2007	Within a few days they had their goals and plan sketched out
Sept. 2007	They continued to design the new language whenever they had time
Jan. 2008	Thompson started the compiler
May 2008	Taylor started the gcc front end for Go using specs
Late 2008	Russ Cox joined in helped to implement the language and libraries

A Very Brief History

Sept. 2007 Robert Griesemer, Rob Pike and Ken Thompson started sketching the goals for a new language on a white board

Sept. 2007 Within a few days they had their goals and plan sketched out

Sept. 2007 They continued to design the new language whenever they had time

Jan. 2008 Thompson started the compiler

May 2008 Taylor started the gcc front end for Go using specs

Late 2008 Russ Cox joined in helped to implement the language and libraries

Motivation for a New Language

- Frustration with existing languages and environments for systems programming
- They felt that programming languages were partly to blame for programming becoming “too difficult”
- Didn’t want to have to choose anymore between:
  - Efficient compilation
  - Efficient execution
  - Ease of programming

Goals for a New Language

1. As easy to program as an interpreted, dynamically typed language
2. The efficiency and safety of a statically typed, compiled language
3. Modern: support for networked and multicore computing
4. Fast
5. Make programming fun again

A Very Brief History

Sept. 2007	September 2007: Robert Griesemer, Rob Pike and Ken Thompson started sketching the goals for a new language on the white board
Sept. 2007	Within a few days they had their goals and plan sketched out
Sept. 2007	They continued to design the new language whenever they had time
Jan. 2008	Thompson started the compiler
May 2008	Taylor started the gcc front end for Go using specs
Late 2008	Russ Cox joined in helped to implement the language and libraries

A Very Brief History

Sept. 2007	September 2007: Robert Griesemer, Rob Pike and Ken Thompson started sketching the goals for a new language on the white board
Sept. 2007	Within a few days they had their goals and plan sketched out
Sept. 2007	They continued to design the new language whenever they had time
Jan. 2008	Thompson started the compiler
May 2008	Taylor started the gcc front end for Go using specs
Late 2008	Russ Cox joined in helped to implement the language and libraries

A Very Brief History

Sept. 2007	September 2007: Robert Griesemer, Rob Pike and Ken Thompson started sketching the goals for a new language on the white board
Sept. 2007	Within a few days they had their goals and plan sketched out
Sept. 2007	They continued to design the new language whenever they had time
Jan. 2008	Thompson started the compiler
May 2008	Taylor started the gcc front end for Go using specs
Late 2008	Russ Cox joined in helped to implement the language and libraries

A Very Brief History

Sept. 2007	September 2007: Robert Griesemer, Rob Pike and Ken Thompson started sketching the goals for a new language on the white board
Sept. 2007	Within a few days they had their goals and plan sketched out
Sept. 2007	They continued to design the new language whenever they had time
Jan. 2008	Thompson started the compiler
May 2008	Taylor started the gcc front end for Go using specs
Late 2008	Russ Cox joined in helped to implement the language and libraries

Go's Ancestors

- Basic Syntax:
  - C
  - Pascal
- Concurrency:
  - Newsqueak
  - Limbo

Go's Ancestors

- Basic Syntax:
  - C
  - Pascal
- Concurrency:
  - Newsqueak
  - Limbo

Hello World

```
package main

import fmt "fmt"

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

Hello World! in Go

- Basic structure of a Go program

Hello World

```
package main

import fmt "fmt"

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

Hello World! in Go

- All source code requires a package name

Hello World

```
package main

import newName "fmt"

func main() {
    newName.Println("Hello, world")
```

Hello World! in Go

- You can import packages
- Imported packages can have qualified identifiers

Objects in Go

Go is Object-Oriented (OO)-*ish*

- Has types and methods
- Allows for OO programming
- All types can have methods (even integers and strings)
- “Objects” implicitly satisfy interfaces
- Not an OO language:
  - No classes
  - No subclassing

Types

Definition:

- A type determines the set of values and operations specific to values of that type

Syntax:

```
Type      = TypeName | TypeLit |
            "(" Type ")" .
TypeName  = QualifiedIdent.
TypeLit   = ArrayType | StructType |
            PointerType | FunctionType |
            InterfaceType | SliceType |
            MapType | ChannelType .
```

Anonymous Types

Types can be anonymous:

```
type ABC struct {
  x float
  int
  string
}

c := ABC{ 3.5, 7, "hello" }
fmt.Println(c.x, c.int, c.string)
```

An example of an anonymous type

Prints:

```
3.5 7 hello
```

Types Example

```
type Day int

var dayName = []string{"Sunday", "Monday",
                       "Tuesday", "Wednesday",
                       "Thursday", "Friday",
                       "Saturday"}
```

Using integer types as days of the week.

Initializing Types: var vs. :=

Different ways to initialize types:

1. `var`
   - Initializes a *zeroed* instance of the type
   - Initializes int, float, etc., and any new type T
   - `var v1 ABC // 1. type ABC`
2. `:=`
   - Initializes a to a value if provided
   - Compiler guesses type if not p

```
var i int
j := 0      // i == j = true!

var k int
k = 3
l := 3      // k == l = true!
```

Initializing Types: new vs. make

Different ways to initialize types:

1. `new`
   - Returns a *reference* to a newly allocated, zeroed instance of type T
   - Creates new instances of types not listed in 3
   - `v2 := new(ABC)  // 2. type *ABC`
2. `make`
   - Returns an initialized (not zero) value of type T (not T*)
   - Creates slices, maps and channels only
   - `var v3 []int = make([]int, 100)`
     - Slice to reference to new array of 100 ints

Arrays

Arrays are different then they are in languages like C:

- Arrays are values
- Assigning an array to another copies all of the elements
- When passing an array to a function, the function receives a copy of elements
- The size of an array is part of it's type

```
[3]int { 1, 2, 3 }
[10]int { 1, 2, 3 }
[...]int { 1, 2, 3 }
[10]int { 2:1, 3:1, 5:1, 7:1 }
```

Slices Defined

Definition:

- A reference to a contiguous segment of an array and contains a numbered sequence of elements from that array

Syntax:

```
SliceType = "[" "]" ElementType .
```

Understanding Slices

Details:

- Slices wrap arrays to give a more flexible, powerful, and convenient interface to sequences of data
- Conceptually, slices have 3 elements: base array reference, length, capacity
- Run-time data still passed by value (pointer, length and capacity (max length))
- Length of a slice may change (so long as it still fits within the limits of the underlying array)

```
var a []int
a = ar[7:9];
var slice = []int{ 1,2,3,4,5 }
```

Sample initializations of slices

Creating Methods for Types

Syntax:

```
MethodExpr    = ReceiverType "." MethodName .
ReceiverType  = TypeName |
                "(" "*" TypeName ")" .
```

Sample method for our previously defined type

```
func (d Day) String() string {
  if 0 <= d && int(d) < len(dayName) { return
     dayName[d] }
  return "NoSuchDay"
}
```

Using integer types as days of the week.

Interfaces

Definition:

- An interface type specifies a method set called its interface

Syntax:

```
InterfaceType      = "interface"
                     "{" { MethodSpec ";" } "}" .
MethodSpec         = MethodName Signature |
                     InterfaceTypeName .
```

Interface Example

```
type Stringer interface {
  String() string
}

func print(args ...Stringer) {
  for i, s := range args {
    if i > 0 { fmt.Print(" ") }
    fmt.Print(s.String())
    }
}
```

Example of an interface for the Stringer function

```
print(Day(1))
=> Monday
```

Example of a General Interface

```
func print(args ...interface{}) {
  for i, a := range args {
    if i > 0 { fmt.Print(" ") }
    switch a.(type) {
      case Stringer: fmt.Print(a.String())
      case int:      fmt.Print(itoa(a))
      case string:   fmt.Print(a)
    }
  }
}
```

Creating an print method that works for many types

Advantages of using Interfaces

The advantages of using interfaces over similar OO concepts include:

1. A type can satisfy many interfaces
2. The original implementations of the interfaces do not need to know about the what's using it, or even that that interface exists
   - Don't need do explicitly declare dependencies between the types
3. Interfaces are lightweight

Go is a Concurrent Language

Go is concurrent **not parallel**.

- Intended for program structure, not to maximize performance
- However, this style does keep work nicely distributed on a multi-core system

Simple Example

A new flow of control starts whenever you put `go` in front of the work that you want done.

```
func main() {
    go expensiveComputation(x, y, z)
    anotherExpensiveComputation(a, b, c)
}
```

Using a goroutine is similar to a thread, but it's lighter weight since the stacks are small, segmented and sized on demand.

Channels

Channels provide a mechanism for two concurrently executing functions to synchronize execution and communicate.
Syntax:

```
ChannelType = ( "chan" [ "<-" ] | "<-" "chan" )
              ElementType .
```

- ElementType can be any type (int, float, ...)
- New channels are easily made using `make`, by passing it the channel type and size of buffer:

```
make(chan int, 100)
```

Compilers

There are currently two Go compilers:

1. 6g/8g/5g (the compilers for AMD64, x86, and ARM respectively)
2. gccgo: a GCC frontend written in C++
   - Not complete as of last update of documentation
3. Also a very small runtime environment

All run on Unix-like systems and a port to Windows have recently been integrated into main distributions.

Goals Reviewed

Did Go meet it's goals?

- As easy to program as an interpreted, dynamically typed language
  - Yes! Really easy to pick up and code creation is very fast.
- Efficiency and safety of a statically typed, compiled language
  - Yes! Go's Type system is expressive but lightweight
- Modern - support for networked and multicore computing
  - Yes! We've shown that concurrency is easy and well supported in Go
- Fast
  - Yes! Runtimes of Go with standard C implementations show very comparable results
- Make programming fun again

Goals Reviewed

Did Go meet it's goals?

- As easy to program as an interpreted, dynamically typed language
  - Yes! Really easy to pick up and code creation is very fast.
- Efficiency and safety of a statically typed, compiled language
  - Yes! Go's Type system is expressive but lightweight
- Modern - support for networked and multicore computing
  - Yes! We've shown that concurrency is easy and well supported in Go
- Fast
  - Yes! Runtimes of Go with standard C implementations show very comparable results
- Make programming fun again

Goals Reviewed

Did Go meet it's goals?

- As easy to program as an interpreted, dynamically typed language
  - Yes! Really easy to pick up and code creation is very fast.
- Efficiency and safety of a statically typed, compiled language
  - Yes! Go's Type system is expressive but lightweight
- Modern - support for networked and multicore computing
  - Yes! We've shown that concurrency is easy and well supported in Go
- Fast
  - Yes! Runtimes of Go with standard C implementations show very comparable results
- Make programming fun again

Goals Reviewed

Did Go meet it's goals?

- As easy to program as an interpreted, dynamically typed language
  - Yes! Really easy to pick up and code creation is very fast.
- Efficiency and safety of a statically typed, compiled language
  - Yes! Go's Type system is expressive but lightweight
- Modern - support for networked and multicore computing
  - Yes! We've shown that concurrency is easy and well supported in Go
- Fast
  - Yes! Runtimes of Go with standard C implementations show very comparable results
- Make programming fun again

Goals Reviewed

Did Go meet it's goals?

- As easy to program as an interpreted, dynamically typed language
  - Yes! Really easy to pick up and code creation is very fast.
- Efficiency and safety of a statically typed, compiled language
  - Yes! Go's Type system is expressive but lightweight
- Modern - support for networked and multicore computing
  - Yes! We've shown that concurrency is easy and well supported in Go
- Fast
  - Yes! Runtimes of Go with standard C implementations show very comparable results
- Make programming fun again

Goals Reviewed

Did Go meet it's goals?

- As easy to program as an interpreted, dynamically typed language
  - Yes! Really easy to pick up and code creation is very fast.
- Efficiency and safety of a statically typed, compiled language
  - Yes! Go's Type system is expressive but lightweight
- Modern - support for networked and multicore computing
  - Yes! We've shown that concurrency is easy and well supported in Go
- Fast
  - Yes! Runtimes of Go with standard C implementations show very comparable results
- Make programming fun again

Usage

Go is being used on some small scale productions.

- Go is still experimental
- Go is in use internally at Google
  - The server that runs golang.org was written in Go

How You Can Contribute

Go is open source and would like your help!

1. Download it and play and report any bugs to the Issue Tracker
2. Contribute code

References

golang.org:

- Tutorials
- Videos
- Language Definition
- Information on where and how to contribute
- etc.

Others:

- Wikipedia: `http://en.wikipedia.org/wiki/Go_(programming_language)`
- Wikipedia: `http://en.wikipedia.org/wiki/Newsqueak`

Thank you.
Questions?