Exploring Go Language Design

CS263 Spring 2014

Patrick Baxter

Talk Overview

This talk is meant to be both a ger
rationales

- language overview
- language goals
- major design decisions and ph
- interfaces
- concurrency and pipelines

Go Overview

Go Overview

Aimed to replace c and c++ code

- Start with c, remove complex parts
- add inerfaces, concurrency
- garbage collection, closures, strings...
- memory safe

```
package main

import "fmt"

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

Run

Hello

```
package main

import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/", hello)
    http.ListenAndServe(":8000", nil)
}

func hello(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello, 世界")
}
```

Run

Concurrency

- Build in Lightweight threads, no callback hell
- Easy to write servers

```
for {
    conn, err := listener.Accept()
    //check err
    go serve(conn)
}
```

Concurrency

Channels are thread-safe queues used to synchronize go-routines and share control of data.

```
func worker(done chan bool) {
    fmt.Print("working...")
    time.Sleep(time.Second)
    fmt.Println("done")
    done <- true
}
func main() {
    done := make(chan bool, 1)
    go worker(done)

    <-done
}
```

Run

Concurrency

The select statement allows multi-way conncurent control.

```
select {
case v := <-ch1:
    fmt.Println("channel 1 sends", v)
case v := <-ch2:
    fmt.Println("channel 2 sends", v)
default: // optional
        fmt.Println("neither channel was ready")
}
```

Object Oriented Language?

Not objected oriented in the traditional sense. We have types, structures, and method recievers on those strutures.

```
type struct Foo {
    i int
}
func (f *Foo) inc() {
    f.i += 1
}
x := new(Foo) // x is a pointer to Foo
x.inc()  // x is now 1
```

Embedding

Closest thing go has to inheritance. The methods of embedded types can be accessed directly by a structure or interface
that it is embedded in.

```
type Inner struct {
    Data string
}
func (i Inner) String() string {return i.Data}

type Outer struct {
    Inner
}

func main() {
    var o Outer
    o.Data = "This works!"
    fmt.Println(o.String())
}
```

Run

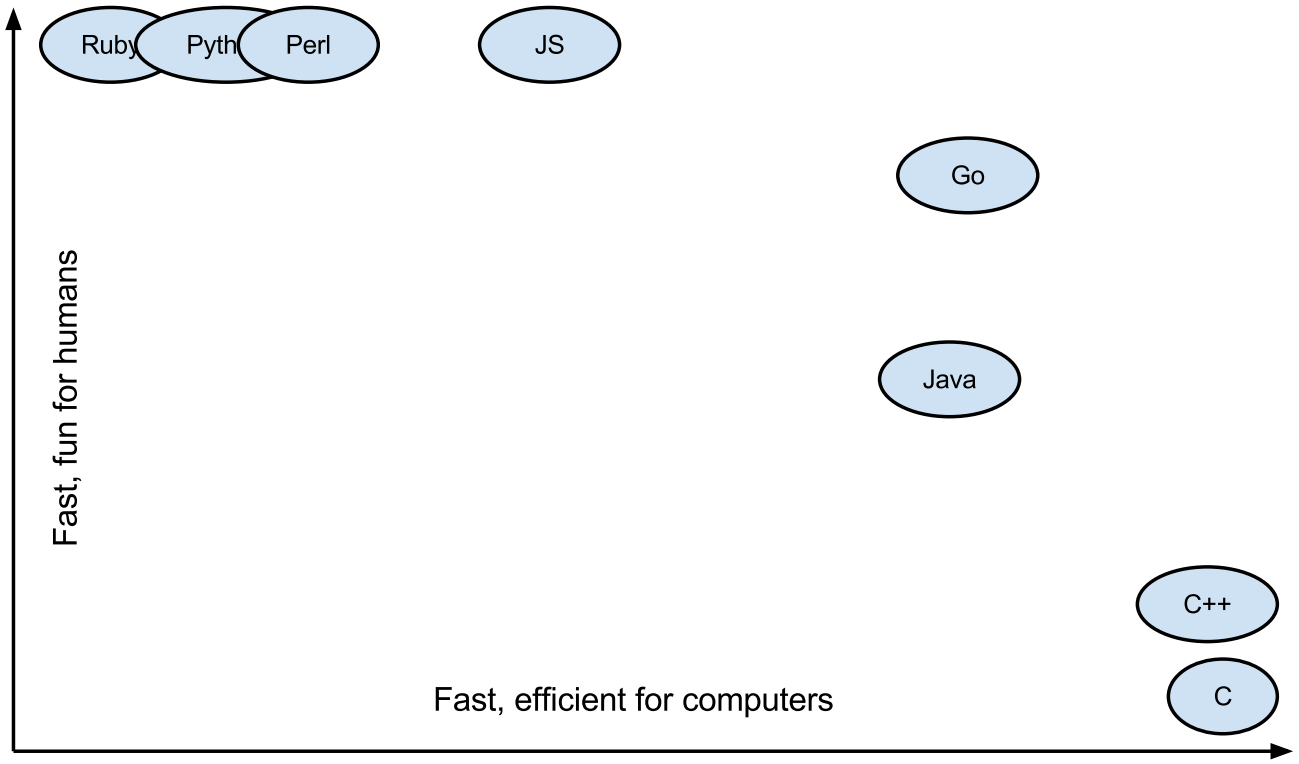
Go Design

Go Design

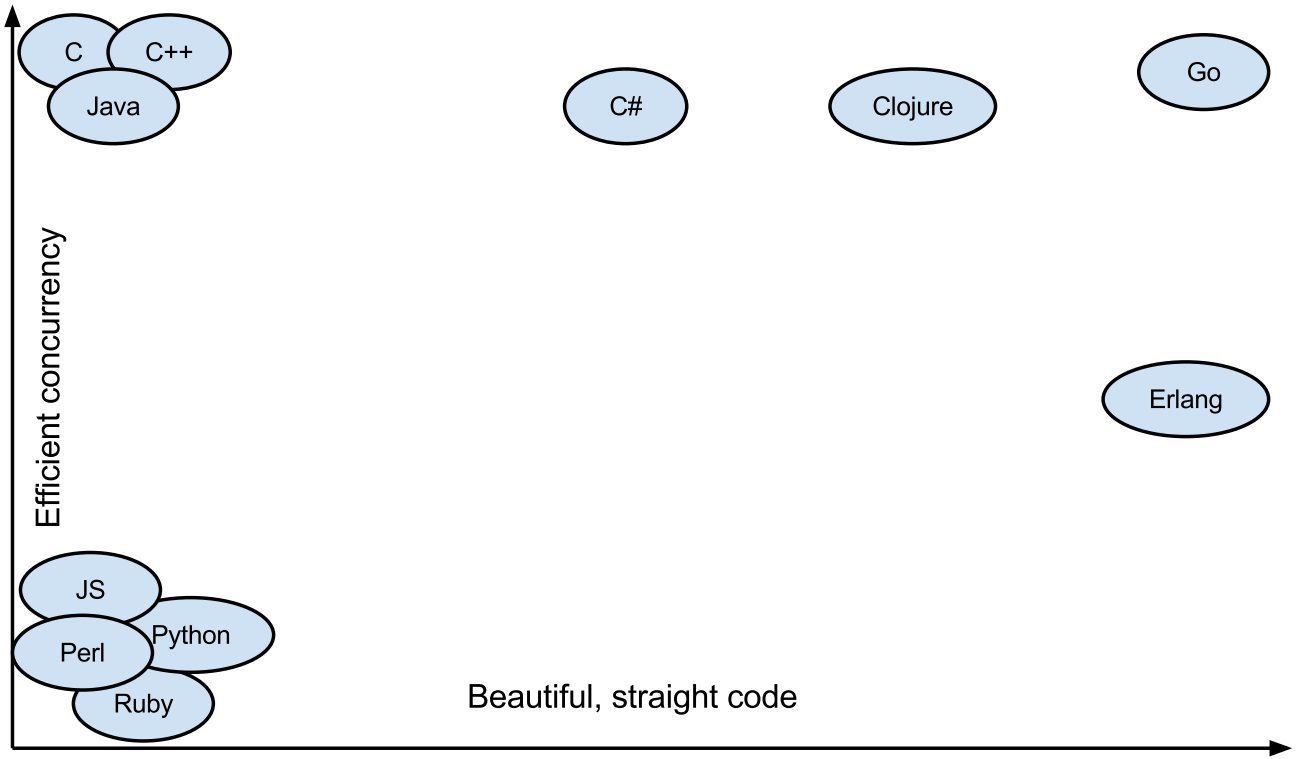
Designed at Google to solve their software problems. They needed a productive language that targetted networked and multicore systems.

- Designed to scale for large projects and for many programmers
- Fast Compile
- Attracts more Python and Java Devs then C++

Fun and fast!


Ruby
Pyth
Perl
JS
Go
Java
C++
C
Fast, fun for humans
Fast, efficient for computers

Concurrent!


C
C++
Java
C#
Clojure
Go
Erlang
JS
Python
Perl
Ruby
Efficient concurrency
Beautiful, straight code

Design philosophy

- simplicity
- orthogonality
- non-goal: break new ground in programming language research

Despite this, Go has many interesting research questions about how to implement Go well.

- concurrency
- polymorphism
- garbage collection

Garbage Collection

Garbage Collection

Garbage collection simplifies APIs

- C and C++ have memory magagement interfering with APIs
- Fundamental to Interfaces as a result

Fundamental to concurreny: tracking ownership is too hard amoung goroutines

This, of course, adds cost, latentcy, and complexity into the runtime

Avoiding Garbage Collection

Go lets you limit allocation by controlling memory layout. Pointers are allowed.

Escape analysis for stacks vs heap

Slices

Implementing Garbage Collection

Allocator: objecst are allocated in pages with out objects of the same size

GC: stop the world, parallel mark, start the world, background sweep

Future research: make the collector lower latency and explore incremental designs

Polymorphism

Interfaces

Interfaces define a set of methods.

```
package io

type Writer interface {
    Write(data []byte) (n int, err error)
}
```

Interfaces

A type implements the interface by implementing the methods

```
package bytes

type Buffer struct {
    ...
}

func (b *Buffer) Write(data []byte) (n int, err error) {
    ...
}
```

Interfaces

An implementation of an interface can be assigned to a variable of that interface type.

```
package fmt

func Fprintf(w io.Writer, format string, args ...interface{})
```

```
    b := new(bytes.Buffer)
    var w io.Writer
    w = b
    fmt.Fprintf(w, "hello, %s\n", "world")
    os.Stdout.Write(b.Bytes())
```

Run

Interfaces

- No dependence between interface and implementation
- avoids overdesign, rigit heirarchy of inheritance-based OO

All generality in the Go language is through interfaces

Implementing Interfaces

How do you make method dispatch effcient

```
b := new(bytes.Buffer)
var w io.Writer
w = b
fmt.Fprintf(w, "hello, %s\n", "world")
    ... w.Write(text) // what happens here?
```

Implementing Interfaces

How do you make method dispatch effcient

```
b := new(bytes.Buffer)
var w io.Writer
w = b                   // do the work here!
fmt.Fprintf(w, "hello, %s\n", "world")
    ... w.Write(text) // what happens here?
```

Interface holds two words: "itable" and actual value or pointer to value

Itable contains type information plus list of function pointers for methods in interface.

One extra indirection, no branching.

Interfaces benchmark

Lets test the effciency of interface dispatch

```
type one struct {
    a string
    ...
}
func (s *one) Append(str string) {
}
...
type two struct {
    b int
    ...
}
...

type intAppender interface {
    GetInt() int
    Append(string)
}
```

Interfaces benchmark

```
//Takes list of intAppenders and uses interface functions to modify array
func genericAppend(s []intAppender) {
    for _,item := range s {
        item.Append("x")
        x := item.GetInt()
        x++
    }
}

//Takes underlying struct and calls methods directly
func staticAppend(s []one) {
    for _,item := range s {
        item.Append("x")
        x := item.GetInt()
        x++
    }
}
```

Run

Interfaces for algorithms

`sort.Interface` describes the operations required to sort a collection:

```
type Interface interface {
    Len() int
    Less(i, j int) bool
    Swap(i, j int)
}
```

`IntSlice` can sort a slice of ints:

```
type IntSlice []int

func (p IntSlice) Len() int           { return len(p) }
func (p IntSlice) Less(i, j int) bool { return p[i] < p[j] }
func (p IntSlice) Swap(i, j int)      { p[i], p[j] = p[j], p[i] }
```

`sort.Sort` uses can sort a `[]int` with `IntSlice`:

```
    s := []int{7, 5, 3, 11, 2}
    sort.Sort(IntSlice(s))
    fmt.Println(s)
```

Run

Interfaces and Containers

This is were people feel like generics are missing from go. There is no easy way to implement a polymorphic container.

```
list := []interface{}{0, "cat", "bat", true, 1239}
for _, item := range list {
    fmt.Println(item)
}
for _, item := range list {
    switch v := item.(type) {
    case int:
        fmt.Println(v, "(int)")
    case string:
        fmt.Println(v, "(string)")
    case bool:
        fmt.Println(v, "(bool)")
    }
}
```

Run

There is a cost to this, both in the conversion of types into the generic container and the type switch that may be necessary upon taking an item out of the container.

Interfaces and Containers

In practice, most containers only need to hold a single type. This means its usually easiest to just rewrite cointainers
to be type specific.

Polymorphism: Can Go do better?

This has been an outstanding question from the go developers for awhile. They aren't happy with any of the existing
implementation of generics:

- C says don't bother
- C++ makes many copies of the same function (templates)
- Java boxes everything implicity: expensive

Slow programmers, slow compilers and large binaries, or slow execution?

Concurrency

Concurrency:CSP

Go concurrency is a varient of the CSP style model with first-class channels. Remains orthogonal to the rest of the language design.

- Famililar model of computation
- Easy composition of regular code

CSP - "Communicating Sequential Processes" - C.A.R. Hoare 1978
"Don't communicate by sharing memory, share memory by communicating."

- goroutines: light-weight threads, has own stack and local variables
- channels: thread-safe queues that carry typed messages between goroutines
- select: based partially off unix epoll to achieve asynchrony and performance

Caveat: not purely memory safe, goroutines share memory

Goroutines

What is a goroutine? It's an independently executing function, launched by a go statement.

It has its own call stack, which grows and shrinks as required.

It's very cheap. It's practical to have thousands, even hundreds of thousands of goroutines.

There might be only one thread in a program with thousands of goroutines.

Instead, goroutines are multiplexed dynamically onto threads as needed to keep all the goroutines running.

Conncurrency is not parallelism

Uses a channel send instead of a python yeild

```
func fib(n int) chan int {
    c := make(chan int)
    go func() {
        a, b := 0, 1
        for i := 0; i < n; i++ {
            a, b = b, a+b
            c <- a
        }
        close(c)
    }()
    return c
}

func main() {
    for x := range fib(10) {
        fmt.Println(x)
    }
}
```

Run

Concurrency enables parallelism

blog.golang.org/pipelines (http://blog.golang.org/pipelines)

Consider a pipeline with three stages:

```
func gen(nums ...int) <-chan int
func sq(int <- chan int) <-chan int

func main() {
    //set up the pipeline
    c := gen(2,3)
    out := sq(c)

    //consume the output.
    fmt.Println(<-out) // 4
    fmt.Println(<-out) // 9
}
```

Pipelines

or we can do:

```
func main() {
    // Set up the pipeline and consume the output.
    for n := range sq(sq(gen(2, 3))) {
        fmt.Println(n) // 16 then 81
    }
}
```

Pipelines

Awesome example of how CSP concurrency easily composes into something parallel

At each stage we add waitgroups and fan-in fan-out methods.

```
// walkFiles starts a goroutine to walk the directory tree at root and send the
// path of each regular file on the string channel.  It sends the result of the
// walk on the error channel.  If done is closed, walkFiles abandons its work.
func walkDirs(done <-chan struct{}, root string) (<-chan string, <-chan error) {

func dirParser(done <-chan struct{}, dirs <-chan string, c chan<- result) {

func indexer(root string) error {
```

Pipelines

Results:

```
$ go run parser/superParser.go -in=../
2014/06/03 12:43:43 ../
2014/06/03 12:43:45 Indexed 4890 unique terms in 895 packages in 1.693358829s:
2014/06/03 12:43:45 Serializing index of size 4890 to file
2014/06/03 12:43:45 Wrote index file in 75.554495ms

$ go run parser/parser.go -in=../
2014/06/03 12:45:24 Indexer is quitting!
2014/06/03 12:45:24 Indexer exits cleanly
2014/06/03 12:45:24 Serializing index of size 4890 to file

Indexed 4890 unique terms in 999 packages in 3.294514543s
```

Take-aways

Despite not being specifically designed as a research language Go has many features with unique implementations and open
research questions:

- Concurrency
- Polymorphism
- Garbage Collection

Thank you

Patrick Baxter