Another Go at Language Design

Rob Pike
golang.org

April 28, 2010

http://golang.org

Who

Russ Cox
Robert Griesemer
Rob Pike
Ian Taylor
Ken Thompson

plus David Symonds, Nigel Tao, Andrew Gerrand, Stephen Ma, and others,

plus many contributions from the open source community.

Part 1

1. History
2. A niche
3. Tour of Go
4. Status

History

I'm always delighted by the light touch and stillness of early programming languages. Not much text; a lot gets done. Old programs read like quiet conversations between a well-spoken research worker and a well-studied mechanical colleague, not as a debate with a compiler. Who'd have guessed sophistication bought such noise?

-Dick Gabriel

Google

Sophistication

If more than one function is selected, any function template specializations in the set are eliminated if the set also contains a non-template function, and any given function template specialization F1 is eliminated if the set contains a second function template specialization whose function template is more specialized than the function template of F1 according to the partial ordering rules of 14.5.6.2. After such eliminations, if any, there shall remain exactly one selected function.
(C++0x, §13.4 [4])

Sophistication

Which Boost templated pointer type should I use?

- linked_ptr
- scoped_ptr
- shared_ptr
- smart_ptr
- weak_ptr
- intrusive_ptr
- exception_ptr

Noise

```
public static <I, O> ListenableFuture<O>
chain(ListenableFuture<I> input, Function<? super
I, ? extends ListenableFuture<? extends O>>
function) dear god make it stop
```

- a recently observed chat status

```
foo::Foo *myFoo = new foo::Foo(foo::FOO_INIT)
```

- but in the original `Foo` was a longer word

How did we get here?

A personal analysis:

1) C and Unix became dominant in research.

2) The desire for a higher-level language led to C++, which grafted the Simula style of object-oriented programming onto C. It was a poor fit but since it compiled to C it brought high-level programming to Unix.

3) C++ became the language of choice in parts of industry and in many research universities.

4) Java arose as a clearer, stripped-down C++.

5) By the late 1990s, a teaching language was needed that seemed relevant, and Java was chosen.

Google

Programming became too hard

These languages are hard to use.

They are subtle, intricate, and verbose.

Their standard model is oversold, and we respond with add-on models such as "patterns".

(Norvig: patterns are a demonstration of weakness in a language.)

Yet these languages are successful and vital.

A reaction

The inherent clumsiness of the main languages has caused a reaction.

A number of successful simpler languages (Python, Ruby, Lua, JavaScript, Erlang, ...) have become popular, in part as a rejection of the standard languages.

Some beautiful and rigorous languages designed by domain experts (Scala, Haskell, ...) have also arisen, although they are not as widely adopted.

So despite the standard model, other approaches are popular and there are signs of a growth in "outsider" languages, a renaissance of language invention.

Google

A confusion

The standard languages (Java, C++) are statically typed.

Most outsider languages (Ruby, Python, JavaScript) are interpreted and dynamically typed.

Perhaps as a result, non-expert programmers have confused "ease of use" with interpretation and dynamic typing.

This confusion arose because of how we got here: grafting an orthodoxy onto a language that couldn't support it cleanly.

Part 2

1. History
2. A niche
3. Tour of Go
4. Status

The good

The standard languages are very strong: type-safe, effective, efficient.

In the hands of experts, they are great.

Huge systems and huge companies are built on them.

In practice they work well for large scale programming: big programs, many programmers.

The bad

The standard languages are hard to use.

Compilers are slow and fussy. Binaries are huge.

Effective work needs language-aware tools, distributed compilation farms, ...

Many programmers prefer to avoid them.

The languages are at least 10 years old and poorly adapted to the current computing environment: clouds of networked multicore CPUs.

Flight to the suburbs

This is partly why Python et al. have become so popular:
They don't have much of the "bad".

- dynamically typed (fewer noisy keystrokes)
- interpreted (no compiler to wait for)
- good tools (interpreters make things easier)

But they also don't have the "good":

- slow
- not type-safe (static errors occur at runtime)
- very poor at scale

And they're also not very modern.

A niche

There is a niche to be filled: a language that has the good, avoids the bad, and is suitable to modern computing infrastructure:

- comprehensible
- statically typed
- light on the page
- fast to work in
- scales well
- doesn't require tools, but supports them well
- good at networking and multiprocessing

Part 3

1. History
2. A niche
3. Tour of Go
4. Status

The target

Go aims to combine the safety and performance of a statically typed compiled language with the expressiveness and convenience of a dynamically typed interpreted language.

It also aims to be suitable for modern systems programming.

How does Go fill the niche?

Fast compilation

Expressive type system

Concurrency

Garbage collection

Systems programming capabilities

Clarity and orthogonality

Google

Garbage collection

Automatic memory management simplifies life.

GC is critical for concurrent programming; otherwise it's too fussy and error-prone to track ownership as data moves around.

GC also clarifies design. A large part of the design of C and C++ libraries is about deciding who owns memory, who destroys resources.

But garbage collection isn't enough.

Memory safety

Memory in Go is intrinsically safer:

- pointers but no pointer arithmetic
- no dangling pointers (locals move to heap as needed)
- no pointer-to-integer conversions*
- all variables are zero-initialized
- all indexing is bounds-checked

Should have far fewer buffer overflow exploits.

* Package unsafe allows this but labels the code as dangerous; used mainly in some low-level libraries.

Control of bits and memory

Like C, Go has

- full set of unsigned types
- bit-level operations
- programmer control of memory layout

```
type T struct {
    x   int
    buf [20]byte
    ...
}
```

- pointers to inner values

```
p := &t.buf
```

Constants

Numeric constants are "ideal numbers": no size or signed/ unsigned distinction, hence no L or U or UL endings.

```
077 // octal
0xFEEDBEEEEEEEEEEEEEEEEEEEEF // hexadecimal
1 << 100
```

Syntax of literal determines default type:

```
1.234e5  // float
1e2      // float
100      // int
```

But they are just numbers that can be used at will and assigned to variables with no conversions necessary.

```
seconds := time.Nanoseconds()/1e9  // result has integer type
```

Systems language

By systems language, we mean suitable for writing systems software.

- web servers
- web browsers
- web crawlers
- search indexers
- databases
- compilers
- programming tools (debuggers, analyzers, ...)
- IDEs
- operating systems (maybe)

...

Systems programming

From http://loadcode.blogspot.com/2009/12/go-vs-java.html

"[Git] is known to be very fast. It is written in C. A Java version JGit was made. It was considerably slower. Handling of memory and lack of unsigned types [were] some of the important reasons."

Shawn O. Pearce wrote on the git mailing list:

"JGit struggles with not having an efficient way to represent a SHA-1. C can just say "unsigned char[20]" and have it inline into the container's memory allocation. A byte[20] in Java will cost an *additional* 16 bytes of memory, and be slower to access because the bytes themselves are in a different area of memory from the container object."

Hello, world 2.0

Serving http://localhost:8080/world:

```
package main
import (
   "fmt"
   "http"
)

func handler(c *http.Conn, r *http.Request) {
   fmt.Fprintf(c, "Hello, %s.", r.URL.Path[1:])
}

func main() {
   http.ListenAndServe(":8080",
                       http.HandlerFunc(handler))
}
```

Go is different

Go is object-oriented not type-oriented

- inheritance is not primary
- methods on any type, but no classes or subclasses

Go is (mostly) implicit not explicit

- types are inferred not declared
- objects have interfaces but they are derived, not specified

Go is concurrent not parallel

- intended for program structure, not max performance
- but still can keep all the cores humming nicely
- ... and many programs are more nicely expressed with concurrent ideas even if not parallel at all

Channels

Our trivial parallel program again:

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

Need to know when the computations are done.
Need to know the result.

A Go channel provides the capability: a typed synchronous communications mechanism.

Channels

Goroutines communicate using channels.

```
func computeAndSend(x, y, z int) chan int {
   ch := make(chan int)
   go func() {
      ch <- expensiveComputation(x, y, z)
   }()
   return ch
}

func main() {
   ch := computeAndSend(x, y, z)
   v2 := anotherExpensiveComputation(a, b, c)
   v1 := <-ch
   fmt.Println(v1, v2)
}
```

A worker pool

Traditional approach (C++, etc.) is to communicate by sharing memory:
- shared data structures protected by mutexes

Server would use shared memory to apportion work:

```
type Work struct {
   x, y, z int
   assigned, done bool
}

type WorkSet struct {
   mu sync.Mutex
   work []*Work
}
```

But not in Go.

Share memory by communicating

In Go, you reverse the equation.

- channels use the <- operator to synchronize and communicate

Typically don't need or want mutexes.

```
type Work struct { x, y, z int }
func worker(in <-chan *Work, out chan <- *Work) {
    for w := range in {
        w.z = w.x * w.y
        out <- w
    }
}
func main() {
    in, out := make(chan *Work), make(chan *Work)
    for i := 0; i < 10; i++ { go worker(in, out) }
    go sendLotsOfWork(in)
    receiveLotsOfResults(out)
}
```

Google

Part 4

1. History

2. A niche

3. Tour of Go

4. Status

Acceptance

Go was the 2009 TIOBE "Language of the year" two months after it was released.

Year	Winner
2009	Go
2008	C
2007	Python
2006	Ruby
2005	Java
2004	PHP
2003	C++

Testimonials

"I have reimplemented a networking project from Scala to Go. Scala code is 6000 lines. Go is about 3000.

Even though Go does not have the power of abbreviation, the flexible type system seems to out-run Scala when the programs start getting longer.

Hence, Go produces much shorter code asymptotically."

- Petar Maymounkov

"Go is unique because of the set of things it does well. It has areas for improvement, but for my needs it is the best match I've found when compared to: C, C++, Erlang, Python, Ruby, C#, D, Java, and Scala."

- Hans Stimer

Utility

For those on the team, it's the main day-to-day language now. It has rough spots but mostly in the libraries, which are improving fast.

Productivity seems much higher. (I get behind on mail much more often.) Most builds take a fraction of a second.

Starting to be used inside Google for some production work.

We haven't built truly large software in Go yet, but all indicators are positive.