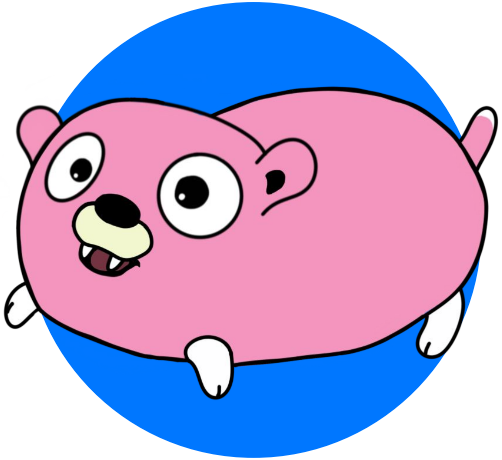
quasigo

A new Go interpreter

VK Tech Talk 2022

About me



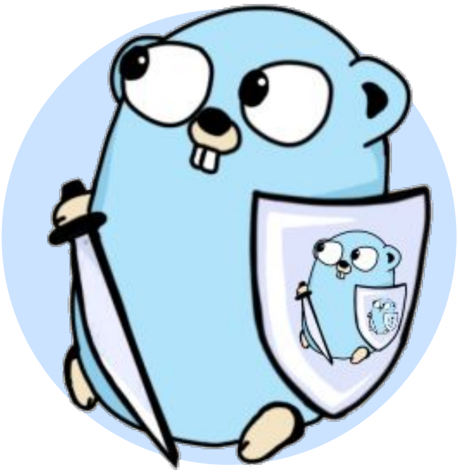
- Go compiler (Intel, Huawei)
- KPHP compiler (VK)
- Static analyzers (open source)
- Developer tools like phpgrep

Why does this talk exist?

I'll prove that it's possible to create efficient interpreter in Go that can compete with interpreters written in C.

Why bother?

I needed an efficient Go interpreter in my ruleguard project.

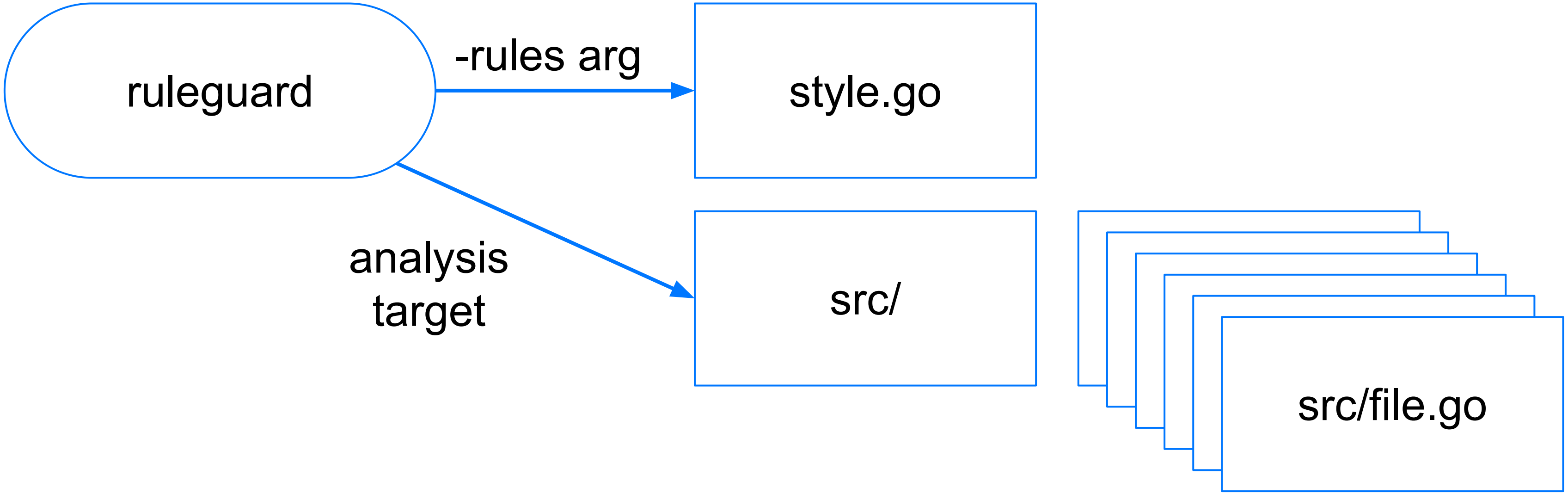


What is ruleguard?

An engine to execute dynamic rules for Go.

It loads the rules written in Go DSL and then executes them over a given project.

ruleguard overview




```
$ ruleguard -rules style.go src/...
```

ruleguard in action

before

```
-	rng, ok := n.(*ast.RangeStmt)
-	if !ok {
-		return nil
-	}
-	if rng.Value == nil || astcast.ToIdent(rng.Key).Name != "_" {
-		return nil
-	}
-	if len(rng.Body.List) != 1 {
-		return nil
-	}
-	rangeExprType := ctx.TypeOf(rng.X)
-	if !typeis.Slice(rangeExprType) {
-		return nil
-	}
-	assign, ok := rng.Body.List[0].(*ast.AssignStmt)
-	if !ok || assign.Tok != token.ASSIGN || len(assign.Lhs) != 1 || len(assign.Rhs) != 1 {
-		return nil
-	}
-	lhs := assign.Lhs[0]
-	rhs := assign.Rhs[0]
-	lhsType := ctx.TypeOf(lhs)
-	if !types.Identical(rangeExprType, lhsType) {
-		return nil
-	}
-	call, ok := rhs.(*ast.CallExpr)
-	if !ok {
-		return nil
-	}
-	called, ok := call.Fun.(*ast.Ident)
-	if !ok || called.Name != "append" || len(call.Args) != 2 {
```

after (ruleguard DSL)

```
//doc:summary Detects range loops that can be turned into a single append call
//doc:tags    o1 score3
func rangeToAppend(m dsl.Matcher) {
	m.Match(`for _, $x := range $src { $dst = append($dst, $x) }`).
		Where(m["src"].Type.Is(`[]$_`) && !m["dst"].Contains(`$x`) && m["src"]
		Suggest(`$dst = append($dst, $src...)`).
		Report(`for … { … } => $dst = append($dst, $src...)`)
}
```

Why does ruleguard need an interpreter?

Matcher pipeline

```
m.Match(`string($x)`).
  Where(
    m["x"].Filter(f),
  ).
  Report("message")
```

Matcher pipeline

```
m.Match(`string($x)`).
  Where(
    m["x"].Filter(f),
  ).
  Report("message")
```

1. Find an AST (syntax) match

Matcher pipeline

```
m.Match(`string($x)`).
  Where(
    m["x"].Filter(f),
  ).
  Report("message")
```

1. Find an AST (syntax) match

(if AST matched)

2. Apply filters to the match

Matcher pipeline

```
m.Match(`string($x)`).
  Where(
    m["x"].Filter(f),
  ).
  Report("message")
```

1. Find an AST (syntax) match

(if AST matched)

2. Apply filters to the match

(if filters accepted the match)

3. Perform an action

Matcher pipeline

```
m.Match(`string($x)`).
  Where(
    m["x"].Filter(f),
  ).
  Report("message")
```

f

could be a custom function

A custom filter example

```go
func implementsStringer(ctx *dsl.VarFilterContext) bool {
  stringer := ctx.GetInterface(`fmt.Stringer`)

  // pointer to the captured, type, T -> *T
  ptr := types.NewPointer(ctx.Type)

  return types.Implements(ctx.Type, stringer) ||
         types.Implements(ptr, stringer)
}
```

A custom filter example

```
func implementsStringer(ctx *dsl.VarFilterContext) bool {
  stringer := ctx.GetInterface(`fmt.Stringer`)

  // pointer to the captured, type, T -> *T
  ptr := types.NewPointer(ctx.Type)

  return types.Implements(ctx.Type, stringer) ||
         types.Implements(ptr, stringer)
}
```

How to execute?

Ruleguard resources

Ruleguard by example

Introduction article (RU, EN)

Ruleguard workshop videos (RU)

Ruleguard vs SemGrep vs CodeQL (EN)

Trying out existing interpreters

Ruleguard use case

- A lot of calls to small functions
  Go -> interpreter

- Filters call native bindings
  Interpreter -> Go calls

Need performance in both directions

Yaegi

Popular

Reliable

User-friendly, easy API

github.com/traefik/yaegi

My experience with yaegi

I built [Gophers & Dragons] game using yaegi.

Gophers & Dragons   Run   Normal speed   Resume   Next Turn   Format   Share   Tactics editor   Open documentation

```go
package tactic

import "github.com/quasilyte/gophers-and-dragons/game"

func ChooseCard(s game.State) game.CardType {
  return tactic2(s)
}

// tactic1 is a trivial tactic that always retreats.
// You can't win like this, but it's a minimal working
// example that manages to pass the entire game without dying.
func tactic1(s game.State) game.CardType {
  return game.CardRetreat
}

// tactic2 will only fight with the easiest kind of
// monsters and will run away if wounded.
func tactic2(s game.State) game.CardType {
  if s.Avatar.HP < 10 {
    return game.CardRetreat
  }
  if s.Creep.Type == game.CreepCheepy {
    return game.CardAttack
  }
  return game.CardRetreat
}
```

--- Turn 1 ---
Your Attack deals 2 damage
Cheepy deals 1 damage
--- Turn 2 ---
Your Attack deals 2 damage
Cheepy is defeated! 3 score points received
Collected Firebolt card
--- Turn 3 ---
Trying to retreat...
Imp deals 3 damage
Retreated from Imp!
--- Turn 4 ---
Your Attack deals 4 damage
Cheepy is defeated! 3 score points received
Collected Stun card

Avatar  HP: 36 MP: 20

Creep  Lion HP: 10

Next creep  Lion HP: 10

Offensive cards	Tactical cards
Attack (∞) 0 MP	Retreat (∞) 0 MP
MagicArrow (∞) 1 MP	Rest (∞) 2 MP
PowerAttack (0) 0 MP	Heal (0) 4 MP
Firebolt (1) 3 MP	Parry (0) 0 MP
Stun (1) 0 MP	

Retreat card info

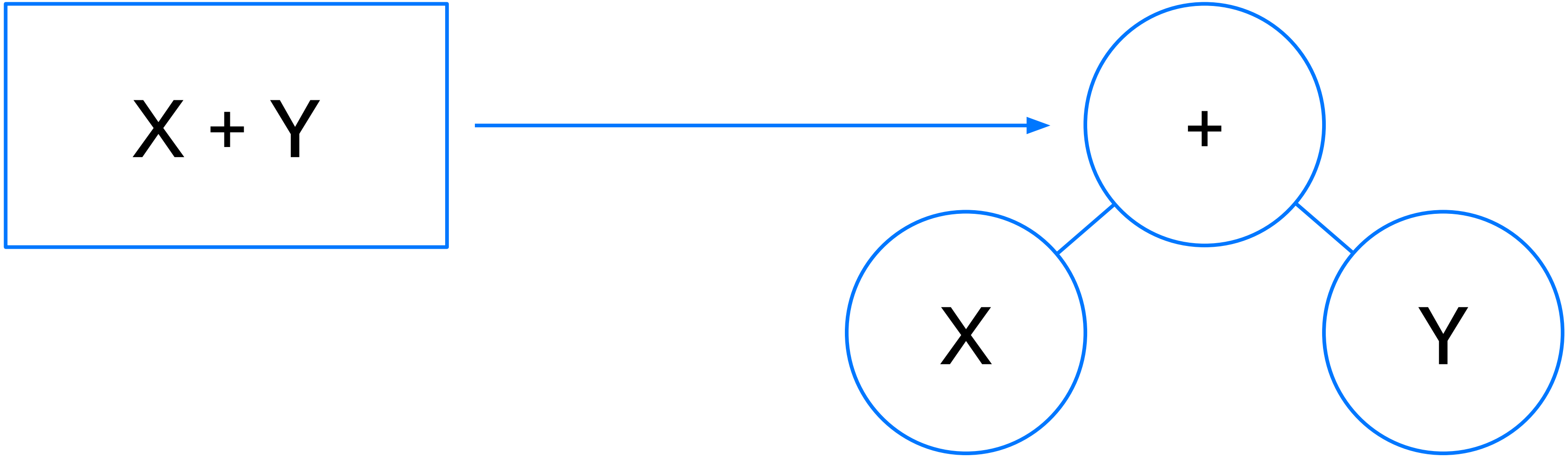
Avoid the current encounter.

MP cost: 0

Score: 6   Turn: 4   Round: 4/10

Yaegi interpretation model

Yaegi builds an annotated AST and interprets it directly



Yaegi performance issues

reflection.Value as value type

Tons of heap allocations

Direct AST interpretation

Interpreters comparison

Interpreter	Eval performance	Eval entry overhead
Yaegi	Very low	High

To Yaegi
or not to Yaegi?

Scriggo

Scriggo

Fast

Part of the template engine

Younger than yaegi

github.com/open2b/scriggo

Scriggo interpretation model

Scriggo creates bytecode and then evaluates that

Scriggo multi stacks

```
type Registers struct {
  Int     []int64
  Float   []float64
  String  []string
  General []reflect.Value
}
```

Scriggo multi stacks

```
type Registers struct {
  Int     []int64   // efficient!
  Float   []float64 // efficient!
  String  []string  // efficient!
  General []reflect.Value
}
```

Scriggo multi stacks

```
type Registers struct {
    Int     []int64 // also handles int8/16/32…
    Float   []float64
    String  []string
    General []reflect.Value
}
```

Scriggo multi stacks

```
type Registers struct {
   Int      []int64
   Float    []float64
   String   []string
   General []reflect.Value // slow
}
```

Scriggo performance issues

Some types have bad performance (e.g. [ ]byte)

Expensive Go->interpreter call costs

Interpreters comparison

Interpreter	Eval performance	Eval entry overhead
Yaegi	Very low	High
Scriggo	High	Very High

Quasigo interpreter

quasigo principles

Subset only

quasigo principles

Subset only

Performance matters

quasigo principles

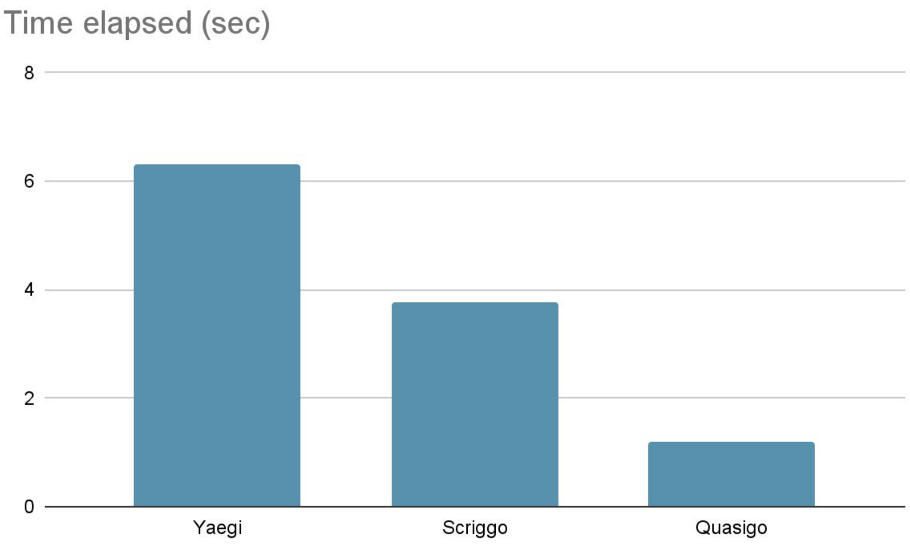
`Subset only`

`Performance matters`

`Toolchain as a library`

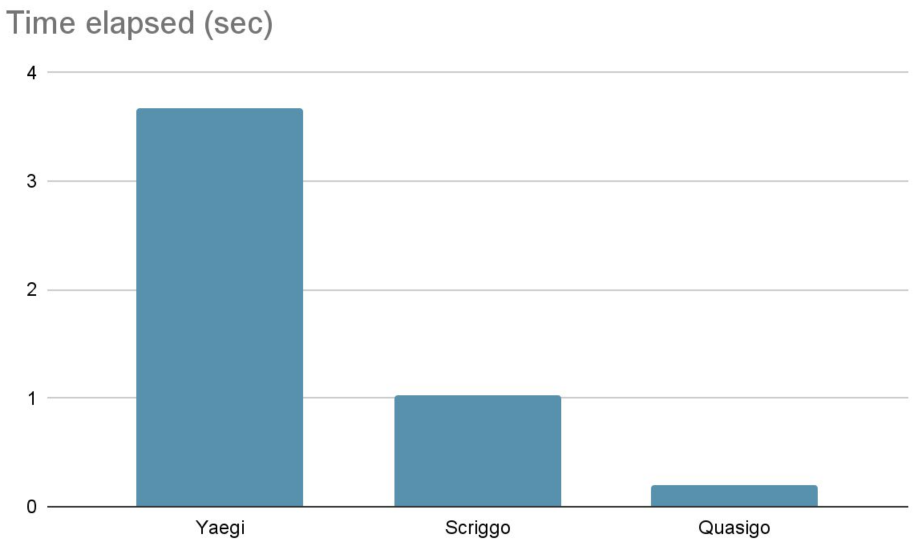
Sqrt benchmark

Interpreter	Elapsed
Yaegi	6.32s (x4.2)
Scriggo	3.78s (x2.1)
Quasigo	1.21s



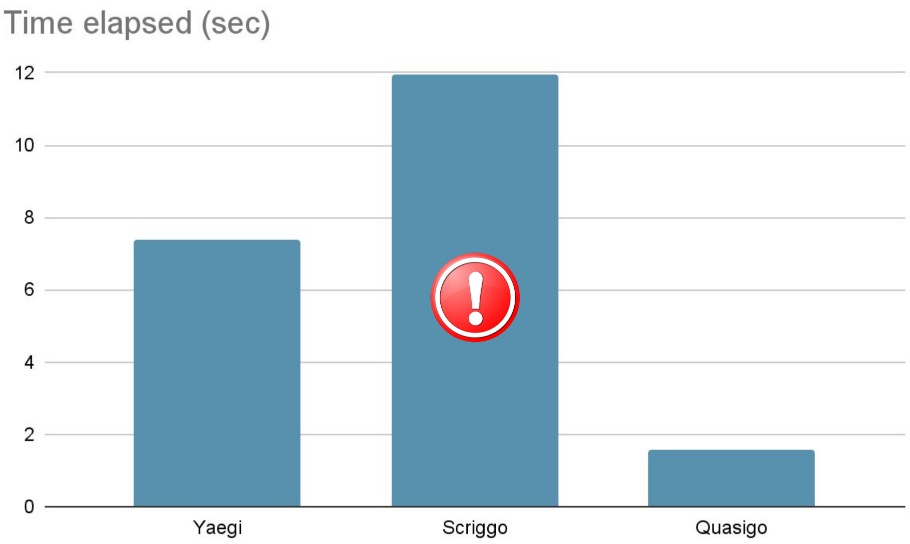
Spectral norm benchmark

Interpreter	Elapsed
Yaegi	3.67s (x17.3)
Scriggo	1.03s (x4.1)
Quasigo	0.20s



Mandelbrot benchmark

Interpreter	Elapsed
Yaegi	7.37s (x3.7)
Scriggo	11.93s (x6.6)
Quasigo	1.57s

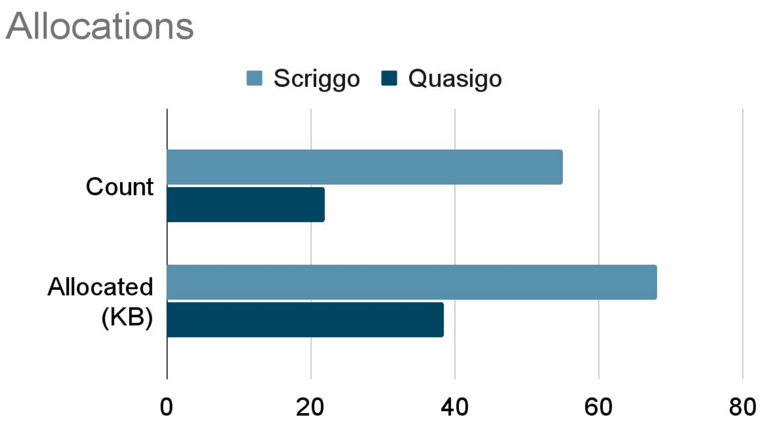


Sqrt benchmark (allocs)

Interpreter	Count	Bytes
Yaegi	62985026	4013885208
Scriggo	8	29408
Quasigo	**0**	**0**

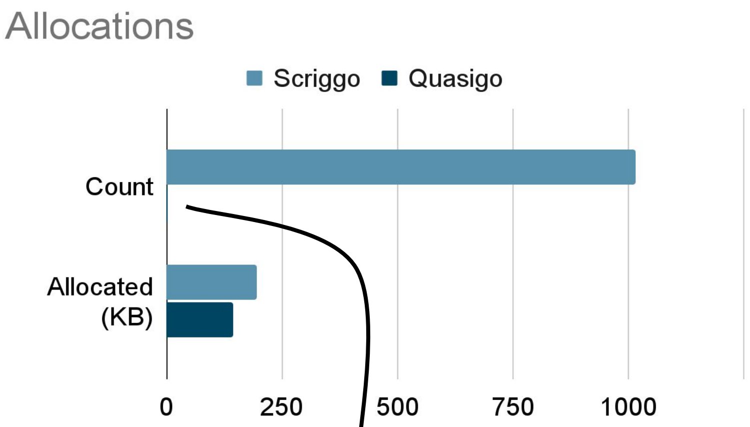
Spectral norm benchmark (allocs)

Interpreter	Count	Bytes
Yaegi	19209002	793746584
Scriggo	55	69824
Quasigo	22	39416



Mandelbrot benchmark (allocs)

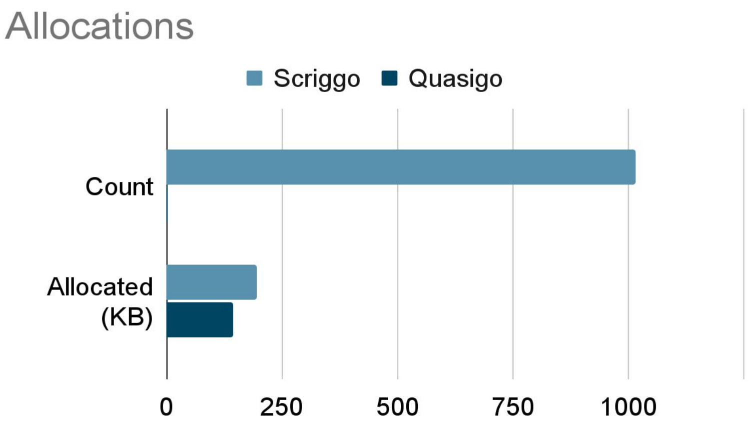
Interpreter	Count	Bytes
Yaegi	1189064	11006432
Scriggo	1016	201016
Quasigo	3	147416



3 allocs 🔥

Mandelbrot benchmark (allocs)

Interpreter	Count	Bytes
Yaegi	1189064	11006432
Scriggo	1016	201016
Quasigo	3	147416



What is wrong with Scriggo here?

Why does Scriggo allocate a lot in Mandelbrot?

```go
rowOffset := 0
for i := 0; i < numRows; i++ {
        rowBytes := rowsData[rowOffset : rowOffset+bytesPerRow]
        renderRow(initialR, initialI, rowBytes, i)
        rowOffset += bytesPerRow
}
return rowsData
```

Mandelbrot size for benchmark is 1000, so numRows=1000

Why does Scriggo allocate a lot in Mandelbrot?

```go
rowOffset := 0
for i := 0; i < numRows; i++ {
        rowBytes := rowsData[rowOffset : rowOffset+bytesPerRow]
        renderRow(initialR, initialI, rowBytes, i)
        rowOffset += bytesPerRow
}
return rowsData
```

Scriggo stores [ ]byte in reflect.Value
Slicing creates a new 24-byte allocation

Why quasigo is faster than Scriggo?

Quasigo doesn′t shy away from unsafe package:

1. Faster interpreter core (instructions dispatch)
2. Almost free native calls
3. Efficient frames layout and slots representation
4. Reflection-free access to arbitrary data

Is “unsafe” package usage justified here?

Go runtime itself is written with a help of “unsafe”.

It’s OK to use unsafe package in runtimes and low-level libraries that need all performance they can get.

Interpreters comparison

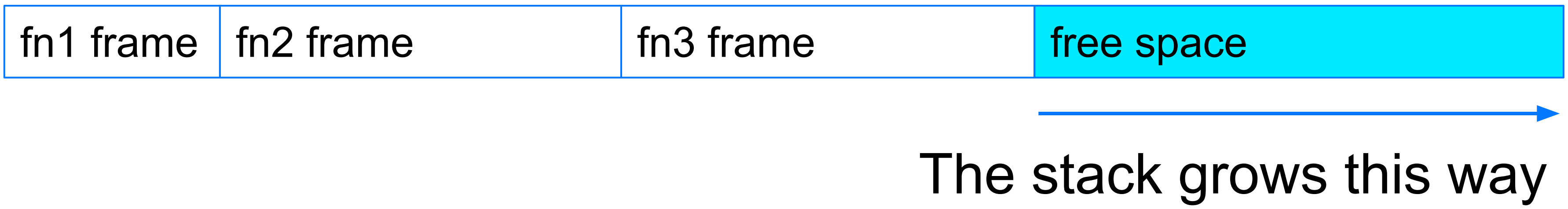
Interpreter	Eval performance	Eval entry overhead
Yaegi	Very low	High
Scriggo	High	Very High
Quasigo	Very high	Very low

Interpreters comparison (more)

Interpreter	Interpretation type	Relies on
Yaegi	AST traversal	reflection
Scriggo	Bytecode, reg VM	reflection
Quasigo	Bytecode, reg VM	unsafe

Quasigo runtime

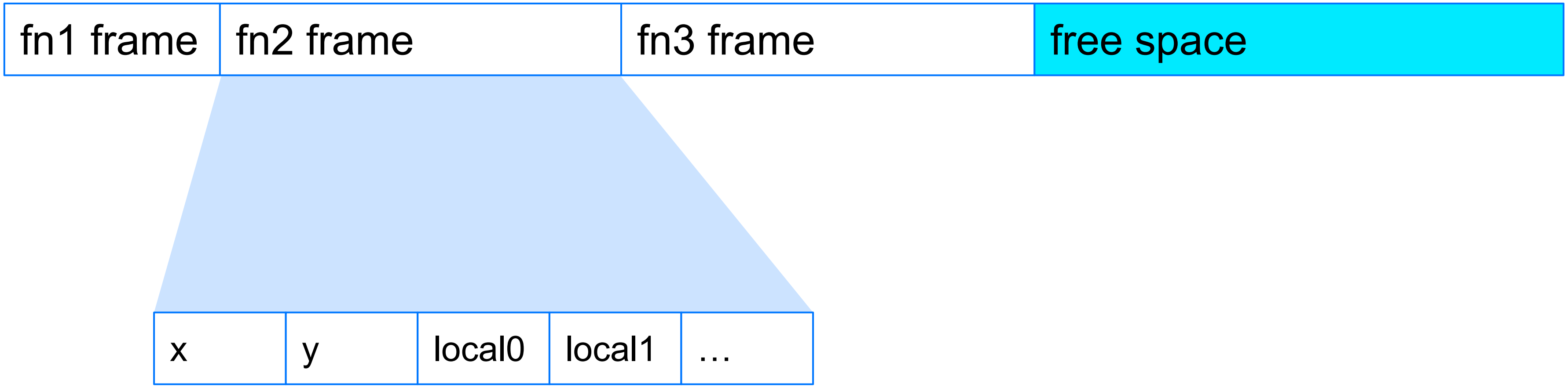
VM stack frame



VM stack frame

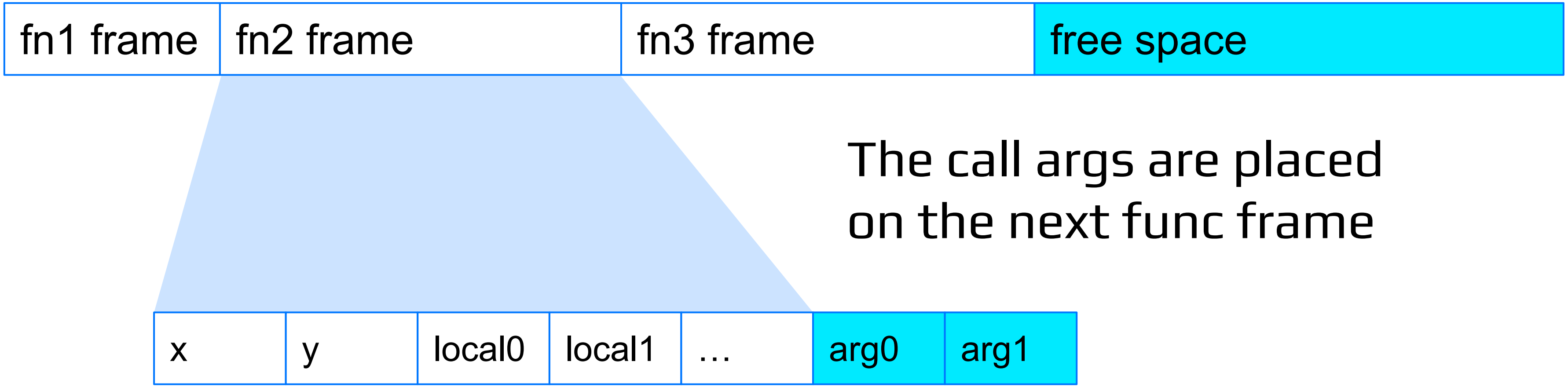
Interpreter memory

VM stack frame




```
func fn2(x, y int) int
```

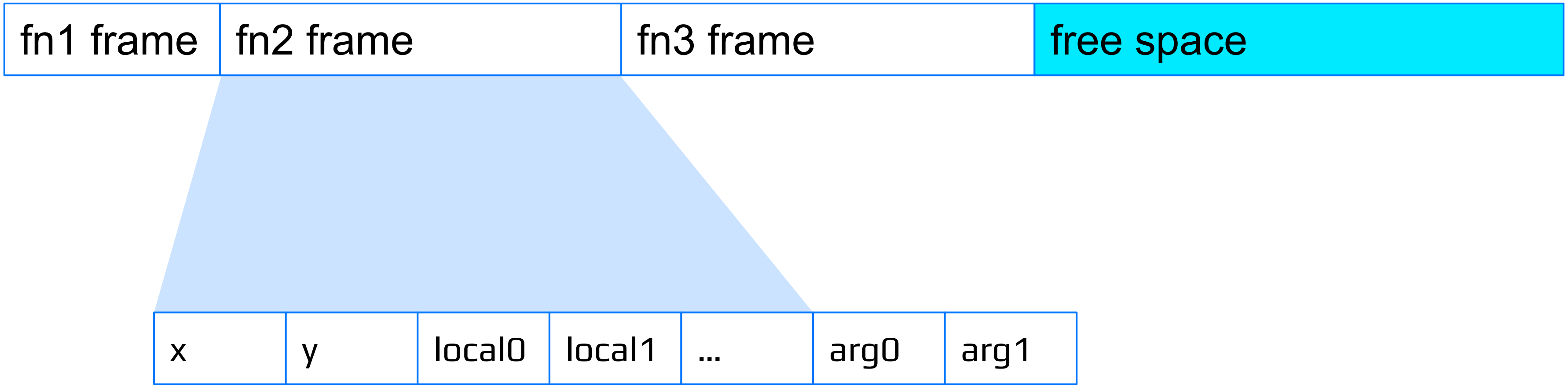
VM stack frame



The call args are placed on the next func frame

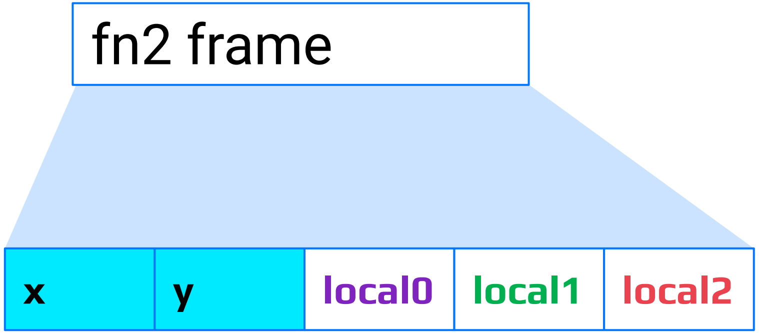
```
func fn2(x, y int) int
```

VM stack frame



These cells are called “slots” (or “virtual registers”)

VM stack frame




```
func fn2(x, y int) int {
  return x + y * 3
}
```

```
LoadScalarConst local2 = 3
IntMul64 local1 = y local2
IntAdd64 local0 = x local1
ReturnScalar local0
```

VM stack slots

```
type Slot struct {
  Ptr     unsafe.Pointer
  Scalar  uint64
  Scalar2 uint64
}
```

VM stack slots

```
type Slot struct {
  Ptr     unsafe.Pointer
  Scalar  uint64
  Scalar2 uint64
}
```

sizeof(slot) == 24 bytes
(on 64-bit platforms)

VM stack slots: pointer types

```
type Slot struct {
  Ptr      unsafe.Pointer
  Scalar   uint64
  Scalar2  uint64
}
```

Pointer types are stored in Pointer field

VM stack slots: scalar types

```go
type Slot struct {
  Ptr     unsafe.Pointer
  Scalar  uint64
  Scalar2 uint64
}
```

Simple numeric types are stored in Scalar field

VM stack slots: strings

```go
type Slot struct {
    Ptr     unsafe.Pointer
    Scalar  uint64
    Scalar2 uint64
}
```

Strings are stored in Ptr+Scalar.

This matches the Go runtime string layout!

VM stack slots: slices

```go
type Slot struct {
    Ptr     unsafe.Pointer
    Scalar  uint64
    Scalar2 uint64
}
```

Slices occupy all the slots.

This matches the Go runtime slices layout!

VM stack slots: interfaces

```
type Slot struct {
  Ptr     unsafe.Pointer
  Scalar  uint64
  Scalar2 uint64
}
```

Interfaces are stored in Ptr+Scalar.

Data and typeinfo pointers are swapped.

VM stack slots: structs

```go
type Slot struct {
  Ptr     unsafe.Pointer
  Scalar  uint64
  Scalar2 uint64
}
```

Small structs are stored directly inside the slot, if possible.

Otherwise they're heap allocated and we store a pointer to it.

Quasigo compiler

Compiler architecture

Go code	The input Go sources
AST+types	go/ast and go/types data
IR	Low-level intermediate representation
bytecode	The final compiler output

Compiler architecture

Go code

AST+types

IR — Good for optimizations and transformations

bytecode — Good for the execution (it’s also compact)

Bytecode instructions encoding

Simple variadic-length scheme:

- 1-byte opcode
- 1 or 2 bytes per instruction argument
- Constants are loaded from external slice using the index

3-address instruction format: dst + src1 + src2

Bytecode instructions encoding

```
dst = x + y            =>        IntAdd64 dst = x y
```

- Opcode=IntAdd64 (1 byte)
- Arg0 dst (1 byte)
- Arg1 x (1 byte)
- Arg2 y (1 byte)

Frame slot index

Constant propagation

```
LoadScalarConst local0.v0 = 1
LoadScalarConst local2.v0 = 1
IntAdd64 local1.v0 = local0.v0 local2.v0
LoadScalarConst local3.v0 = 1
IntAdd64 local2.v1 = local1.v0 local3.v0
LoadScalarConst local4.v0 = 1
IntAdd64 local3.v1 = local2.v1 local4.v0
ReturnScalar local3.v1


# function IR
```

```
func f() int {
  x1 := 1
  x2 := x1 + 1
  x3 := x2 + 1
  return x3 + 1
}
```

Constant propagation

```
LoadScalarConst local0.v0 = 1
LoadScalarConst local2.v0 = 1
IntAdd64 local1.v0 = local0.v0 local2.v0
LoadScalarConst local3.v0 = 1
IntAdd64 local2.v1 = local1.v0 local3.v0
LoadScalarConst local4.v0 = 1
IntAdd64 local3.v1 = local2.v1 local4.v0
ReturnScalar local3.v1


# Slots have unique "versions"
```

```
func f() int {
  x1 := 1
  x2 := x1 + 1
  x3 := x2 + 1
  return x3 + 1
}
```

Constant propagation

```
LoadScalarConst local0.v0 = 1
LoadScalarConst local2.v0 = 1
IntAdd64 local1.v0 = local0.v0 local2.v0
LoadScalarConst local3.v0 = 1
IntAdd64 local2.v1 = local1.v0 local3.v0
LoadScalarConst local4.v0 = 1
IntAdd64 local3.v1 = local2.v1 local4.v0
ReturnScalar local3.v1


# Same slots can have different
# versions in a single block
```

```
func f() int {
  x1 := 1
  x2 := x1 + 1
  x3 := x2 + 1
  return x3 + 1
}
```

Constant propagation

```
# Result bytecode

LoadScalarConst local0 = 4
ReturnScalar local0
```

```
func f() int {
  x1 := 1
  x2 := x1 + 1
  x3 := x2 + 1
  return x3 + 1
}
```

Jump condition optimizations

```
Len local2.v0 = s
Zero local3.v0
ScalarEq local1.v0 = local2.v0 local3.v0
Not local0.v0 = local1.v0
JumpZero Label0 local0.v0
```

```
if !(len(s) == 0) {
  …
}
```

Jump condition optimizations

```
Len local2.v0 = s
Zero local3.v0
ScalarEq local1.v0 = local2.v0 local3.v0
Not local0.v0 = local1.v0
JumpZero Label0 local0.v0


# Can inverse the jump cond
```

```
if !(len(s) == 0) {
  …
}
```

Jump condition optimizations

```
Len local2.v0 = s
Zero local3.v0
ScalarEq local1.v0 = local2.v0 local3.v0
Not local0.v0 = local1.v0
JumpNotZero Label0 local1.v0
```

```
if !(len(s) == 0) {
  …
}
```

Jump condition optimizations

```
Len local2.v0 = s
Zero local3.v0
ScalarEq local1.v0 = local2.v0 local3.v0
JumpNotZero Label0 local1.v0


# Can inject the zero comparison
# into the jump cond
```

```
if !(len(s) == 0) {
  …
}
```

Jump condition optimizations

```
Len local2.v0 = s
Zero local3.v0
ScalarEq local1.v0 = local2.v0 local3.v0
JumpZero Label0 local2.v0
```

```
if !(len(s) == 0) {
  …
}
```

Jump condition optimizations

```
# Result bytecode

Len local2 = s
JumpZero Label0 local2
```

```
if !(len(s) == 0) {
  …
}
```

Other optimizations

Implemented:

- Inlining (with post-inlining optimizations)
- Some idioms and corner cases recognition
- Frames trimming
- Unused constants removal

Other optimizations

Planned:

- Comparisons fusing and rewrites
- Jump threading
- More peephole optimizations

Where can I use quasigo?

Quasigo for game development

Write a game in Go (using ebiten or other game library).

Allow the users to write plugins/scripts for your game in Go, using quasigo as embedded interpreter.

Ebiten

Quasigo for query languages

For example, a DB like tarantool, but written in Go and with Go as a query language instead of Lua.

It’s also possible to use Go scripts as custom filtering lambdas for your internal services.

Quasigo for template engines

Since Scriggo is used as a template engine core, I could imagine quasigo used in the same context.

It will probably be at least as efficient as Scriggo in that domain.

Quasigo state

Good enough for ruleguard

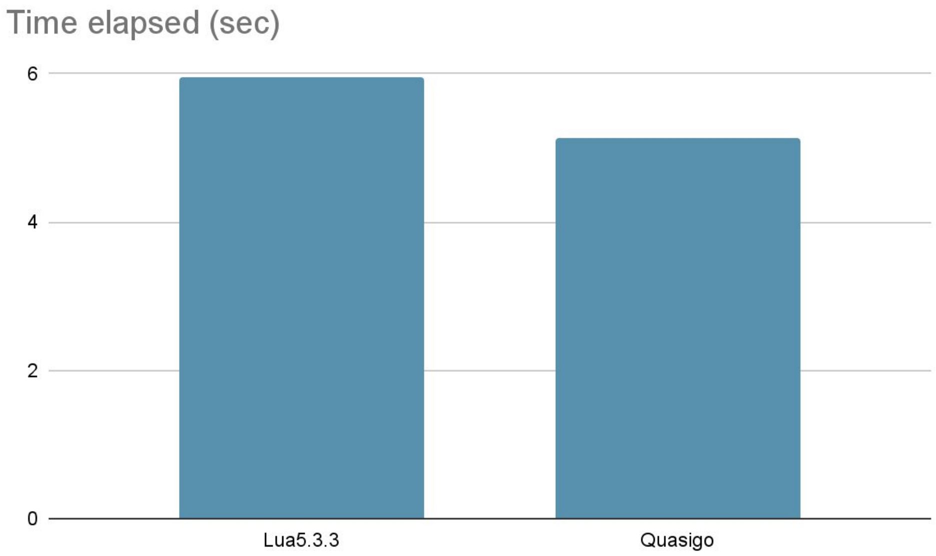
Not ready for general purpose production use yet

Alpha release can be expected in 4-6 months

Comparing with Lua

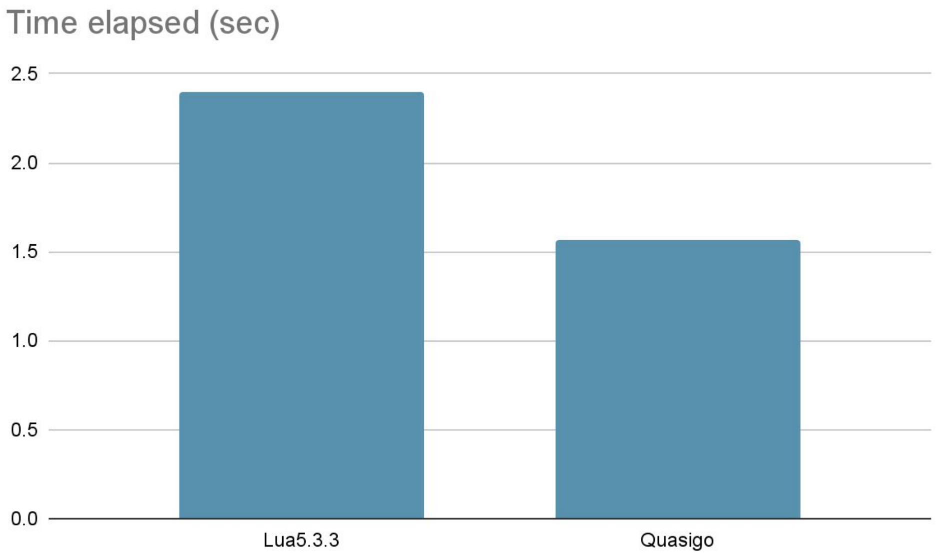
Running spectral norm (n=1000)

Interpreter	Elapsed
Lua5.3.3	5.95s
Quasigo	5.13s (18% faster)



Running mandelbrot (size=1000)

Interpreter	Elapsed
Lua5.3.3	2.40s
Quasigo	1.57s (34% faster)



Lua vs quasigo

Interpreter	Implemented in	Target language
Lua	C	Lua
Quasigo	Go	Go

Creating interpreter in Go

Cons:

- Higher bytecode instruction dispatch cost
- Harder to fine-tune runtime-related code without asm
- Paying extra price to be Go GC friendly

Overall, the raw performance can be ~20% slower for identically optimal interpreters of the same target language.

Creating interpreter in Go

Pros:

- No need to use CGo to embed the interpreter
- Getting a GC for free
- Cheap interop with Go (in both directions)
- Can use Go stdlib in the target language stdlib
- Great benchmarking/testing/ profiling support

But why is quasigo sometimes faster?

- Statically typed values (therefore, instructions)
- Go has true integer type (and unboxed scalars in general)
- For array-like data, slices are better than Lua tables
- Structs are better than Lua tables

The raw performance is lower,
but Go is “faster” than Lua.

Conclusions

1. Interpreters benefit from “unsafe” a lot

Conclusions

1. Interpreters benefit from “unsafe” a lot
2. Interpreters written in Go can be quite fast if done right

Conclusions

1. Interpreters benefit from “unsafe” a lot
2. Interpreters written in Go can be quite fast if done right
3. Go is a good interpretation target language

Conclusions

1. Interpreters benefit from "unsafe" a lot
2. Interpreters written in Go can be quite fast if done right
3. Go is a good interpretation target language
4. You can embed Go instead of Lua in your Go apps

Related resources

[SSA form alternative]

[Efficient VM with JIT in Go]

quasigo

A new Go interpreter

VK Tech Talk 2022