GoLightly

A Go library for building Virtual Machines

Eleanor McHugh

Go

a new language

* statically-typed and compiled
* object-oriented but no classes
* garbage collected
* concurrency via communication (CSP)
* type polymorphism via interfaces

a new way of working

* the safety of a static type system
* the feel of a dynamic runtime
* the performance of a compiled language

Virtual Machines

from inspiration...

* computation
* serialisation
* communication

...to perspiration

* simulation v. emulation
* instruction set design
* bytecodes
* memory abstraction
* operation dispatch

GoLightly

principles

* interface driven delegation
* architecture agnostic
* inspired by hardware techniques
* decoupled components
* scaling through specialisation

caveat

* references current dev branch
* fundamental changes from github
* still evolving
* next release due for Strange Loop

interfaces

```
type Executable interface {
    Execute(p Processor)
}

type Processor interface {
    Reset()
    Step()
    Jump(a Address)
    Call(a Address)
    Return()
    Load(program Program)
    Run()
    Execute()
    Sleep(n int64)
    Halt(n int)
    IsActive() bool
}
```

```
type Comparable interface {
    Identical(v Value) bool
    Compare(v Value) int
}

type Value interface {
    fmt.Stringer
    Comparable
    Nil() Value
    IsNil() bool
}

type Program []Executable
```

instructions

* obey the Executable interface
* expressed as separate types
* no assumptions about semantics
* not tied to a bytecode representation

flow control

```
type NoOp struct {}
func (n *NoOp) String() string {
    return "NOOP"
}
func (n *NoOp) Execute(p Processor) {}

type Halt struct {}
func (h *Halt) String() string {
    return "HALT"
}
func (h *Halt) Execute(p Processor) {
    p.Halt(PROGRAM_TERMINATED)
}

type Jump Address
func (j Jump) String() string {
    return Sprint("JMP", Address(j))
}
func (j Jump) Execute(p Processor) {
    p.Jump(Address(j))
}
```

```
type Call Address
func (c Call) String() string {
    return Sprint("CALL", Address(c))
}
func (c Call) Execute(p Processor) {
    p.Call(Address(c))
}

type Return struct {}
func (r Return) String() string {
    return "RET"
}
func (r Return) Execute(p Processor) {
    p.Return()
}
```

bytecode

```
type ByteCode []int
func (b *ByteCode) String() string {
        return "BYTECODE"
}
func (b *ByteCode) Execute(p Processor) {
        var PC, C, W int
        var R [2]int
        CS := new(vector.IntVector)
        DS := new(vector.IntVector)
        for p.Active() {
                switch b[PC] {
                case 0:         // NOOP
                case 1:         // NSLEEP n
                        PC++
                        p.Sleep(int64(b[PC]))
                case 2:         // SLEEP n
                        PC++
                        p.Sleep(int64(b[PC]) << 32)
                case 3:         // HALT
                        p.Halt(USER_HALT)
                        return
                case 4:         // JMP n
                        PC++
                        PC += b[PC] - 1
                case 5:         // JMPZ n
                        PC++
                        if C == 0 {
                                PC++
                                PC += b[PC] - 1
                        } else {
                                PC++
                        }
                case 6:         // JMPNZ n
                        PC++
                        if C != 0 {
                                PC++
                                PC += b[PC] - 1
                        } else {
                                PC++
                        }
                case 7:         // CALL n
                        PC++
                        CS.Push(PC)
                        PC = b[PC]
                case 8: // RET
                        PC = CS.Pop
                case 9: // PUSH r
                        PC++
                        DS.Push(R[b[PC]])
                case 10:        // POP r
                        PC++
                        R[b[PC]] = DS.Pop()
                case 11:        // LOAD r, v
                        PC++
                        W = b[PC]
                        PC++
                        R[W] = b[PC]
                case 12:        // ADD r1, r2
                        PC++
                        W = b[PC]
                        PC++
                        R[b[PC]] += R[W]
                default:
                        p.Halt(ILLEGAL_INSTRUCTION)
                }
        }
}
```

processors

* typical Turing machines
* support flexible dispatch
* not tied to a given instruction set
* memory model agnostic
* ripe for specialisation

a scalar processor

```
type SProc struct {
	Running   bool
	PC int
	R   IntBuffer
	F   FloatBuffer
	DS vector.IntVector
	Program   []Executable
}

func (s *SProc) Run() {
	s.Running = true
	s.Call(0)
}

func (s *SProc) Step() {
	s.PC++
}

func (s *SProc) Jump(a Address) {
	s.PC += int(a)
}

func (s *SProc) Execute() {
	s.Program[s.PC].Execute(s)
}

func (s *SProc) Sleep(i int64) {
	syscall.Sleep(i)
}

func (s *SProc) Halt(n int) {
	s.Running = false
}

func (s *SProc) IsActive() bool{
	return s.Running && s.PC <
	len(s.Program)
}

func (s *SProc) Reset() {
	s.R.ClearAll()
	s.PC = 0
}

func (s *SProc) Load(program
Program) {
	s.Reset()
	s.Program = program
}

func (s *SProc) Call(a Address){
	PC := s.PC
	s.PC = int(a)
	for s.IsActive() {
		s.Execute()
		s.Step()
	}
	s.PC = PC
}

func (s *SProc) Return() {
	s.PC = len(s.Program)
}
```

memory

* based on allocated Values
* aggregate as ValueStores
* require boxing/unboxing
* buffers for specific types
* buffers are also ValueStores

benefits

* encourages simple designs
* many different VMs per executable
* can run in separate goroutines
* easily duplicated and serialised
* split across processes or hosts

the future

* single dispatch for vector operations
* pervasive threading
* runtime code rewriting
* speculative loading
* JIT compilation

find out more

* http://github.com/feyeleanor/GoLightly
* http://golightly.wikidot.com
* twitter://#golightly