Jae Kwon

Gno: Lessons in Building a Go Interpreter in Go

Background

gno.land

Tendermint 2014

- first non-PoW BFT consensus algo & impl

Cosmos 2017,

- first complete solution to “proof of stake”
- first blockchain-to-blockchain communication protocol IBC
- CosmosSDK, most popular Go framework for blockchain dev

What is Gno

Gno is a <u>deterministic</u> Go interpreter*
<u>transactional</u>
<u>persistent</u>
<u>magical</u>

* goroutines and generics planned

Why Gno?

The Gno VM enables seamless interoperability of untrusted user programs written in the Go language.

gno.land

Seamless Interoperability

```go
package alice

var x int

func GetX() int {
	return x
}

func SetX(n int) {
	x = n
}
```

```go
package bob

import "alice"

func IncrAlice() {
	x := alice.GetX()
	alice.SetX(x+1)
}
```

gno.land

Seamless Interoperability

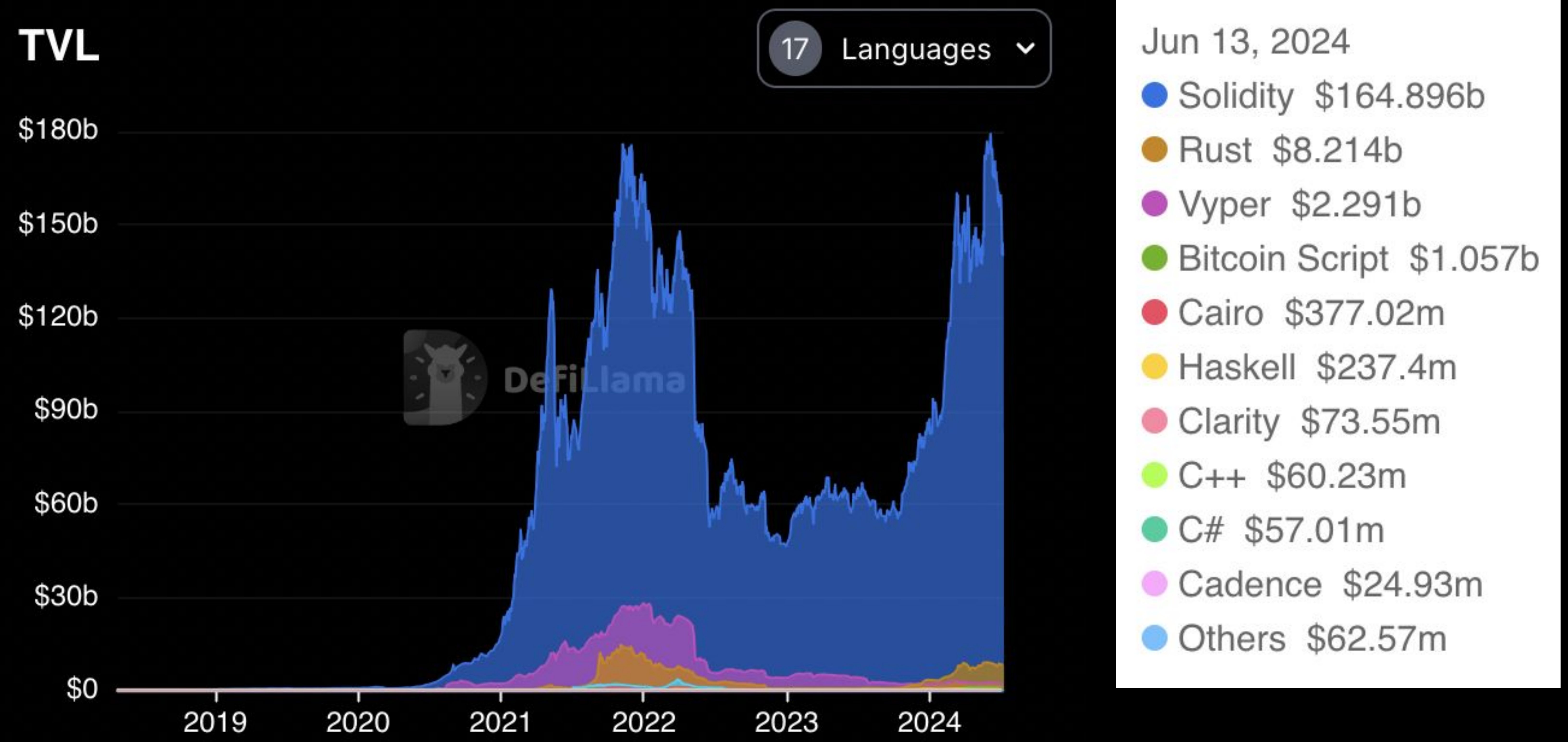
```go
package carl

import "bob"

func RelayToBob(msg string) {
	bob.SendMessage(msg, func() {
		println("success!")
	})
}
```

State of Smart Contract Langs

gno.land



EVM/Solidity?

gno.land

* EVM limitations
* Solidity limitations
* No garbage collector
* Max 16 variables per function
* No closures
* No self-referential structs
* ...

gno.land

Near/WASM/Rust?

CROSS-CONTRACT CALLS ARE INDEPENDENT

You will need two independent functions: one to make the call, and another to receive the result

CROSS-CONTRACT CALLS ARE ASYNCHRONOUS

There is a delay between the call and the callback execution, usually of **1 or 2 blocks**. During this time, the contract is still active and can receive other calls.

Near/WASM/Rust?

gno.land

```
#[ext_contract(external_trait)]
trait Contract {
    fn function_name(&self, param1: T, param2: T) -> T;
}

external_trait::ext("external_address")
.with_attached_deposit(DEPOSIT)
.with_static_gas(GAS)
.function_name(arguments)
.then(
  // this is the callback
  Self::ext(env::current_account_id())
  .with_attached_deposit(DEPOSIT)
  .with_static_gas(GAS)
  .callback_name(arguments)
);
```

gno.land

Near/WASM/JS?

```
NearPromise.new("external_address").functionCall("function_name", JSON.stringify(arguments), DEPOSIT, GAS)
.then(
  // this function is the callback
  NearPromise.new(near.currentAccountId()).functionCall("callback_name", JSON.stringify(arguments), DEPOSIT,
);
```

Solana/WASM/Rust?

CPI call depth - `CallDepth` error

Cross-program invocations allow programs to invoke other programs directly, but the depth is constrained currently to `4`.

When a program exceeds the allowed cross-program invocation call depth, it will receive a `CallDepth` error

WASM/Go?

gno.land

knqyf263/go-plugin

* Choose the interface(s) you want to expose for plugins.
  - Define Messages and Services in a proto file.
* Generate SDK for a host and plugin by go-plugin.
* Implement the Go interface defined in the plugin SDK.
* Compile your plugin to Wasm.
* Load the plugin and call the defined methods.

Why not WASM?

Solved: loading/unloading modules.

Solved: memory limitations

Solved: determinism (sort of)

Solved: CPU limitations

Unsolved: complex frameworks

Unsolved: relies on actor model (message passing)

Gno VM

* Language-level interoperability

* Fine-grained memory/cpu/storage limitations

* Automatic persistence

* Determinism for replicability

* Simple, < 30k lines of code

* Interprets the Go language

Gno VM - Stack-based AST VM

gno.land

```go
type Machine struct {

	// State
	Ops        []Op          // operations stack
	Values     []TypedValue  // buffer of values to be operated on
	Exprs      []Expr        // pending expressions
	Stmts      []Stmt        // pending statements
	Blocks     []*Block      // block (scope) stack
	Frames     []*Frame      // func call stack
	Package    *PackageValue // active package
	Realm      *Realm        // active realm
	Alloc      *Allocator    // memory allocations
	Exceptions []Exception   // exceptions stack
	NumResults int           // number of results returned
	Cycles     int64         // number of "cpu" cycles performed

	// Configuration
	...
}
```

Gno VM - Op codes

gno.land

```
/* Control operators */
OpHalt                Op = 0x01 // halt (e.g. last statement)
OpNoop                Op = 0x02 // no-op
OpExec                Op = 0x03 // exec next statement
OpPrecall             Op = 0x04 // sets X (func) to frame
OpCall                Op = 0x05 // call(Frame.Func, [...])
OpReturn              Op = 0x07 // return ...
...

/* Unary & binary operators */
OpUpos   Op = 0x20 // + (unary)
OpLor    Op = 0x26 // ||
OpLand   Op = 0x27 // &&
OpEql    Op = 0x28 // ==
...

/* Other expression operators */
OpEval          Op = 0x40 // eval next expression
OpBinary1       Op = 0x41 // X op ?
OpIndex1        Op = 0x42 // X[Y]
OpIndex2        Op = 0x43 // (_, ok :=) X[Y]
OpSelector      Op = 0x44 // X.Y
OpSlice         Op = 0x45 // X[Low:High:Max]
...
```

```
/* Type operators */
OpFieldType       Op = 0x70 // Name: X `tag`
OpArrayType       Op = 0x71 // [X]Y{}
OpSliceType       Op = 0x72 // []X{}
OpPointerType     Op = 0x73 // *X
...

/* Statement operators */
OpAssign       Op = 0x80 // Lhs = Rhs
OpAddAssign    Op = 0x81 // Lhs += Rhs
OpDefine       Op = 0x8C // X... := Y...
OpInc          Op = 0x8D // X++
...

/* Decl operators */
OpValueDecl Op = 0x90 // var/const ...
OpTypeDecl  Op = 0x91 // type ...

/* Loop (sticky) operators (>= 0xD0) */
OpSticky            Op = 0xD0 // not a real op.
OpBody              Op = 0xD1 // if/block/switch/select.
OpForLoop           Op = 0xD2
OpRangeIter         Op = 0xD3
...
```

gno.land

Gno VM - Persistence

There are two types of packages

* Pure packages - immutable, stateless
* Realm packages - mutable, stateful

Any changes to Realm packages are persisted

(at the end of a transaction boundary)

Gno VM - Persistence

gno.land

```go
// gno.land/r/alice/arealm
package arealm

var x int

func GetX() int {
	return x
}

func SetX(n int) {
	x = n
}
```

```go
// gno.land/r/bob/brealm
package brealm

import "gno.land/r/alice/arealm"

func IncrAlice() {
	x := arealm.GetX()
	arealm.SetX(x+1)
}
```

Gno VM - Persistence

gno.land

A transaction is just function call (that crosses realm boundaries).

```
Frame3: Pkg   = gno.land/p/avl,           Func = Set
Frame2: Realm = gno.land/r/alice/alream, Func = SetItem
Frame2: Realm = gno.land/r/alice/alream, Func = SetX        v__ realm boundary
Frame1: Realm = gno.land/r/bob/brealm,   Func = IncrAlice   ^
```

Lessons learned

gno.land

Interpreter VMs need memory management for performance.

* e.g. each for/if/range/select/switch/call creates a *Block
* `go tool pprof`
* `go tool pprof --alloc_space --alloc_counts`
* `go tool compile -S`
* pool.Get()/pool.Put()

Lessons learned

Primitive types as interface values allocate pointers.

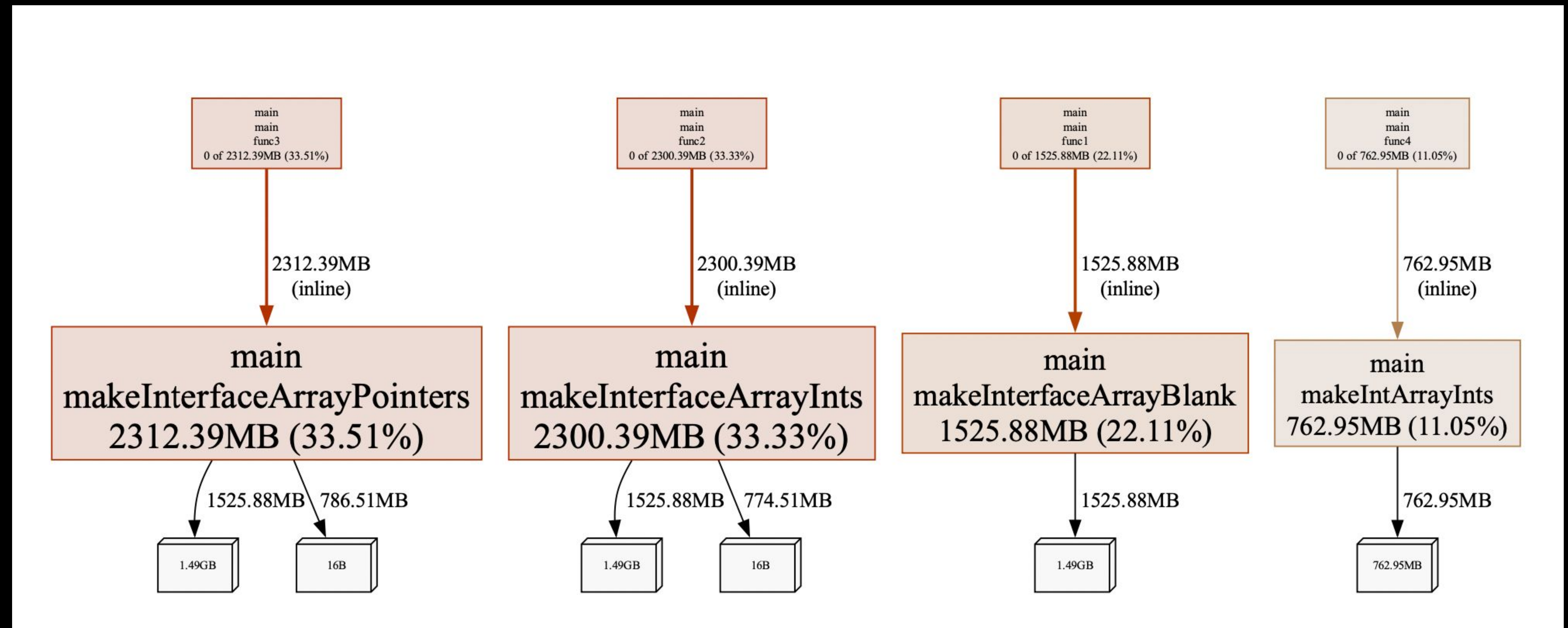
Lessons learned

```go
func makeInterfaceArrayBlank(N int) {
	x := make([]interface{}, N)
}

func makeInterfaceArrayInts(N int) {
	x := make([]interface{}, N)
	for i := 0; i < N; i++ {
		x[i] = i
	}
}

func makeInterfaceArrayPointers(N int) {
	x := make([]interface{}, N)
	for i := 0; i < N; i++ {
		p := &i
		x[i] = p
	}
}

func makeIntArrayInts(N int) {
	x := make([]int, N)
	for i := 0; i < N; i++ {
		x[i] = i
	}
}
```



gno.land

Lessons learned

```go
type TypedValue struct {
	T Type    `json:",omitempty"` // never nil
	V Value   `json:",omitempty"` // an untyped value
	N [8]byte `json:",omitempty"` // numeric bytes
}
```

gno.land

Lessons learned

Avoid switching on interface values.

```go
type PointerType struct {...}
type ArrayType struct {...}
type SliceType struct {...}
type StructType struct {...}
type FuncType struct {...}
type MapType struct {...}
type InterfaceType struct {...}
type TypeType struct {...}
type DeclaredType struct {...}
type PrimitiveType int
```

```go
const (
	InvalidType PrimitiveType = 1 << iota
	UntypedBoolType
	BoolType
	UntypedStringType
	StringType
	IntType
	Int8Type
	Int16Type
	UntypedRuneType
	Int32Type
	Int64Type
	UintType
	Uint8Type
	DataByteType
```

Lessons learned

gno.land

```go
var lv, rv TypedValue
...
switch baseOf(lv.T) {
case StringType, UntypedStringType:
        lv.V = alloc.NewString(lv.GetString() + rv.GetString())
case IntType:
        lv.SetInt(lv.GetInt() + rv.GetInt())
case Int8Type:
        lv.SetInt8(lv.GetInt8() + rv.GetInt8())
case Int16Type:
        lv.SetInt16(lv.GetInt16() + rv.GetInt16())
case Int32Type, UntypedRuneType:
        lv.SetInt32(lv.GetInt32() + rv.GetInt32())
case Int64Type:
        lv.SetInt64(lv.GetInt64() + rv.GetInt64())
case UintType:
```

Lessons learned

Scope != Allocation

```
var ptrs []*int
for {
        i := 0
        ptrs = append(ptrs, &i)
}
```

Lessons learned

Scope != Allocation

```
	var ptrs []*int
LABEL1:
	i := 0
LABEL2:
	i += 1
	ptrs = append(ptrs, &i)

	// goto LABEL1, then LABEL2, then break
	switch len(ptrs) {
	case 1:
		goto LABEL1
	case 2:
		goto LABEL2
	}

	// print pointers
	for _, ptr := range ptrs {
		println(ptr, *ptr)
	}
```

gno.land

Lessons learned

Go reflection is limited

* cannot create named types
→ cannot make recursive types
* cannot create interfaces

→ limited/hacky "go-native" support, will be removed.

Future Work

Upgrading (runtime) logic will be a challenge.

Bugs happen. What then?

Future Work

* Replacing a function or method w/ same signature is OK.
  → like iOS "swizzle".
* Appending fields to a struct:
  → will break old logic w/ use of reflection.
  → will break old logic w/ use of .(type) checks.
* Appending methods is almost OK,
  → but will break old logic w/ use of .(type) checks.
* User-land upgrade patterns preferred.

Future Work

intra-transaction GC

* piggy back on Go's runtime GC (today)
* but Go’s GC doesn’t make free memory available (yet)

→ increment memory counter for every value allocation

→ when memory limit is reached, count everything reachable

Future Work

global persistent GC

* after a transaction, all reachable objects are saved to disk.
* cycles will lead to persistent-memory-leaks.
* PLAN 0: don't create post-transaction cycles.
* PLAN 1: implement a GC for persistent objects.
* PLAN 2: extend the language with ownership rules.

Gno.land is...

* a distributed multi-user language-based operating system.
* the Go lover's answer to Ethereum.
* a repository of open auditable Gno code.

gno.land

/r/gnoland/home

[source] [help]

Welcome to gno.land

We’re building gno.land, the first open-source smart contract system, using Gno, an interpreted and fully deterministic variation of the Go programming language for succinct and composable smart contracts.

Thank you!

gno.land

Website

gno.land

Gno Docs

docs.gno.land

Gno Playground

play.gno.land

Twitter

@_gnoland

Github

gnolang/gno

We’re hiring!

@jaekwon