CS4215 Standard Project 2
Concurrent Virtual Machine for Go

Teow Hua Jun (A0251683B)
Lim An Jun (A0252731L)

1 Running The Code

Repository: `https://github.com/huajun07/go-virtual-machine`

Online demo: `https://huajun07.github.io/go-virtual-machine`

2 Scope

We implemented the baseline sequential language constructs (variable and function declarations, blocks, conditionals statements and expressions, while loops) and concurrent constructs (Go routines, wait group). In addition, we implemented channels, select statements and defer statements.

We also implemented the three optional components: memory management, type checking, and visualization of heap and runtime stack.

3 Overview

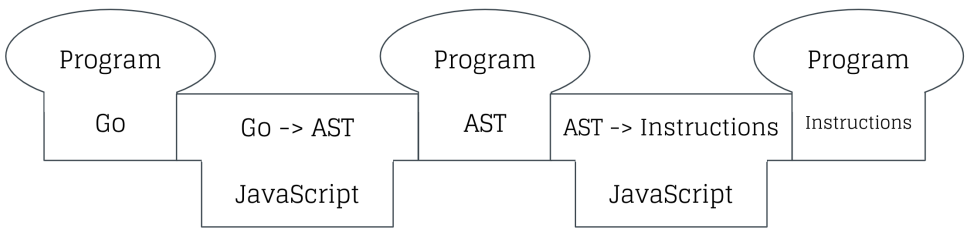
Our virtual machine runs in 3 stages.

1. Parsing: Parse the source code written in Golang into an Abstract Syntax Tree of tokens.
2. Compilation: Recursively compile the tokens to emit instructions.
3. Execution: Execute the instructions with our virtual machine.

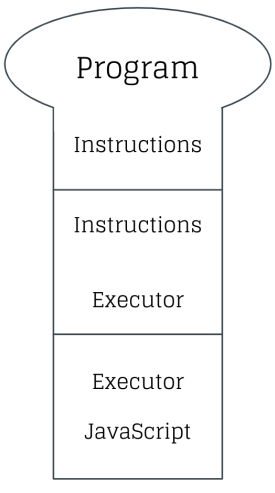
The below tombstone diagrams illustrate the 3 steps above, where JavaScript can be executed on any supported platform (e.g. on a browser running on x86 architecture).

3.1 Language Specifications

We mostly implemented a sublanguage of Golang, hence for the constructs that we implemented, their syntax can be found in Golang specifications, specifically go1.22 released on Feb 6, 2024.



(a) Parsing and Compilation



(b) Execution

3.2 Tokens

The source code is parsed into tokens, which closely follows the tokens in Golang specifications. The list of tokens and their description can be found in Appendix A.

3.3 Compilation

The tokens form an Abstract Syntax Tree, with a SourceFile token at the root. We then recursively compile the tokens into machine instructions.

3.4 Execution

Finally, we execute the instructions in our virtual machine. The section below covers our instruction set and what each instruction does.

4 Instruction Set

4.1 Load Instructions

Instructions	Parameters	Description	Execution
LDCI	`val`: `value` `data_type`: `Type`	Load Constant	Creates a Node of that data type and assign it the value val, pushing it on the OS
LDD	`data_type`: `Type`	Load Default	Creates a Node of that data type with its default value
LDA	`length`: `number`	Load Array	Creates a new array on the heap, with its values popped from the `OS` in reverse order. Pushes the array address onto `OS`.
LDAE		Load Array Element	Pops the index and array address from `OS`, then pushes the index element of the array onto the `OS`.
LDS		Load Slice	Pops the end index, start index, and underlying array address from `OS`. Constructs a slice object on the heap, then pushes the address onto `OS`.
LDSE		Load Slice Element	Pops the index and slice address from `OS`, then pushes the index element of the slice onto the `OS`.
LD	`frame_idx`: `number` `var_idx`: `number`	Load Variable	Retrieves the address of the variable at frame `frame_idx` index `var_idx` and push the address on the OS
LDP		Load Package	Pop a string address from `OS`, and load the corresponding package of that name.

4.2 Expressions and Declarations

Instructions	Parameters	Description	Execution
UNARY	`op`: `string`	Unary Operator	Pop value from the OS and apply the `op` unary operator on the value saving the result on a new node on the heap and pushing this address on the OS
BINOP	`op`: `string`	Binary Operator	Pop two values from the OS and apply the `op` binary operator on the values saving the result on a new node on the heap and pushing this address on the OS
BLOCK	`frame`: `Type[]` `for_loop`: `boolean`	Enter Block Op	Extends current evironment with new frame with variable of various types (provided in `frame`) pushing it into our `RTS`. The variables in the frame are created with their default values in the heap. Note also store whether environment is a for loop block
EXIT_BLOCK		Exit Block Op	Pops the current environment from `RTS`

STORE		Store	First pop source address from OS. Then pop destination address from OS. Then copy data from source to destination address. (Assumes two nodes are of same length)
SLICEOP		Slice Operation	Pops the end index, start index, and object address from the OS. Then we alloacte a new Slice node pointing to that object, sliced from start index to end index. Then we push this new slice's address onto the OS.
SELECTOP		Selector Operation	Pops the field/method name, and object address from the OS. Then we call the object's selection function to get its corresponding field/method's address, and push it onto the OS.

4.3 Control Statements

Instructions	Parameters	Description	Execution
JUMP	`addr: number`	Jump	Set PC to addr
JUMP_IF_FALSE	`addr: number`	Jump If False	Pop value from OS. If the value is false then set PC to addr
JUMP_LOOP	`addr: number`	Jump and Exit Loop	Set PC to addr Also continuously pop from RTS until it encounters a for loop block

4.4 Functions

Instructions	Parameters	Description	Execution
LF	`PC: number`	Load Function	Creates a new function node on the heap with PC as given in the instruction and with current environment. Then push this function literal on the OS
CALL	`args: number`	Call	**Assumes that there are args on the OS followed by the closure.** Starts the function call by prepping the necessary variables Access the arg+1-th element on the OS (the closure). Push CallRef node on RTS with current PC Push the environment of the closure on RTS Set PC to the closure's PC
FUNC_BLOCK	`args: number`	Function Block	Creates the functional scope of a function, is at the start of every function call. Identical to BLOCK instruction for the first part. Then pops args values from the OS and assign them in reverse order to the first args variables in the frame. Also popping the closure
DEFERRED_CALL	`args: number`	Deferred Call	Pops arguments in reverse order, then pops the function / method object from OS. Creates a new DeferredFunction or DeferredMethod node using those arguments, and pushes it onto the OS
RET		Return	If the context's current DeferStack is empty, then we return normally. Otherwise take the pop a DeferredFunction / DeferredMethod node from the DeferStack and execute it, jumping back to this instruction when done.

4.5 Concurrent

Instructions	Parameters	Description	Execution
FORK	addr: number	Fork	Creates a new child thread context with the same PC and current environment but with an empty OS and RTS Set parent thread PC to be addr and child thread to be PC+1
LDCH		Load Channel	Pop and get the channel buffer size from the OS. Create a new channel node on the heap and push its address on the OS
LDCR	recv: boolean PC: number	Load Channel Request	Pop the value from the OS, and clone it and use the cloned address as the data. Create a new channel request node on the heap using the parameters as well as the current context and acquired data address and push its address on the OS.
TRY_CHAN_REQ		Try Channel Request	Pop the channel request from the OS and try to fulfil that channel, if unsuccessful then add request to channel waitlist and block.
SELECT	cases: number default: boolean	Select Block	Pop cases number of channel requests from the OS. Shuffle then and check if any of them can be fulfilled if yes, then fulfil that request and jump to the request's PC If not, then add all requests to waitlist and block. Add all waitlists to context waitlist as well so that they can all be erased if any of them are fulfilled. If default is true, then pop OS for default instruction address and if none of the requests can be fulfilled jump to default address **Note that the channel requests are randomised on EVERY execution of the SELECT instruction**

4.6 Built-in

Instructions	Parameters	Description	Execution
BUILTIN_LEN		Built-in Length	Pops an object from the OS, and executes that object's len method (which is guarenteed to exist by our type checking), and pushes its length onto the OS.
BUILTIN_CAP		Built-in Capacity	Pops an object from the OS, and executes that object's cap method (which is guarenteed to exist by our type checking), and pushes its capacity onto the OS.

5 Inference Rules

In this section, we will describe inference rules of our virtual machine. We will not provide inference rules for basic instructions such as loading a constant or entering a block scope as this is fairly well discussed during lectures and lessons. But rather focus our attention on our concurrency constructs. Namely goroutines, channels and waitgroups.

Note that we are not covering select-case opearations as they involve more complex instructions.

5.1 Definitions

In the subsequent inference rules, we will use the following notations:

$\emptyset$: Empty List
$s(pc)$: Instruction at instruction address pc
$l(X)$: Length of list X
$Clone(x)$: Cloned address of x

For a context, it will be represented by (pc, os, rts) where

pc: Program Counter
os: Operand Stack
rts: Runtime Stack

Note here that we are ignoring the waitlist and defer stack for simplicity.
The context queue which contain queue of unblocked contexts will be represented by $ctxs$.

A channel request info is represented by $(recv, p, ctx, x)$ where

$recv$: Boolean for if request is recv
p: New program counter if request is fulfilled
ctx: Context that made request
x: Data

A channel request is represented by (ch, req) where

ch: Channel
req: Request Info

A channel is represented by $(sz, buf, send, recv)$ where

sz: Buffer Size
p: New program counter if request is fulfilled
$send$: Send Waitlist
$recv$: Recv Waitlist

A waitgroup is represented by (cnt, w) where

cnt: Counter value
w: List of waiting contexts

A method is represented by $(obj, name)$ where

obj: Object which method is called
$name$: String of method name

5.2 Concurrent Constructs

5.2.1 Fork Instruction

A fork instruction pushes a new context into the context queue

$$\frac{s(pc) = \text{FORK } pc'}{(pc, os, e.rts).ctxs \rightrightarrows (pc', os, e.rts).ctxs.(pc+1, \emptyset, e)}$$

A context switch transition would be the following:

$$\frac{}{c.ctxs \rightrightarrows ctxs.c}$$

5.2.2 Channel Instructions

A load channel request instruction pushes a new channel request into the os

$$\frac{s(pc) = \text{LDCR } recv\ pc' \quad y = Clone(x)}{\underset{c.ctxs}{c = (pc, x.ch.os, rts)} \rightrightarrows \underset{c.ctxs}{c = (pc+1, (ch, (recv, pc', c, y)).os, e.rts)}}$$

Try Channel Request instruction
Case 1:
Send request cannot be fulfilled

$$\frac{s(pc) = \text{TRY\_CHAN\_REQ} \quad l(buf) = sz \quad recv = \emptyset}{\begin{array}{c}ch = (sz, buf, send, recv) \\ c = (pc, (ch, (false, pc', c, x)).os, rts) \rightrightarrows \\ c.ctxs\end{array} \quad \begin{array}{c}ch = (sz, buf, send.(false, pc', c, x), recv) \\ c = (pc, os, rts) \\ ctxs\end{array}}$$

Case 2:
Recv request cannot be fulfilled

$$\frac{s(pc) = \text{TRY\_CHAN\_REQ} \quad l(buf) = sz \quad send = \emptyset}{\begin{array}{c}ch = (sz, buf, send, recv) \\ c = (pc, (ch, (true, pc', c, x)).os, rts) \rightrightarrows \\ c.ctxs\end{array} \quad \begin{array}{c}ch = (sz, buf, send, recv.(true, pc', c, x) \\ c = (pc, os, rts) \\ ctxs\end{array}}$$

Note that in both of these cases, the context is blocked and removed from the context queue

Case 3:
Recv request can be fulfilled
Case 3.1:
From buffer and send waitlist empty

$$\frac{s(pc) = \text{TRY\_CHAN\_REQ} \quad send = \emptyset}{\begin{array}{c}ch = (sz, y.buf, send, recv) \\ c = (pc, (ch, (true, pc', c, x)).os, rts) \rightrightarrows \\ c.ctxs\end{array} \quad \begin{array}{c}x = y \\ ch = (sz, buf, send, recv) \\ c = (pc', os, rts) \\ c.ctxs\end{array}}$$

Case 3.2:
From buffer and send waitlist not empty

$$\frac{s(pc) = \text{TRY\_CHAN\_REQ}}{\begin{array}{c} ch = (sz, y.buf, (false, pc2', c2, y2).send, recv) \\ c = (pc, (ch, (true, pc', c, x)).os, rts) \\ c2 = (pc2, os2, rts2) \\ c.ctxs \end{array} \rightrightarrows \begin{array}{c} x = y \\ ch = (sz, buf.y2, send, recv) \\ c = (pc', os, rts) \\ c2 = (pc2', os2, rts2) \\ c.ctxs.c2 \end{array}}$$

Case 3.3:
From send waitlist

$$\frac{s(pc) = \text{TRY\_CHAN\_REQ} \quad buf = \emptyset}{\begin{array}{c} ch = (0, buf, (false, pc2', c2, y).send, recv) \\ c = (pc, (ch, (true, pc', c, x)).os, rts) \\ c2 = (pc2, os2, rts2) \\ c.ctxs \end{array} \rightrightarrows \begin{array}{c} x = y \\ ch = (0, buf, send, recv) \\ c = (pc', os, rts) \\ c2 = (pc2', os2, rts2) \\ c.ctxs.c2 \end{array}}$$

Case 4:
Send request can be fulfilled
Case 4.1:
Into buffer

$$\frac{s(pc) = \text{TRY\_CHAN\_REQ} \quad l(buf) < sz}{\begin{array}{c} ch = (sz, buf, send, recv) \\ c = (pc, (ch, (false, pc', c, x)).os, rts) \\ c.ctxs \end{array} \rightrightarrows \begin{array}{c} ch = (sz, buf.x, send, recv) \\ c = (pc', os, rts) \\ c.ctxs \end{array}}$$

Case 4.1:
From recv waitlist

$$\frac{s(pc) = \text{TRY\_CHAN\_REQ} \quad buf = \emptyset}{\begin{array}{c} ch = (sz, buf, send, (true, pc2', c2, y)recv) \\ c = (pc, (ch, (false, pc', c, x)).os, rts) \\ c2 = (pc2, os2, rts2) \\ c.ctxs \end{array} \rightrightarrows \begin{array}{c} y = x \\ ch = (sz, buf, send, recv) \\ c = (pc', os, rts) \\ c2 = (pc2', os2, rts2) \\ c.ctxs.c2 \end{array}}$$

5.2.3 Wait Groups

Accessing Methods of Objects
Example: obj is a waitgroup and $name$ is "Add"

$$\frac{s(pc) = \text{SELECTOP}}{(pc, name.obj.os, rts).ctxs \rightrightarrows (pc + 1, (obj, name).os, rts).ctxs}$$

Wait Group Add
x is the number to add to the wait group counter

$$\frac{s(pc) = \text{CALL } 1 \quad name = \text{"Add"}}{\begin{array}{c} obj = (cnt, w) \\ (pc, x.(obj, name).os, rts).ctxs \end{array} \rightrightarrows \begin{array}{c} obj = (cnt + x, w) \\ (pc + 1, os, rts).ctxs \end{array}}$$

Wait Group Wait
Case 1:
Wait group Counter is 0

$$\frac{s(pc) = \text{CALL } 0 \quad name = \text{"Wait"} \quad cnt = 0}{(pc, ((cnt, w), name).os, rts).ctxs \rightrightarrows (pc + 1, os, rts).ctxs}$$

Case 2:
Wait group Counter is not 0

$$\frac{\begin{array}{c} s(pc) = \text{CALL } 0 \quad name = \text{"Wait"} \quad cnt \neq 0 \\ \hline obj = (cnt, w) \qquad\qquad obj = (cnt, w.c) \end{array}}{\begin{array}{c} c = (pc, (obj, name).os, rts) \rightrightarrows c = (pc + 1, os, rts) \\ c.ctxs \qquad\qquad\qquad ctxs \end{array}}$$

Wait Group Done
Case 1:
Wait group Counter does not reduce to 0

$$\frac{s(pc) = \text{CALL } 0 \quad name = \text{"Done"} \quad cnt > 1}{\begin{array}{c} obj = (cnt, w) \qquad\qquad obj = (cnt - 1, w) \\ (pc, (obj, name).os, rts).ctxs \rightrightarrows (pc + 1, os, rts).ctxs \end{array}}$$

Case 2:
Wait group Counter reduces to 0

$$\frac{s(pc) = \text{CALL } 0 \quad name = \text{"Done"} \quad cnt = 1}{\begin{array}{c} obj = (cnt, w) \qquad\qquad obj = (cnt - 1, \emptyset) \\ (pc, (obj, name).os, rts).ctxs \rightrightarrows (pc + 1, os, rts).ctxs.w \end{array}}$$

Note that the case where $cnt \leq 0$ is not present as this transition is not allowed and if exist will throw an error.

6 Memory Management

Our memory management mainly consist of a single `Heap` object which uses a Javascript `ArrayBuffer` as its underlying memory of bytes. For our virtual machine we will be processing values in word length of 4 bytes.

6.1 Allocation and Freeing

The heap provides two basic APIs, namely `Heap::allocate(size: number)` which allocates a node with size $\geq$ `size` and returns the address of this allocated node. As well as `Heap::free(addr: number)` which frees the node at address `addr`

To efficiently allocate a node that satisfies the size requirement while also minimising wasted memory, a buddy block allocation algorithm is used. Namely, in the heap 32 variables are initialised that point to a head of a linked list of free nodes of different levels, where the linklist of the i-th level consist of free node of size 2^i.

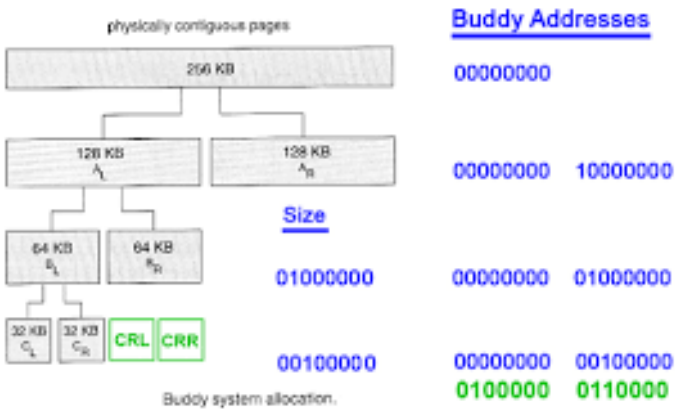


Figure 2: Buddy Block Allocation

Thus, when allocating, a free node of a sufficiently large size can be found quickly in $O(\log(maxsize))$. If the smallest valid block is too large, thus possibly wasting unused memory, it can also be split into a smaller layer to be used for subsequent allocations.

When freeing a node, the freed node can also be merged if its sibling node is also free to form a larger free block. Note that this involves deleting nodes from the linked list. Thus the linked list needs to be doubly linked.

6.2 Node Structure

For most browsers, the maximum size of `ArrayBuffer` is 2^{32} bytes.

Note that for each free node, we need to store the previous and next node in the link list since its a doubly linked list. There is no way we can store this in a single word. Thus every node in the heap is restricted to have a size of at least 2 words.

Now there are $2^{32}/(2 \times 4) = 2^{29}$ possible addresses for the nodes this can be stored in 29 bits, furthermore there are 32 possible levels a node can be in which takes $\log_2(32) = 5$ bits. Thus, we can store the values in our free nodes as such: Fitting all our metadata nicely in 2 words exactly.

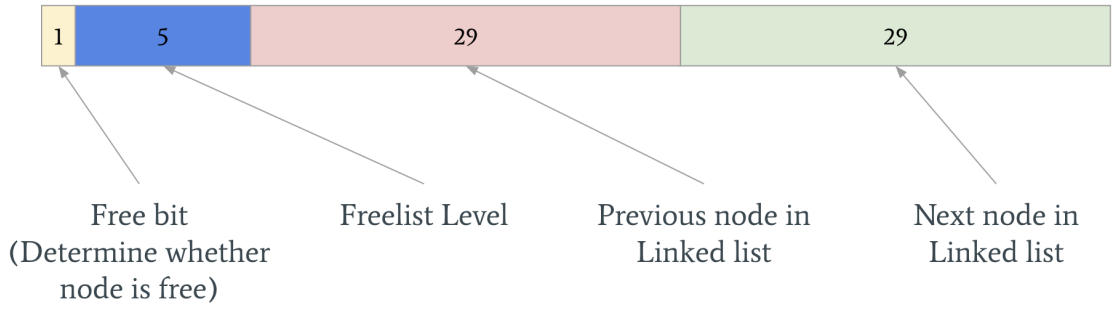


Figure 3: Free Node in Heap

For allocated nodes, we still still maintain the free bit and level data, but include a bit for mark and sweep as well as a heap type tag, which is used to distinguish between the multiple node types that exists on the heap. (Which will be explained later)

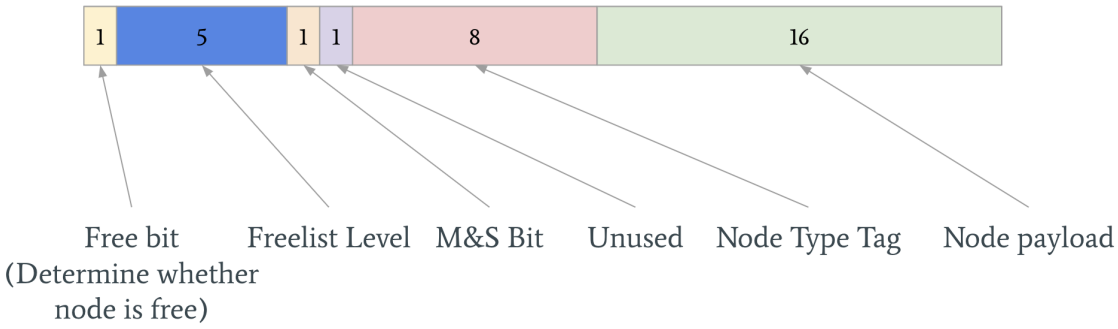


Figure 4: Allocated Node in Heap

6.3 Nodes

As mentioned above, every allocated node in the heap has a node type tag which allows the vm to identify what kind of node it is and what values it contain and where in the node it contains it. For example, primitive type nodes could store their value in the 2nd word of the node, while link list entry nodes could store their prev and next pointers in their 2nd and 3rd word. The full list of nodes and their values can be found in Appendix B.

6.4 Garbage Collection

Mark and Sweep garbage collection is implemented, where the roots are all the contexts and temporary roots which is a stack on the heap that stores dangling nodes that may be accidentally swept. Which could happen when declaring multiple nodes that are not yet reference by anything in the context.

The marking traversal is implemented by implementing a `get_children` method in all heap node types to retrieve the references to other nodes on the heap. For example, for the context node this would consist of the addresses to its OS, RTS, waitlist and defer stack.

6.5 Unique Structures

6.5.1 Environments

As mentioned in 8.3., environments do not store a list of frames but instead store a single frame and store references to its ancestor environments that it has been extended from to gain access to frames in outer scopes. Namely, we store the every 2^i-th ancestor of the environment. This allows binary lifting to be performed to extend an environment in $O(\log(scopes))$ (to get the ancestors) instead of $O(scope)$ by just copying all the frames. And allows access of a frame in $O(\log(scopes))$ as well.

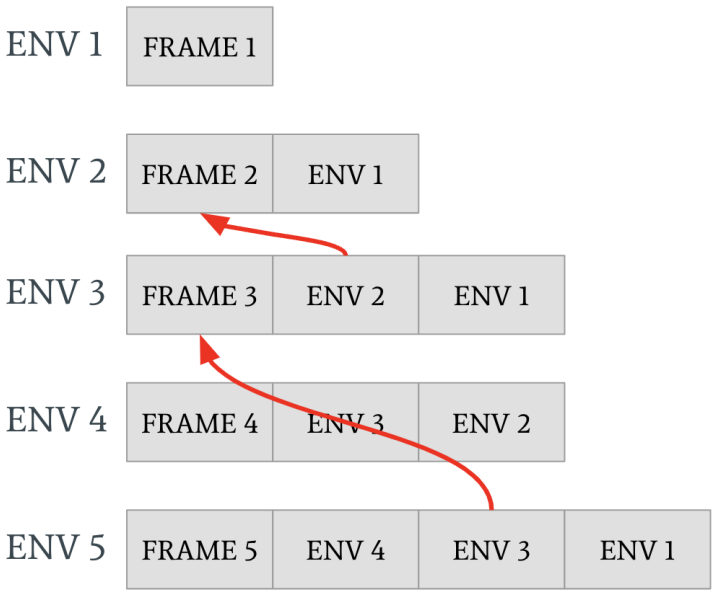


Figure 5: Retrieve Frame 2

6.5.2 Resizable Structures

For structures such as `StackNode` and `QueueNode`, the number of elements to be stored in the list is unknown as such when the number of elements exceed the initially allocated size we need to resize the list, allocating more memory and copying the items over. Conversely to reduce memory wastage, when the number of items decrease significantly we can resize to a smaller list size. This is done through a standard amortised resizing algorithm.

However, when a list resizes, the address of the list changes and external nodes that reference the stack/queue would then be referencing the wrong address. Thus to solve this, we have a `StackNode` and `StackListNode` where the `StackListNode` stores the actual list and address can change when resizing but the `StackNode` will reference this new address and be updated. Thus, other nodes will reference `StackNode` which has a constant address.

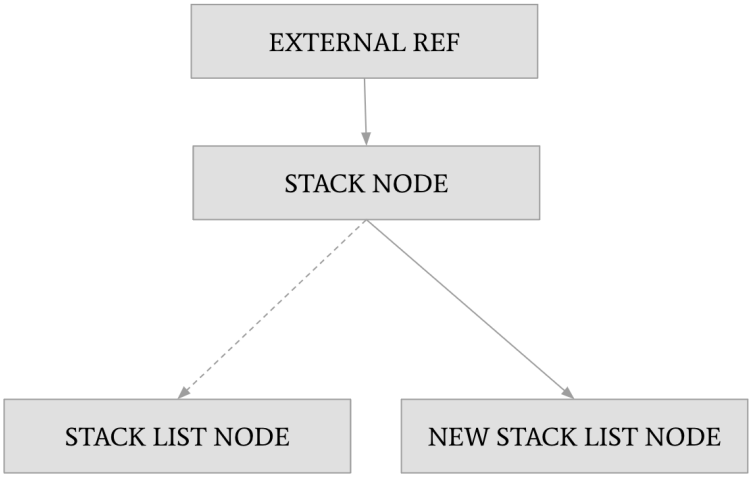


Figure 6: Stack Resizing

7 Type Checking

We define the following types: `NoType, BoolType, IntType, FloatType, StringType, FunctionType(parameters[], return), ArrayType(element, length), SliceType(element), ChannelType(element)`

Every token will return its own type when it is compiled. This enables us to do type checking, and to throw a compilation error if there are any type errors. In addition, during compilation we maintain a type environment in the form of a stack of type frames. Type frames store the type of an identifier within the current scope. Each time we enter a scope, we extend the type environment by adding a new type frame. Each time we exit a scope, we pop the type environment to remove the top frame.

Each type frame also maintains an expected return type, which is modified when we enter a new function. This lets us type check return statements (or lack of) within the function.

8 Visualisation

Visualisation is implemented by adding additional debugging symbols and data in both the compilation and execution of the code. Snapshots are then taken every time an instruction is ran which contain data about the heap as well as runtime stack of the contexts.

8.1 Debugging Symbols

Debugging symbols help the debugger access useful data for the visualisation in the snapshot

8.1.1 Identifier Map

This is a map of variable addresses to their identifier in the code. This is implemented by storing the identifiers in the BLOCK instructions when creating a new frame and then storing a key value pair from the allocated address to the identifier.

8.1.2 Environment Map

This is a map of environment addresses to their identifier. There are compile time identifiers to specify what type of scope it is, such as ”FOR INIT BLOCK”, ”FOR INNER BLOCK”, ”ANONY FUNC”, ”IF INIT BLOCK”. There are also runtime identifiers in the case of function calls, where the identifier of the closure is stored as the identifier of the environment. For example, if the ”main” function is called, the function scope will be displayed as ”main”.

8.1.3 Context Map

This maps context addresses to a thread id that is incremented evertime a thread is created.

8.1.4 TokenLocation

This maps token and instructions to the code segment that this instruction is created from.

8.2 Snapshot data

This is the data that is captured after every instruction to be displayed at the visualisation

8.2.1 Contexts

The state about the context (blocked/unblocked) is stored as well as its data which will be discussed below.

8.2.2 Instructions

Show a contiguous range of 7 instructions that contain the current instruction that was just ran. Each instruction has a `toString()` method that may display information about its parameters.

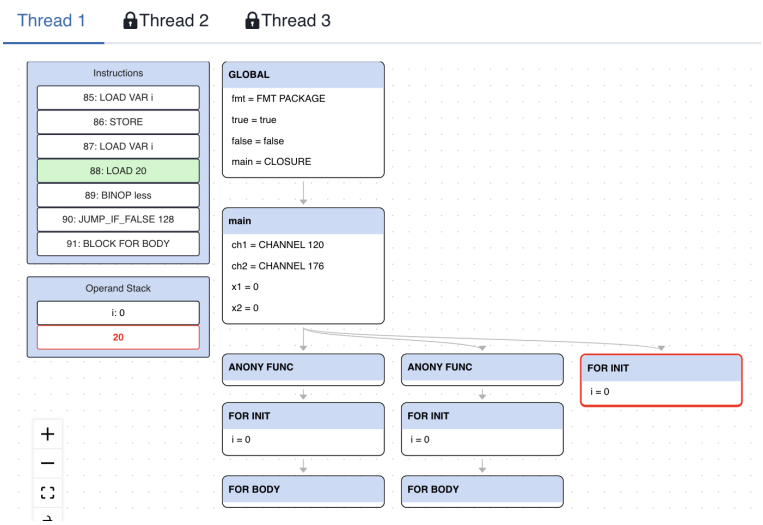


Figure 7: Visualisation

8.2.3 Operand Stack

Displays the entire operand stack. Contains value located at the address of the elements in the stack as well as variable identifier if its the address of a variable. Similar to instructions, each heap node has a `toString()` method that may display information about its inner values.

8.2.4 Environments

For each context, through a traversal find all relevant environments and parse it as a tree-like graph. Showing the frame of the environment including the variable identifiers and values.

8.2.5 TokenLocation

Allowing the code segment to be highlighted when the instruction is being run to display the progress of the program.

8.2.6 Output

The current stdout is also captured.

9 Running Locally

To run the site locally:

```
git clone https://github.com/huajun07/go-virtual-machine
cd go-virtual-machine
nvm install   # to use correct node version
npm install
npm run dev
```

10 Tests

We have 82 tests in our `tests/` folder. They can be run by the command `npm run test`. The tests check our heap management, type checking and all the language constructs we implemented.

Here we list 10 sample testcases:

Test	Purpose

```
c1 := make(chan int)
n := 25

go func() {
 for i := 0; i < n; i++ {
  select {
  case c1 <- 1:
   fmt.Println("Write 1 1")
  case <-c1:
   fmt.Println("Read 1 2")
  }
 }
}()

go func() {
 for i := 0; i < n; i++ {
  select {
  case c1 <- 2:
   fmt.Println("Write 2 2")
  case <-c1:
   fmt.Println("Read 2 1")
  }
 }
}()
for i:=0; i < 100; i++{}
```

Test that select statements and channels work properly. We expect to see that a goroutine cannot send to itself through an unbounded channel, and it must send to other goroutines only.

```
count := 0
var wg sync.WaitGroup
for i := 0; i < 1000; i++ {
  wg.Add(1)
  go func() {
    count++
    wg.Done()
  }()
}
wg.Wait()
fmt.Println(count)
```

Test that wait groups work properly, and we do wait for the counter to reach 0 before continuing.

`var a int = 1`
 	
`func f(x, y int) int {`
`  return x + y + a`
`}`
 	
`func main() {`
`  f := func(x, y int) int {`
`  return x + y + 100`
`  }`
`  fmt.Println(f(1, 2))`
`}`
	Test that we can create closures, and that variable shadowing works. Here we expect to use the local `f` closure instead of the global function.

`a := [3][3]int{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}` `fmt.Println(a[1][2])`	Test that nested arrays work properly. We expect to see 6 printed.

`defer func(){ fmt.Println("!!!") }()`
`defer func(){ fmt.Println("world") }()`
`fmt.Println("hello")`
	Test that defers execute in reverse of the order they were encountered. We expect to see hello, world, then !!!

`var a []int = []int{1, "wrong type", 3}`	Test that slice literals are type-checked and must match the correct type. We expect a compile error to be thrown, as we cannot use a string as an integer.
`a := 1 * 1.0`	Test that operators are type-checked. We expect a compile error to be thrown, as we cannot have a binary operator operate on different types (int and float).
`var a chan int = make(chan string)`	Test that builtin functions are type-checked. We expect a compile error to be thrown, as we cannot assign a string channel to an integer channel.

`func f(x int) int {`
`  if x == 0 {`
`  return 0`
`  }`
`  return f(x - 1) + 1`
`}`
 	
`func main() {`
`  fmt.Println(f(10))`
`}`
	Test that recursive functions work. We expect the 10 to be printed.

`fmt.Println(5 * -1 + 3 * 4 / 2 + 3)`	Testing that operator precedence is correct. We expect the correct value of 4 to be printed.

Appendix A - Tokens

Token	Parameters	Description
Block	`statements: Statement[]`	Represents a new scope. Example: {}
FunctionDeclaration	`name: Identifier` `func: FunctionLiteral`	Example: `func f() {}`
ShortVariable... Declaration	`identifiers: Identifier[]` `expressions: Expression[]`	Shorthand declaration of each identifier, initialized to the corresponding expression. Example: `a, b := 1, 2`

VariableDeclaration ConstantDeclaration	`identifiers: Identifier[]`
	`varType?: Type`
	`expressions: Expression[]`
		Declaration of each identifier, initialized to the corresponding expression. The type of the variable(s) is optional if it can be inferred from the expressions.
		Example: `var a int = 1`

PrimaryExpression	`operand: Identifier | Expression` `rest: PrimaryExpressionModifier[]`	A primary expression is an operand (identifier / expression) with a bunch of "modifiers", such as select / index / slice / call applied to it. Example: `a.test()[1]`
Index	`expression: Expression`	An index operation, where the expression should have integer type. Example: `[1]`
Slice	`from: Expression` `to: Expression`	A slice operation, where the from and to expressions should have integer type. Example: `[5:10]`.
Call	`expressions: Expression[]`	Calls a function or method, with the expressions as arguments. Example: `(1, 2, 3)`.
BuiltinCall	`name: string` `args: (Expression | Type)[]`	A builtin function call. This is implemented separately as certain builtin functions take a type as their first argument, while normal functions only take values. Example: `make(chan int, 1)`
Identifier	`identifier: string`	A variable identifier Example: `counter`

IntegerLiteral
FloatLiteral
StringLiteral
	`value: number`
	`value: number`
	`value: string`
		Literals.
		Example: `10, 3.14, "hello!"`

FunctionLiteral	`signature: FunctionType` `body: Block`	Example: `func () {}`
ArrayLiteral	`arrayType: ArrayType` `body: Expression[]`	Example: `[5]{1, 2, 3, 4, 5}`
SliceLiteral	`sliceType: SliceType` `body: Expression[]`	Example: `[]{1, 2, 3, 4, 5}`
UnaryOperator	`name: string` `child: Expression`	The name of the unary operator tells us which unary operator to perform on the child expression. Example: `!true`

BinaryOperator	`name: string`
	`left: Expression`
	`right: Expression`
		The name of the binary operator tells us which binary operator to perform on both the left and right expressions.
		Example: `1 + 5`

SourceFile	`declarations: FunctionDeclaration[]`	This is the root token. It contains the entire source code: all the top-level variable and function declarations, including the main function.

AssignmentStatement	`left: Expression[]`
	`operation: = | += | *=`
	`right: Expression[]`
		Expressions on the left must evaluate to an object with a memory address. They are assigned with corresponding values from the right expressions.
		Example: `a, b = 1, "hello"`

IncDecStatement	`expression: Expression[]` `operation: ++ | --`	Example: `counter++`
ReturnStatement	`returns: Expression[]`	Example: `return "hello"`
BreakStatement		Example: `break`
ContinueStatement		Example: `continue`

IfStatement	`initialization: Statement`
	`predicate: Expression`
	`consequent: Block`
	`alternative?: IfStatement | Block`
		The initialization is executed first, then we check the predicate. If true, we run the consequent, else we run the alternative if there is one.
		Example: `if i := 1; i <= 1 {} else {}`

ForStatement	`initialization: Statement`
	`condition: Expression`
	`post: Expression`
	`body: Block`
		The initialization is executed first, then in each loop iteration, we check the condition. If true, we execute the body, then execute the post expression.
		Example: `for i := 1; i < 10; i++ {}`

DeferStatement	`call: Expression`	The execution of the function call is deferred until the current function returns. Example: `defer fmt.Println("Hi")`
GoStatement	`expression: Expression`	The function call is executed in a new goroutine. Example: `go f()`
SendStatement	`channel: Identifier` `value: Expression`	Sends the value into the channel. Example: `ch <- 1`
ReceiveStatement	`identifier: Identifier` `expression: Unary`	The expression must be a receive operator (`<-ch`). Assigns the value of the channel to a variable. Example: `value <- ch`
SelectStatement	`clause: CommunicationClause[]`	Blocks until one of the communication clause is ready to run, then execute it. See below for CommunicationClause.

CommunicationClause	`body: Statement[]`
	`predicate:`
	`default |`
	`SendStatement |`
	`ReceiveStatement`
		A communication clause will execute once its predicate, a send or receive statement, is able to run.

Appendix B - Heap Nodes

Tag ID	Node	Values	Description
1	Boolean	value - 1 bit	Primitive Type
2	Number	value - 1 word	Primitive Type
3	Context	blocked? - 1 bit PC - 1 word OS - 1 word RTS - 1 word Waitlist - 1 word DeferStack - 1 word	Represents the context of a thread with references to all the structures necessary for a thread as well as data about the thread (like whether its blocked)
4	Frame	variables - `frame_sz` words	Stores all vairables in the frame
5	Environment	frame - 1 word parents - ? words	Stores the current frame and references to ancestor environments
6	Float	val - 1 word	Primitive Type
7	String	String List - 1 word	Primitive Type Stores a reference to a string list (the string is not stored directly since a variable can reference strings of different length through out its lifetime)
8	String List	value - len bytes	List of characters of the string
9	Stack	Stack List - 1 word	Represent a stack data type. Stores reference to stack list
10	Stack List	size - 1 word elements - ? words	Stores size and elements of stack
11	Function	PC - 1 word Env - 1 word	Represent a closure data type. Stores the instructional address and runtime environment of the closure
12	Call Reference	PC - 1 word	Stores the program counter the thread should return to after its function call returns
13	Array	length - 1 word elements - length words	Store the length of the array as well as address of its elements
14	Queue	Queue list - 1 word	Represent a queue data type. Store reference to queue list
15	Queue List	size - 1 word start index - 1 word end index - 1 word elements - ? words	List of elements in a circular buffer with the start and end indices as well as size
16	Linked List	head ptr - 1 word tail ptr - 1 word	Represent a linked list data type. Stores the head and tail of the link list
17	Linked List Entry	prev - 1 word next - 1 word val - 1 word	Represents the individual linked list nodes.
18	Channel	buffer_sz - 1 word buffer - 1 word send waitlist - 1 word recv waitlist - 1 word	Repesents the channel data type. With buffer as a queue node and send/recv waitlist as linkedlist nodes
19	Channel Request	Channel - 1 word Request Info - 1 word	Represent a channel request.
20	Channel Request Info	recv/send - 1 bit data - 1 word context - 1 word PC - 1 word	Represents a channel request info. Data is the address of the value to send or the location to store the recieved value respectively. The context is the context that made the request and the PC the instruction address the context should go to if the request is successful.

21	Slice	array - 1 word start - 1 word end - 1 word	Represents a slice data type. With a reference to the array its create from and the range of the array it covers.
22	Wait Group	counter - 1 word waitlist - 1 word	Represents a wait group data type. Maintains a counter and a waitlist of blocking contexts that are unblocked when the counter reaches 0.
23	Method	receiver - 1 word identifier - 1 word	Represents a method function of a object. Receiver is the object the method is being acted on and identifier is the name of the method.
24	Defer Function	function - 1 word argumentStack - 1 word	Represents a function call that has been deferred, with the Function in function, and arguments stored on a stack called argumentStack.
25	Defer Method	method - 1 word argumentStack - 1 word	Represents a method call that has been deferred, with the Method in method, and arguments stored on a stack called argumentStack.
26	Package	-	Represents an uninitialized package, before we execute the import instructions.
27	Fmt Package	-	Represents the `fmt` package.