

CS4215 Project Report:

Goose - A Concurrent Go-inspired Language

Authors: *Bharath Chandra Sudheer (A0218550J), Hoang Trong Tan (A0219767M)*

GitHub repository: <https://github.com/jushg/goose>

Slides: <https://hackmd.io/i-ogHVzS8a6pDWGXAYXkQ?view=#>

Objectives	2
Deliverables	3
Goose language & Goose Compiler	3
Gosling VM	4
Implementation	4
Gosling Instruction Set	4
Scoping / Function Calls	5
Goroutine event loop	6
Memory Management	6
Heap Implementation	6
Garbage Collection	7
Concurrency Primitives	8
Repository	9
Project Sources & Libraries	9
Install, Build & Deploy	10
Test Cases	10

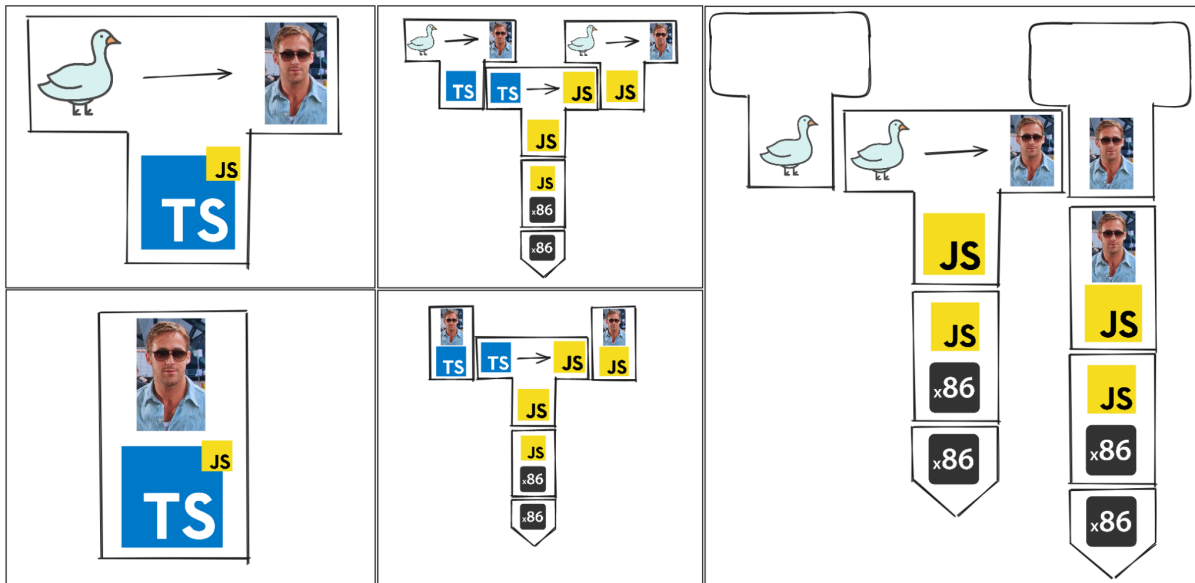
Objectives

This project aimed to develop a compiler and a concurrent virtual machine (VM) for Goose, a sublanguage of Go. Our achieved objectives include:

1. Concurrency objectives:
 - a. Support go-routines in concurrent execution
 - b. Use interleaving machine instructions for granularity of concurrency
 - c. Support concurrency primitives: `mutex`, `semaphore`, `channel`, `waitGroup`.
 - d. Visualise thread status on the frontend.
2. Sequential logic:
 - a. Key Golang syntax must be supported `for`, `if`, `func`, `var`,
 - b. Basic data types and operations on them must be supported: `bool`, `str`, `int`, and `ptr` for any type (including multiple levels of indirection)
 - c. Allow creation, use and return of `func` literals, supporting functions as first-class
3. Low-level memory management

- a. Efficient Garbage collection
- b. Visualisation of memory on the frontend

Deliverables



Pictured: T-diagrams for compiler source code (top-left) interpreter source code (bottom-left), compilation process for compiler and interpreter (middle), and Goose code being compiled to Gosling and interpreted (right).

Goose language & Goose Compiler

Goose is a statically typed language supporting basic data types (int, bool, string) and pointers (to any level of indirection, with a base type of int, bool or string). It supports the majority of sequential constructs in Go, such as **for**, **if**, **switch**, and others.

Functions are a first class object, just as in Go and `func` literals can be used as callable objects, arguments, expressions and return values to support this. Note that applicative-order evaluation is used.

Being a concurrent language, we support goroutine creation with the `go f(args)` statement, (f may be replaced with a function literal). More details on concurrency is explained in the “Concurrency Primitives” section.

Gosling VM

A concurrent VM that interprets Gosling instructions in Typescript. It provides access to thread control objects, system calls, and memory manager. There is also a frontend provided to provide easy access to it.

For the usability as an educational project, it provides critical debugging tools such as breakpoints (at the Gosling instruction level and at Goose code level), a logging visualiser (to filter, search and sort for logs to ensure correct execution) and a memory visualiser (to view operand stack, runtime stack and thread status at any breakpoint).

The frontend is deployed at goose-liard.vercel.app for convenient demonstrations.

Useful single-threaded builtins:

`triggerBreakpoint()`: pauses execution at the point in the Goose code. Operates just like the instruction-level breakpoint selection in the frontend (assumes Gosling VM system calls are available).

`print(arg)`: prints any Goose object provided to it (assumes Gosling VM system calls are available).

`max(a, b); min(a, b)`: returns the maximum / minimum of two integers.

Implementation

Gosling Instruction Set

Name	Description	Parameter
NOP	Skip this PC	-
LDC	Push a constant onto OS	value
DECL	Declare and assign new variable in the current scope	symbol, value
POP	Pop one value from the OS	-
JOF	Pop value from OS (expect that to be a boolean). Jump to toAddress if that is false.	toAddress

GOTO	Change PC to toAddress	toAddress
ENTER_SCOPE	Allocate space for all the symbols in that frame. Make this frame a special frame if needed.	scopeSymbols specialLabel (optional)
EXIT_SCOPE	Exit 1 scope if no special label presents, else pop RTS until find the special frame	specialLabel (optional)
LD	Lookup symbol in the current scope and load to OS	symbol
ASSIGN	Pop symbol and value from OS and assign value to symbol	-
CALL	Load functions symbols and parameters from OS, then create new function call frame	arity
CLEAR_OS	Clear the entire current OS stack	-
TEST_AND_SET	Load desired, expected and ptr of the atomic variable from OS, then perform an atomic test and set sequence. (Key instruction for creating concurrency primitives)	-
GOROUTINE	Spawn a new thread, expect an immediate function call instruction for spawned thread	arity
SYS_CALL	Perform an atomic sysCall function	Function symbol, number of function arguments,
ALU	Pop 1 or 2 values depending on operation and push the result	Operation (binary/unary)

Scoping / Function Calls

In the interest of keeping our operand stack and our runtime stack in the VM managed memory, we utilised **BinPtr** to create a LISP-style 'pair-like' structure of lists. A series of scopes are lists of lists, and lookups and assigns happen by iterating over the frames to identify the correct pointer and modifying the node it points to.

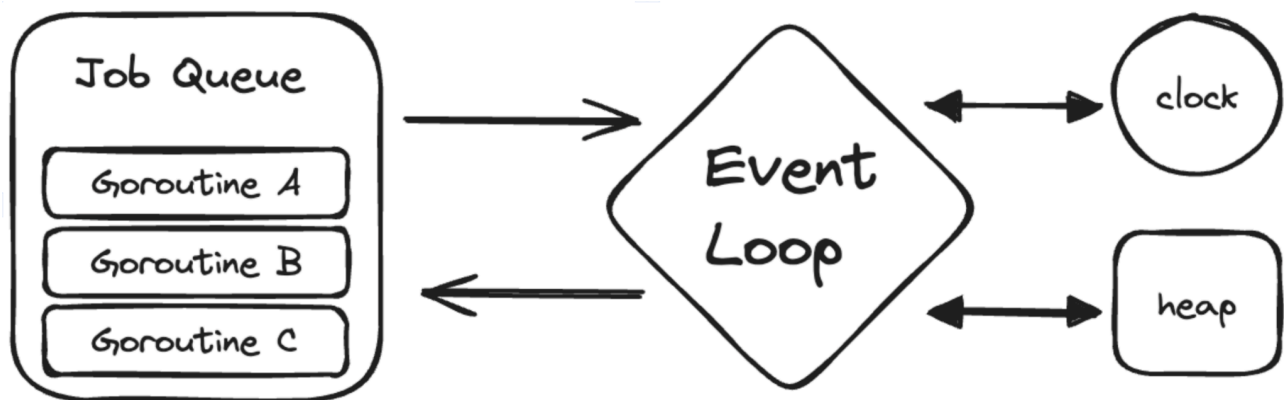
In order to preserve the operand stack, runtime stack of the caller, special frames are inserted on function calls. These introduce addresses of operand stack, runtime stack and the PC

addresses that the VM must restore once the function call is complete. A similar mechanism exists for `for` loops (i.e. on `break`, `continue`).

Suitable diagrams that illustrate this behaviour can be seen from slide 16 - 19 [here](#).

Goroutine event loop

We used an event loop architecture to allow for progress on multiple goroutines at the same time on a single-threaded browser environment. The event loop controls the execution of compiled bytecode by referring to the PC and stack referred to by each goroutine. It will access the clock to interrupt execution (when the time slice elapses) and enforce fair computing time between goroutines, and permits modification and reading from the heap.



A key point to note is that to avoid naive busy-waiting, our concurrency primitives make use of a `yield` sysCall that terminates an executing job's time slice and returns it to the queue immediately. This allows a blocked thread to avoid busy-waiting and consuming CPU time, and instead allows other threads to execute (presumably eventually unblocking the original thread).

Memory Management

Heap Implementation

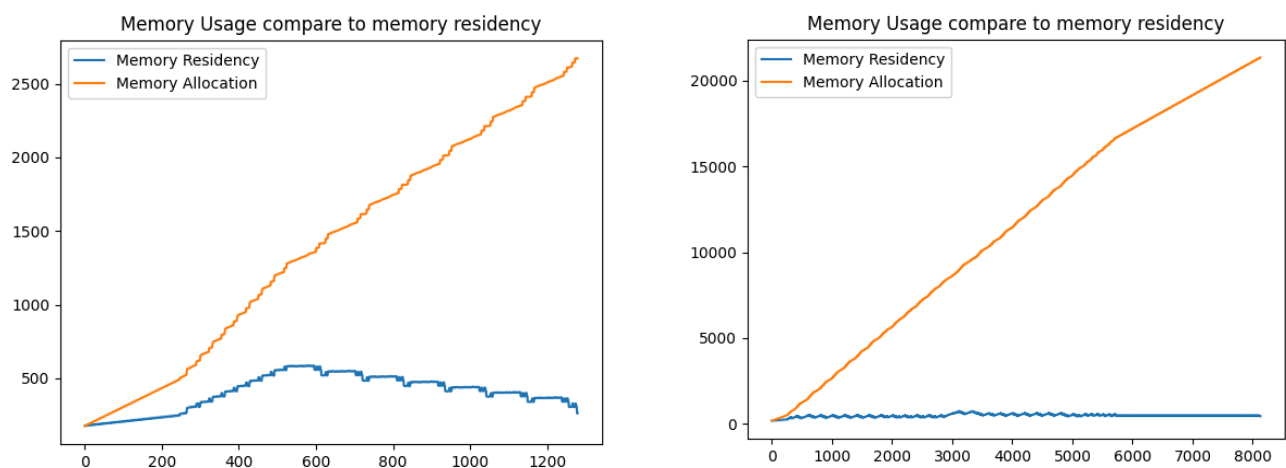
Our heap implementation operates on an `ArrayBuffer`, which is divided into fixed-size nodes. The fixed-sized nodes are categorised into 4 types: `bool`, `str`, `int` and binary pointers (`BinPtr`).

Notably, `BinPtr` contains 2 pointers to its child, here onwards will be called `child1` and `child2`, and is used extensively in our language implementation. To the user, this is used to

represent the pointer `*T` for any type, and only used `child1` with `child2` empty. To us, the developer, this construct works similar to how `pair` is utilised in Source: Complex data structures such as lists, queues, stacks, and function pointers are all constructed using `BinPtr`. This also allows us to use the heap as the storage space for runtime stacks (which encompass environment information) and operand stacks.

Garbage Collection

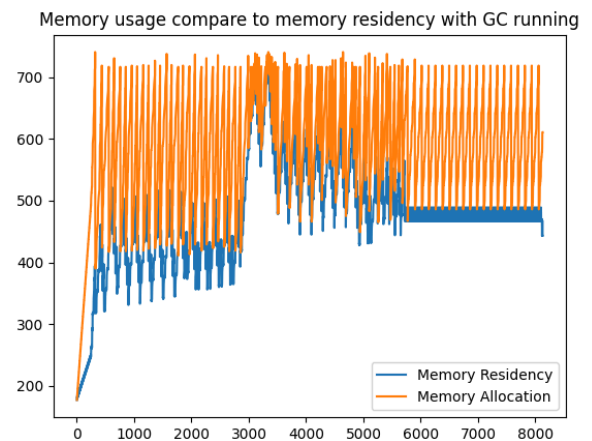
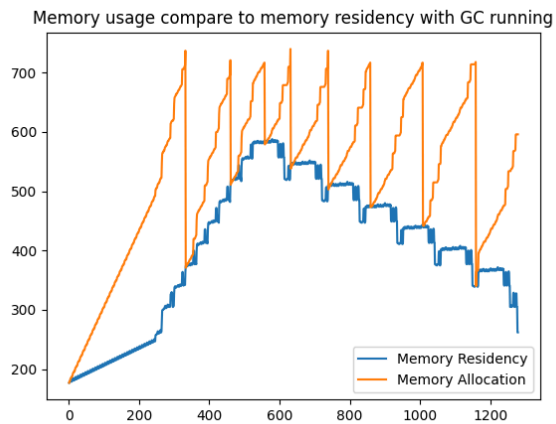
Our Virtual Machine implementation features mostly immutable heap nodes. These objects generate new objects upon modification rather than altering the original object. This approach has greatly helped us with the development of concurrency features, allowing us to effortlessly separate memory objects between different threads upon new thread creation. However, it presents an interesting challenge in that it results in a significant number of temporary object creations and a notably lower memory residency compared to the allocated memory.



Graph1: Single-threaded test case (left), Multi-threaded test case(right), running without Garbage Collection.

Due to how we allocate memory, we've opted for the stop-and-copy scheme for our Garbage Collection method. Specifically, we've selected Cheney's algorithm. This choice gives us a highly efficient Garbage Collector with space complexity of $O(1)$ and time complexity proportional to the number of live nodes.

For the reader's interest, our algorithm implementation can be found in the code repository [here](#).



Graph2: Single-threaded test case (left), Multi-threaded test case(right), running with Garbage Collection.

Concurrency Primitives

Below we list all the concurrency primitive API that we have developed for our implementation:

Mutex API:

```
mutexInit() *int
mutexLock(mutex *int)
mutexUnlock(mutex *int)
```

Semaphore API:

```
boundedSemInit(upperBound, initialValue int) *int
boundedSemPost(sem *int)
boundedSemWait(sem *int)
```

WaitGroup API:

```
wgInit() *int
wgAdd(wg *int)
wgDone(wg *int)
wgWait(wg *int)
```

Channel API:

```
chanInit(capacity int) *int
chanSend(ch *int, val int)
chanRecv(ch *int) int
```

Notable Concurrency Implementation Details

Yield SysCall is an atomic instruction for the current running thread to signal it wants to relinquish its CPU control (implementation-wise, this means setting the time slice value of the current thread to 0).

Done SysCall is another atomic instruction used by the current running thread to terminate its execution. If the current running thread is the main thread, we terminate all threads (following Golang standard behaviour) .

Semaphore & Mutex: Both semaphore and mutex utilise the TestAndSet and SysCall('yield') functions in their implementation to avoid naive busy-waiting. Our semaphore is explicitly an upper-bound semaphore, meaning that the boundedSemPost` API cannot exceed the upper bound value.

Channel:

Channel is a composite type object created using `BinPtr` that include:

- A queue to simulate the first in first out send/receive order for message
- A mutex that need to be acquired for all operations
- Two semaphore, one for empty slot to write, one for full slot that can be read

We used the same logic flow for both unbuffered and buffered version of channel (unbuffered mean channel with capacity 0 in our implementation):

- Capacity of the queue is set to channel's capacity + 1
- On read, same logic for both versions: block if there is nothing to read
- On write, add to the back of the queue (unbuffered version's queue would have capacity 1). If queue is full (size == capacity), both version would have to wait until queue size is at most capacity - 1:
 - In the buffered version, this mimics the scenario of queue full -> block send complete until someone has read.
 - In unbuffered version, this mimic the synchronous send and receive behaviour (we can only unlock if someone read what we have just put in)

Repository

Project Sources & Libraries

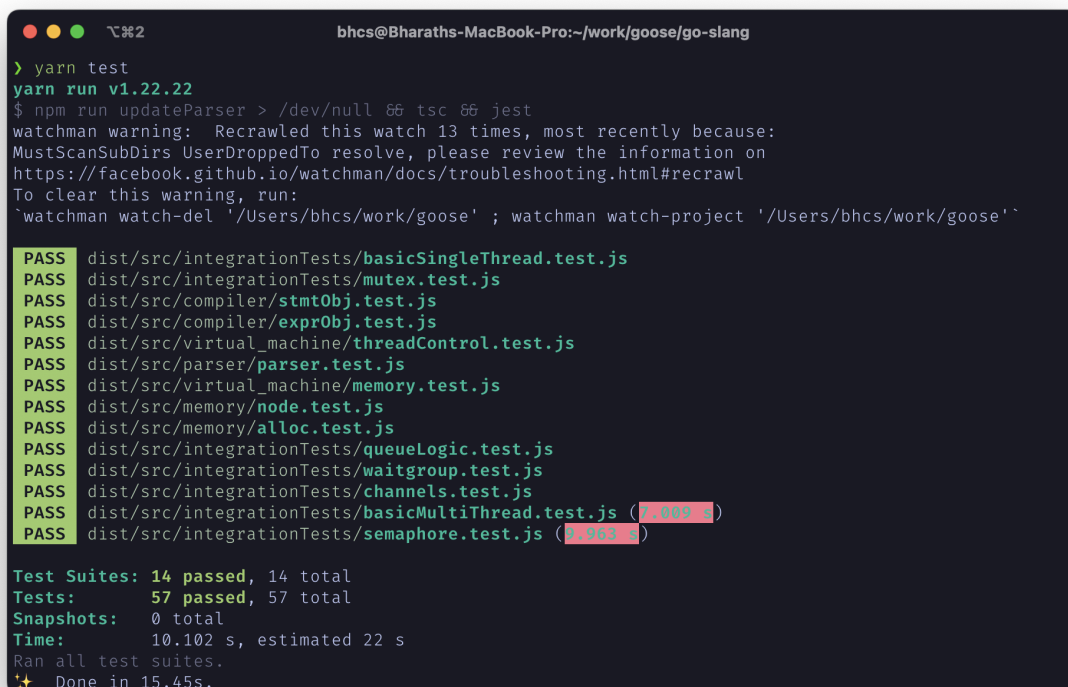
- [Peggyls](#): a left-recursive parser generator that takes in a PEG file (in our parser). See [./go-slang/src/parser/peg](#). (note: our grammar file is heavily adapted from attributed sources)

- TSC: strong, well-typed objects are used in every step of parsing, compilation and execution to give confidence on the lack of mal-formed inputs/outputs and coverage of all execution forms.
- Jest: Key testing library that provides expressive syntax to write BDD tests.

Install, Build & Deploy

The project code is available on a public GitHub repository at: <https://github.com/jushg/goose>. Users can build and run Goose programs using the instructions in the [README.md](#).

Test Cases



```
bhcs@Bharaths-MacBook-Pro:~/work/goose/go-slang
> yarn test
yarn run v1.22.22
$ npm run updateParser && /dev/null && tsc && jest
watchman warning: Recrawled this watch 13 times, most recently because:
MustScanSubDirs UserDroppedTo resolve, please review the information on
https://facebook.github.io/watchman/docs/troubleshooting.html#recrawl
To clear this warning, run:
`watchman watch-del '/Users/bhcs/work/goose' ; watchman watch-project '/Users/bhcs/work/goose'`

PASS dist/src/integrationTests/basicSingleThread.test.js
PASS dist/src/integrationTests/mutex.test.js
PASS dist/src/compiler/stmtObj.test.js
PASS dist/src/compiler/exprObj.test.js
PASS dist/src/virtual_machine/threadControl.test.js
PASS dist/src/parser/parser.test.js
PASS dist/src/virtual_machine/memory.test.js
PASS dist/src/memory/node.test.js
PASS dist/src/memory/alloc.test.js
PASS dist/src/integrationTests/queueLogic.test.js
PASS dist/src/integrationTests/waitgroup.test.js
PASS dist/src/integrationTests/channels.test.js
PASS dist/src/integrationTests/basicMultiThread.test.js (7.009 s)
PASS dist/src/integrationTests/semaphore.test.js (9.963 s)

Test Suites: 14 passed, 14 total
Tests: 57 passed, 57 total
Snapshots: 0 total
Time: 10.102 s, estimated 22 s
Ran all test suites.
🌟 Done in 15.45s.
```

The test cases screenshot here cover all the major functionalities of the language, including both sequential constructs as well as concurrency features. All of our integration test cases, which can be implemented as separate Goose programs to verify the correctness of the compiler and VM functionalities, can be found [here](#).