Implementation of a Garbage-Collected LLVM Front End

LIEUWE VAN DEN BERG, University of Twente, The Netherlands

Garbage collection is an important component of modern programming languages. This paper explores the design trade-offs of different garbage collection strategies such as mark-sweep and reference counting through a comparative analysis of the garbage collectors of Go, Java, and Python. Building on this research, a minimal, statically typed programming language with a C++ mark-sweep garbage collector is implemented. The language is compiled to LLVM intermediate representation using a front end written in Go, using TinyGo's LLVM bindings.

Additional Key Words and Phrases: LLVM, garbage collection, compiler, Go

1 INTRODUCTION

Efficient memory management is key to writing performant applications. Rust uses this fact by handing control of memory management to the programmer using its borrow checker, potentially increasing application performance but also making the language more difficult to use [7]. Many popular programming languages instead make use of a garbage collector (often referred to hereafter as GC), which abstracts away memory management by automatically freeing unused objects in memory. Such languages include JavaScript, Python, Java and Go, among others. This allows the programmer to spend more time on their own application rather than memory management. However, there are many different approaches to implementing garbage collection and each comes with its own trade-offs in terms of performance, pause time (commonly referred to as "stop-the-world"), memory overhead, and complexity. [17]

There are several garbage collection methods, most importantly mark-sweep (tracing) and reference counting. In mark-sweep, the program is halted while the GC work is performed. In the first stage, mark, the GC starts from a set of roots, which are global and local variables and function parameters. All objects reachable by following pointers from the roots are recursively marked. In the sweep stage, the heap is scanned and all objects that are not marked are freed. On the other hand, a reference counting GC keeps track of the number of references to an object. If an object's counter becomes zero, the object is freed. This way, memory is freed as soon as an object becomes unused. [17]

These methods have shortcomings and advantages that need to be considered when designing a GC. This paper identifies these trade-offs.

Also in this paper, a minimal garbage-collected programming language is implemented. To start, a simple custom language compiler (front end) is written in Go that compiles the language into LLVM intermediate representation (IR). The IR is then handed to LLVM which can provide optimization passes and compiles it to the native code of any of the supported back ends such as x86. [11] The front end supports basic language features like conditionals and variables. Knowledge from the previous GC research part helps design the GC implemented in the language. The Git repository for this front end can be found on GitHub [20].

This leads to this paper's research goals:

- **RG1**: Identifying trade-offs between garbage collector implementations in various languages. The languages assessed here are Go, Java and Python, as they cover a variety of GC features and reflect different use cases which grant an additional insight into those GC features.
- **RG2**: Creating a custom LLVM front end and implementing a minimal garbage collector. Analysis of existing GCs in the previous research goal helps decide the most simple way to implement a working GC while still considering trade-offs when sacrificing performance for simplicity.

2 RELATED WORK

This section will briefly show related work in LLVM GCs and Go-based LLVM front ends.

Go has its own compiler, a GCC compiler and also an LLVM compiler, gollvm. However, the latter is still in development, does not have any releases, and the language front end is written in C++, not in Go. [8]

The 2022 master thesis by Ramberg [15] successfully used TinyGo's LLVM bindings to generate the LLVM IR. Another package was also used, llir/llvm, but was later removed because it does not integrate into the LLVM toolchain but rather only generates the IR. It has better types, but is now unmaintained and they recommend using TinyGo's bindings [12].

Searching for 'tinygo.org/x/go-llvm language:"Go Module"', the Go import path used for TinyGo's LLVM bindings combined with filtering only for go.mod files[1], on GitHub, returns little results: 66 at time of writing. Most are forks of the TinyGo repository itself. However, some results are similar to what this paper implements: Candice[6] and Telia[18] are both compilers for custom languages written in Go, but both do not feature a GC.

At the Go Conference 2017 Spring, an LLVM-based front end called gocaml [16] was presented. This front end is a subset of the OCaml language. The entire compiler is written in Go, except for the runtime which is written in C. This is because the GC it uses is the

43rd Twente Student Conference on IT, July 4, 2025, Enschede, The Netherlands

[1] go.mod is the file that specifies, among other things, the imports required by your own module that are not part of the standard library.

Boehm GC, a C mark-sweep GC [5]. Furthermore, gocaml uses the official Go LLVM bindings.

3 EXISTING GARBAGE COLLECTORS

To answer RG1, the GC implementations of Go, Java and Python will be compared. The goal is to identify how different language runtimes approach GC, and the trade-offs involved in terms of performance, pause time, memory overhead, and complexity.

3.1 Go

Go's GC is not specified in the language specification. It is up to the language's implementation to decide on a GC strategy.

The default Go compiler uses a tri-color mark-sweep GC that is mostly concurrent. The tri-color marking runs concurrently in parallel with the application and allows a write barrier to be implemented that marks objects gray (to be scanned) when a pointer in the heap is modified during marking. This way, memory writes during the scan are tracked by the GC and the scan is able to be performed concurrently. [10] The aim of the GC is to make sure that application pauses are as short as possible. Go is often used to write highly concurrent applications such as web servers that expect low latency, so it is designed to sacrifice some throughput for lower latency by running almost entirely concurrently. Only brief pauses are done when necessary: to identify roots on the stack, and when marking is complete. The sweep phase is also run concurrently. After sweeping, the GC is turned off until the next time it is run. [9]

The GC is non-moving: it does not copy memory [1]. There can be heap fragmentation, but it is largely solved by using special heap allocation methods detailed in the Go source code [3]:

> The main allocator works in runs of pages. Small allocation sizes (up to and including 32 kB) are rounded to one of about 70 size classes, each of which has its own free set of objects of exactly that size. [...] Round the size up to one of the small size classes and look in the corresponding mspan in this P's mcache. Scan the mspan's free bitmap to find a free slot. If there is a free slot, allocate it.

This helps keep the GC simpler and keep pause times low (no pointers need to be updated during a compacting phase), but can impact memory usage.

A formula for target heap size is used to decide roughly how often the GC will run. A compiler parameter `GOGC` is used to tune the GC. It is a component of the target heap size formula:

$$\textit{Target heap size} = \textit{Live heap} + (\textit{Live heap} + \textit{GC roots}) * \textit{GOGC}/100$$

Clearly, a `GOGC` value of 100 will allow up to approximately twice the current live heap size (size of objects with references to them) before the GC should free memory. Doubling the `GOGC` doubles the heap size overhead of the next target but decreases the amount of times the GC is run, trading increased memory usage for less CPU time.

3.2 Java

Since Java comes in different distributions, this section will discuss Oracle's Java VM distribution. OpenJDK has a similar implementation, but only documentation from Oracle [13] will be used.

Java features four GC implementations: Serial, Parallel, G1, and ZGC, all of which are generational and use a marking-based strategy. The VM selects the best one for the hardware and operating system configuration. G1 is selected by default on most configurations.

A generational GC performs its work incrementally in generations. For example, G1 uses young and old generations. When an object is allocated, it is allocated into a group of the youngest generation. After surviving a GC cycle, it can be promoted to an older generation depending on internal and configurable parameters. Older generations are scanned less often. Newer objects may be more likely to need GC, while older, longer surviving objects do not. This is called the *weak generational hypothesis*.

In addition to being generational, G1 is parallel, mostly concurrent and performs heap compacting. It is designed to scale and provide a balanced approach to garbage collection with small, uniform pauses while still having high throughput. Most of the marking work is done concurrently in parallel with the application that is being run, however some stages like heap compacting (where object references need to be updated) and memory reclaiming cause pauses. G1 tries to keep these pauses short and uniform by doing the work incrementally.

Concurrent marking and space reclamation of old generation regions is only started once the old generation reaches a certain size: the Initiating Heap Occupancy threshold. There are many parameters for tuning the GC behaviour.

The newer ZGC is made to be more concurrent than G1, with similar low pause times as Go's GC. However, it is not the default GC.

3.3 Python

Similar to Go, there is no official language specification for the GC. The commonly used CPython compiler's GC [2] will be discussed.

Python uses two garbage collection strategies at the same time: reference counting and a cyclic GC. This is because the reference counting strategy cannot resolve cycles, where a container object contains a reference to itself. Most objects are still freed by reference counting however. It adds some overhead from managing the counters and freeing the memory, but otherwise introduces no pauses from running a GC function and on its own is cheaper than traversing an object tree with mark-sweep.

To resolve cyclic references, a generational cyclic GC is also used. The GC only handles container objects that can have cyclic references, including lists, classes, and various other container objects. It maintains a list of possibly cyclic objects and when the GC is run, it performs multiple scanning passes to identify the unreachable objects. It resembles a mark-sweep GC, but uses the reference counter fields on objects to find unreachable objects instead. It maintains

two lists: reachable and tentatively unreachable. By traversing all objects in the reachable list, objects with no outside references are moved to the tentatively unreachable list. Objects may be moved back and forth in the two lists if an object is discovered that is reachable and references objects in the unreachable list. Objects that are moved into the reachable list are again scanned. Once all objects are scanned, the objects in the tentatively unreachable list can be freed.

The cyclic GC is run when the number of allocations minus the number of deallocations exceeds a configurable threshold. It is 700 by default. Since Python uses the global interpreter lock to only execute bytecode on one thread at a time, the GC is not concurrent. It blocks Python threads from executing.

3.4 Comparison

Go sacrifices some memory efficiency for speed by not using the optimal but slower heap compacting to reduce heap fragmentation. However, it makes the GC very concurrent as it has only a small amount of very short pauses. Java's default G1 GC works similarly but has longer pauses to perform compacting. Python's GC uses a different approach in the form of reference counting to attempt to prevent pauses at all and provide a very simple GC system, but because of cyclic references becomes complicated and requires another mark-sweep-like GC to prevent memory leaks.

Since mark-sweep does not have inherent issues, is widely used, and can be extended to run concurrently, it is a good starting point for garbage collection. However, it does have performance overhead by taking away more CPU resources than plain reference counting would.

The simplest form of a GC would be mark-sweep with no heap compacting or concurrency. This means the GC will cause heap fragmentation and always pause when it is run, but will reliably collect unused objects.

4 LANGUAGE IMPLEMENTATION

The language compiler is written in Go. Go has good C integration, which makes the bindings only a very small layer on top of the LLVM C bindings. Most functions in the Go bindings directly call C functions. As used in the work by Ramberg [15], the clearest choice for Go LLVM bindings are TinyGo's bindings. This is because the official LLVM bindings for Go were removed from the LLVM project [14]. TinyGo's bindings are a fork of the previously official bindings [19]. Another Go library llir/llvm exists, also used by Ramberg, that provides better and more strict types for generating LLVM IR, however that is now unmaintained and they also recommend using TinyGo's bindings [12].

The implementation consists of a lexer, a parser, and LLVM IR code generation:

(1) The lexer is powered by the standard library `text/scanner` package. This package reads the input token stream and handles most complex tokenization: it handles identifiers, strings, comments, integers, and has options for floats and characters. When tokens are scanned, identifiers are converted to keywords if applicable but otherwise the scanner output is copied. The lexer returns an array of tokens.

(2) The parser converts the array of tokens to an AST (Abstract Syntax Tree). To better understand parsing and AST construction, a custom parser is implemented instead of using ANTLR or yacc or another parser code generator. The parser is a predictive recursive descent parser. It bases its output only on the current and next tokens, rather than backtracking, and uses recursive functions based on the language grammar. Parsing is done multiple times but appended to a single AST: first, for garbage collection functions, then for standard linked functions, and finally for the program source code. This enables GC and standard functions to already be accessible by the program that is being compiled.

(3) Finally, the code generation is performed using the LLVM bindings. The AST is walked depth-first to produce valid IR. This IR is output to a file which is then given as an input argument to Clang++ (as the GC is written in C++) to link and compile the binary.

4.1 Language design

As there is no specific goal for the syntax of the language, this paper implements one that is easy and natural to read. You should not require knowledge of the language to be able to understand it. The language is statically typed. Keywords are not abbreviated (`function` instead of `func`). There are no type aliases. The syntax is inspired mostly by Go's syntax, which in turn is C-like, at least in style.

To support the GC, some features need to be implemented as a language syntax feature or by the compiler:

- There must be a way to allocate memory on the heap. Some languages decide stack or heap allocation automatically, while some use a keyword such as `new`.
- The language must have the ability to call functions with arguments.
- To call the GC's functions, the language must support external functions.
- Clang++ must be able to match function prototypes in order to link external functions, therefore at least the types used by the GC functions need to be able to be emitted by the compiler.
- The language must support pointers. For example, the program roots need to be registered, which are pointers.

4.2 Comments

Comments are the same as in Go, which in turn are the same as in C.

```
// Single-line comment
uint64 two = 2 // Can be placed at the end of a
    ↪ line
```

```
/* Multi-line comment
uint64 two = 2 <- this code is not executed
*/
```

4.3 Types

The following types are supported: `int32`, `int64`, `uint64`, `string`, `void`. The void type is not strictly needed in the language, it is used under the hood only and void is inferred from a missing function return type. However, it is available as a word and can be used where appropriate. Strings are defined using double quotes (") around the string. They automatically get null terminated. All types can be defined as pointers by prepending a `*`, like `*int32` and `**int32`.

4.4 Expressions

Expressions produce a value. These are used any time a value is expected and may be any combination of smaller expressions, such as numbers, variables and function calls. The language supports addition (+), subtraction (-), multiplication (*), integer division (/), and the remainder operation (%). Expressions wrapped in round brackets are evaluated first before surrounding expressions. Otherwise, normal mathematical order of operations is used.

Strings may be expressions but are not usable with mathematical operators.

4.5 Equality

Two expressions can be compared for equality using a double = operator. The boolean value that can be returned is an integer, 1 for true and 0 for false. Equality is also an expression.

```
4 % 2 == 0 // 1
5 % 2 == 0 // 0
```

4.6 Variables

Variables are defined by starting a statement with a type, followed by a variable name, an assignment operator (=) and an expression.

```
string greeting = "hi"
uint64 two = 2
```

The language uses a stacked scope, which means variables can be shadowed:

```
uint64 two = 2
function main() int64 {
    printf("%d\n", two) // 2
    uint64 two = 5
    printf("%d\n", two) // 5
    return 0
}
```

4.6.1 Pointers. Addresses of variables can be obtained by prepending the & operator to its name. A pointer variable can be dereferenced to get the value at the address that the pointer points to using the * operator. Pointer types may point to other variables of a pointer type.

```
int64 two = 2
*int32 ptr = &two
**int32 doublePtr = &ptr
***int32 triplePtr = &doublePtr
printf("%x\n", *ptr) // 2
printf("%x\n", **doublePtr) // 2
printf("%x\n", ***triplePtr) // 2
```

4.7 Structs

Custom struct types can be defined and used in place of regular types. Structs first need to be defined, anonymous structs do not exist. A struct is defined with the `struct` keyword followed by the struct name. The fields are then defined in the form of `name type` separated by a comma.

```
struct Person {
    Name string,
    Age  int32
}
```

Once defined, a struct name can be used as a type.

A struct is instantiated as follows:

```
{"david", 25}
```

No struct name or field names are required, this is inferred from the context. For example, when creating a variable, the variable type is the struct name. This gives the compiler the information it needs to set the struct fields.

Because of the way structs are implemented, it is not currently possible to pass structs to functions.

A variable can be of type Person by using the struct's name as the type.

```
Person david = {"david", 25}
```

4.7.1 Field access. Values of fields of an instance of a struct can be accessed with a period, followed by the field name. Fields can not be modified this way, a new instance of the struct must be assigned instead.

```
printf("Hello %s (%d)\n", david.Name, david.Age)
```

4.7.2 Example. This is an example of a full program combining the previous elements.

```
struct Person {
    Name string,
    Age int32
}

function main() int64 {
    Person david = {"david", 25}
    printf("Hello %s (%d)\n", david.Name, david.
        ↪ Age) // Hello david (25)

    return 0
}
```

4.8 Functions

Functions start with the keyword `function` followed by the function name. Function parameters are comma separated in the form of `name type` between the round brackets. The return type of the function is given before the opening curly bracket and may be omitted, in which case the void type is inferred. The function body is then put between curly brackets and is composed of statements.

```
function add(a int64, b int64) int64 {
    return a+b
}
```

The entry point for the program is a function called *main*. The exit code for the program is the returned integer.

```
function main() uint64 {
    // Statements
    return 0
}
```

Nested functions are syntactically allowed, but closures are not supported. There is no access to any parent scope, only to the global scope. If a global scope variable is shadowed, the nested function will try to access the new variable from the parent scope. The compiler does not currently support this.

4.8.1 Return. Functions must always end with a `return` statement, even if the functions are of void type and even if all code paths already have a return statement. There is no pass performed over the AST to check if all paths return a value already, so the final return statement might be unreachable.

4.8.2 Calling functions. Functions are called by name and then listing all parameters between round brackets. A function call is a statement and an expression and can be used as a parameter as long as it returns a value.

```
printf("hello! 5+7=%d \n", add(5, 7)) // hello!
    ↪ 5+7=12
```

Variables are passed by value. The address operator can be used to pass by reference. The value of a pointer variable is the address, pointers are not automatically dereferenced based on the context (e.g. for print statements).

4.8.3 External functions. External functions are functions not defined in the program itself, only the function prototype is given. Clang++ will try to link the external function if it can find a function (for example standard C functions) that matches the given prototype. The exact name, arguments and return type are important. A function like `printf` can be linked this way.

```
external function printf(format string, args
    ↪ ...) int32
```

The `...` type specifies that the function is vararg (variable arguments). `format` is a required parameter but after that any parameters are allowed. Vararg arguments are named but do not have an underlying type. They are not supported for use in functions and are solely supposed to be used to link external functions that have a vararg argument such as printf.

4.9 If-else conditions

Conditional logic is similar to any other language by using the `if` and `else` keywords. However, a boolean type does not exist. Instead, 0 is used for false and any other integer value for true. The equality operator (==) creates a 0 when the equality is false, and a 1 when the equality is true, and can thus be used in if-statements. If an expression evaluates to 0, the body of the `if` statement is not evaluated and instead an optional else statement will be evaluated. Like Go, there are no brackets needed around the condition.

```
int32 number = 5
if number-(number/2+number/2) {
    printf("uneven\n")
} else {
    printf("even\n")
}
```

Else-statements can be chained directly into new if statements to prevent deep nesting of if-else chains. There is no special keyword for this, instead the `else` keyword is followed immediately by a new if-statement.

```
int64 number = 13
if number % 3 == 0 {
    printf("divisible by 3\n")
} else if number % 3 == 1 {
    printf("1 too many!\n") // 1 too many!
} else {
    printf("1 too little!\n")
}
```

5 GARBAGE COLLECTION

The garbage collector is implemented in C++. It is a very simple mark-sweep collector, as per the conclusion of RG1. C++ is chosen for being low level (fast and it has no GC) but still having advanced data structures like maps and vectors.

The collector keeps a hash map of heap objects and a vector of root pointers. Normally, languages add a small header to heap objects, but for simplicity the map is used. This will add some overhead for the hash map's data structure and needing to access the value through the map first rather than directly from memory. However, its lookups are $O(1)$ if there are no collisions, and should therefore be fast enough for this minimal implementation.

In the mark stage, the root pointers are iterated over and the respective entry in the heap object map is retrieved. The object is marked, and based on its type information any struct fields are recursively marked. Otherwise, the address (without any field offsets) is recursively marked. In the sweep stage, the map of heap objects is iterated over and any unmarked heap objects are deleted from the map and their address freed.

5.1 GC interface

Various functions are exposed by the C++ code that are required by the language code generation to implement the garbage collector.

5.1.1 gc_register_type. Registers a struct type (non-primitive, so also arrays in case they are implemented in future work). Its parameters are the size in bytes to allocate on the heap, the number of fields the struct has and the field offsets. It returns a pointer to a GCType object. The language does not need to know about this object. Only the pointer is used to later identify the type with `gc_alloc` calls.

```
struct GCType {
    uint64_t size_bytes;
    uint64_t num_fields;
    uint64_t* field_offsets;
};
```

5.1.2 gc_alloc. Allocates memory on the heap. Its parameter is a pointer to a GCType returned by `gc_register_type` and a boolean `is_pointer`. The size passed to malloc is the `size_bytes` field of the GCType that is passed. For non-struct values, 0 can be passed as the type, in which case it 8 bytes will be allocated. The pointer to the type field will also be a null pointer in this case. The `is_pointer` field is a quick way to identify pointers without making explicit GCTypes for them as that is not supported. Similar types for smaller byte sizes than 8 bytes are also not supported. It also inserts a HeapObject value to the heap objects map, keyed by the pointer returned by malloc.

```
struct HeapObject {
    GCType* type;
    bool is_pointer;
    bool marked;
};
```

5.1.3 gc_add_root. Adds a new root to the vector of root pointers. Its parameter is a pointer returned by `gc_alloc`. The root pointers are global variables of a pointer type, local variables, and function parameters.

5.1.4 gc_pop_roots. Removes roots. Since n amount of roots are allocated in a function, n must also be popped from the roots vector. This call is inserted before the return statement of a function.

5.1.5 gc. Completes one mark-sweep cycle. A heuristic is added to `gc_alloc` to run the garbage collection every 1000 calls. This is arbitrary and in sophisticated languages is typically chosen based on a formula or other metric. For example, Go's target heap size or Java's Initiating Heap Occupancy.

5.2 Example with pointers

In the following example, a function `returnTwo` is created that allocates a variable `two`. When the function returns, its roots (the `two` variable) are popped. This means it is no longer reachable and will be garbage collected. However, the return value, a pointer to `two`, is stored in a variable called `twoPointer`. Now, `two` again has a valid reference to it and will not be garbage collected. Running the GC at this point will not free any heap objects.

Next, `twoPointer` is set to 0. Now `two` does not have any reference to it and will be garbage collected.

```
function returnTwo() *int32 {
    int32 two = 2
    return &two
}

function main() int64 {
    *int32 twoPointer = returnTwo()

    gc()
    printf("%d\n", *twoPointer) // 2
    twoPointer = 0

    gc()

    return 0
}
```

The program output is as follows. Lines starting with `GC` are emitted from the GC. Comments are added with // but do not mean anything in this textual console output. They are just for annotation purposes.

```
GC: Root added 0x5cb8b2945380 // ...80 = two
```

```
GC: Popping 1 roots, now: 1 // returnTwo
    ↪ function exits
GC: Root added 0x5cb8b2944eb0 // ...b0 =
    ↪ twoPointer
GC: GC starting. Heap: 2, roots: 1
GC: 0x5cb8b2945380: 1 // two is found through
    ↪ twoPointer and marked
GC: 0x5cb8b2944eb0: 1
GC: Bytes freed: 0
GC: GC done. Heap: 2
2 // the printf call
GC: GC starting. Heap: 2, roots: 1
GC: 0x5cb8b2945380: 0 // two is no longer marked
    ↪  as twoPointer is a null pointer now
GC: 0x5cb8b2944eb0: 1
GC: Bytes freed: 8
GC: GC done. Heap: 1 // 8 bytes are freed and 1
    ↪ object remains on the heap, twoPointer
```

The `valgrind` tool, used for memory leak detection, among other things, indeed shows that only 8 bytes remain on the heap when the program is done. This is because after main exits, there is not another mark-sweep run and twoPointer remains on the heap. It is cleared by the operating system.

```
in use at exit: 8 bytes in 1 blocks
```

6 DISCUSSION

The language currently has some drawbacks that need to be addressed in the future.

6.1 Type safety

Types are often not checked by the parser. There are two significant known type issues:

(1) Apart from some issues that are caught by the parser, the main type checker is Clang++. This means most type errors in the generated IR are found, but some can slip through, for example with signed numbers. LLVM does not have unsigned integer types. Instead, some operations have signed and unsigned versions, such as division and the remainder. Clang++ therefore does not check for signed and unsigned mismatches.

(2) Pointers in LLVM are by default opaque as of version LLVM 15 and no option for typed pointers exists as of LLVM 17 [4]. This means LLVM does not carry information about underlying types of pointer variables. Since the parser does not check type correctness for assigning variables, the following code is allowed and works but should not be allowed for a type safe language.

```
int64 two = 2
*int32 ptr = &two
```

This example will work, however for numbers larger than 2^{31} (int32) or for other types (such as structs) this can cause issues with pointers pointing to memory that is not of the correct type. This also suggests the removal of the void type as a word in the language as *void-type variables do not have a known type. To safely cast from *void or a special 'any' type, the runtime would need to support tracking the type of every variable.

6.2 Primitive data types

All primitive data types are allocated 8 bytes (64 bits) in memory. That means variables of type int32 are allocated unnecessary space. Smaller types that can be implemented in the future, such as 16-bit or 8-bit integers, will have an even larger relative overhead. This can be solved by generating standard GCType instances for standard used byte sizes, such as 1, 2, 4, and 8, and passing those to `gc_alloc` instead. This is not currently implemented. Pointers should also get this treatment as it would make the `is_pointer` parameter of `gc_alloc` obsolete.

6.3 Heap allocation

For simplicity, all variables are allocated on the heap. However, stack allocations can be cheaper than heap allocations. The language compiler does not perform escape detection, which would be necessary to identify variables that need to be allocated on the heap. All other variables can be allocated on the stack. Additionally, there is more work for the garbage collector as it needs to collect a large amount of memory that would normally go to the stack from variables that do not escape the function they were allocated in.

6.4 Heap fragmentation

When memory is freed, it leaves gaps in the heap which can be hard to fill (which Go attempts to do). That is why garbage collectors typically move objects on the heap via various means such as copying them to new regions or compacting them into the same region. The empty regions can then be reclaimed by the operating system or reused more easily, especially when large objects are allocated. This is an important feature of garbage collectors but this implementation does not perform any copying or compacting. This leads to increased memory usage as a result of the gaps.

6.5 Object headers

Instead of keeping metadata (the type and mark) about objects in the HeapObject map, it should be added as headers before the heap objects themselves. This reduces any overhead from keeping this data in the map instead.

7 CONCLUSION

This paper implemented a minimal programming language with LLVM and a C++ mark-and-sweep garbage collector. First, three GCs of existing programming languages were compared to answer RG1. Then, a simple mark-sweep GC was identified as a minimal implementation for RG2. The GC successfully identifies and frees objects from the heap when they are no longer referenced by the program.

The implemented language is statically typed (with limited type checking) and supports variables, functions, conditionals, and structs. Various data types and basic mathematical operations are supported. The language also allows pointers and linking external functions.

7.1 Future work

In addition to shortcomings mentioned in the discussion, future work can include:

- Arrays. While a (doubly) linked list can be made in the language, or even arrays themselves through pointer arithmetic (which the language does not prevent), built-in arrays with index support are a must for modern languages.
- Proper runtime string support. Currently strings can only be static and made at compile time. Allowing string operations such as concatenation brings many more possibilities to programs, while keeping a simple interface.
- Struct field modification. Struct fields cannot be modified but this is an essential feature of working with structs and should be added.

8 AI STATEMENT

ChatGPT was used for helping debug LLVM issues with the bindings. Although it understood the bindings extremely poorly, it still had a general idea of LLVM to an extend. For example, it helped find how to initialize and access struct fields and what certain function parameters had to be (everything in the bindings is just a generic Value struct). ChatGPT or other AI was not used for writing this paper or actually doing programming work. No code has been copied from ChatGPT.

REFERENCES

[1] [n. d.]. A Guide to the Go Garbage Collector. https://go.dev/doc/gc-guide

[2] [n. d.]. CPython Garbage collector design. https://github.com/python/cpython/blob/v3.14.0b3/InternalDocs/garbage_collector.md

[3] [n. d.]. Go 1.24.4 runtime/malloc.go source code. https://github.com/golang/go/blob/go1.24.4/src/runtime/malloc.go

[4] [n. d.]. Opaque Pointers. https://llvm.org/docs/OpaquePointers.html

[5] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage collection in an uncooperative environment. *Software: Practice and Experience* 18, 9 (1988), 807–820. https://doi.org/10.1002/spe.4380180902 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380180902

[6] Candice Git [n. d.]. https://github.com/gabivlj/candice

[7] Michael Coblenz, Michelle L. Mazurek, and Michael Hicks. 2022. Garbage collection makes rust easier to use: a randomized controlled trial of the bronze garbage collector. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1021–1032. https://doi.org/10.1145/3510003.3510107

[8] gollvm Git [n. d.]. https://go.googlesource.com/gollvm/

[9] R. Hudson. 2015. Go GC: Latency Problem Solved. https://go.dev/talks/2015/go-gc.pdf

[10] R. Hudson. 2015. Go GC: Prioritizing low latency and simplicity. https://go.dev/blog/go15gc

[11] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[12] Robin ('mewmew'). 2024. https://github.com/llir/llvm/issues/240#issuecomment-2517453355

[13] Oracle. 2025. Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide, Release 24. https://docs.oracle.com/en/java/javase/24/gctuning/hotspot-virtual-machine-garbage-collection-tuning-guide.pdf

[14] Nikita Popov. 2022. [bindings] remove go bindings · LLVM/llvm-project@bc839b4. https://github.com/llvm/llvm-project/commit/bc839b4b4e27b6e979dd38bcde51436d64bb3699

[15] Håvard Ramberg. 2022. *Exploring compiler parallelism with Go.* Master's thesis. NTNU.

[16] rhysd. [n. d.]. rhysd/gocaml: Statically typed functional programming language implementation with GO and LLVM. https://github.com/rhysd/gocaml

[17] Eliot Moss Richard Jones, Antony Hosking. 2023. *The Garbage Collection Handbook: The Art of Automatic Memory Management* (2 ed.). CRC Press.

[18] Telia Git [n. d.]. https://github.com/HicaroD/Telia

[19] TinyGo LLVM bindings Git [n. d.]. https://github.com/tinygo-org/go-llvm

[20] Lieuwe van den Berg. [n. d.]. Language Compiler Implementation Git. https://github.com/lieuweberg/compiler