Mathematical Models of Variable Lifetimes

Makoto Emura

There are two ways to store variables in memory: on the stack or in the heap. The compiler and the runtime of high-level languages manage the lifetimes of variables using structures that are based on sets and graphs.

Set relations can model stack allocation for automatic variables. Automatic variables in a function have shorter lifespans than variables of the parent function in the call stack. In the Go example below, `main()` occupies the stack first, then the `getAnswer()` frame gets pushed onto the stack.

```
func main() {
        theQuestion := "What do you get if you multiply six by nine?"
        theAnswer := getAnswer(theQuestion)
        // result variable not needed anymore
        os.Exit(theAnswer)
}

func getAnswer(question string) int {
        result := 42
        return result
}
```

Figure 1 shows how the stack frames created in the example code can be modeled with a Venn diagram. The set S_0 contains all global variables, S_1 contains variables in `main()`, and S_2 contains variables in `getAnswer()`. If the program counter is in `main()` just after `getAnswer()` exits, variables within S_1 as well as in the subset S_0 are allocated on the stack, but variables of S_2 are popped off.

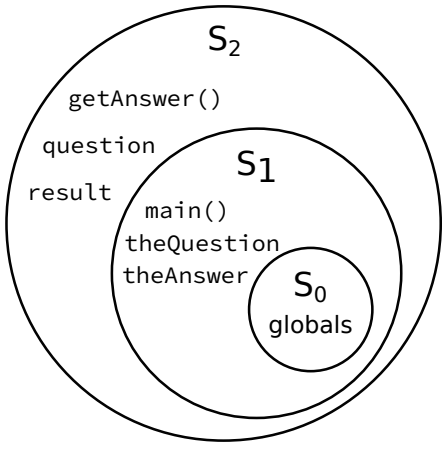

Figure 1: Venn diagram of the stack

Note that the string `theQuestion` is passed to `getAnswer()`, and global variables, if there were any, can be used in `getAnswer()`. Refer-

ences can be passed into functions and must outlive the function that was called. This is why a Venn diagram is used to show the subset relations of lifetimes.

To generalize this model for any stack-based lifetimes, let $m, n \in \mathbb{N}$, $m < n$, and the function scope $S_0 \subset S_1 \subset S_2 \subset \cdots \subset S_n$. The lifetimes of variables in S_n are shorter than those of S_m. Additionally, if the current function is S_m, the frame represented by S_{m+1} should be popped off the stack. Variables in S_m may be used later and must be kept.

Variables with complex lifetimes need to be allocated on the heap because they do not follow the set-based rules of automatic variables. If the stack-based memory structure was applied to heap-allocated variables, they would all occupy S_0 and would last as long as the program is running, potentially consuming all system memory and other resources. Heap-allocated variables need to be manually deleted or periodically cleaned with a garbage collector (GC). The common mark-and-sweep GC uses a graph to represent all heap-allocated variables in a program. [1]

The code below contains two heap-allocated variables. `fortyTwo` is not needed after `answers()` but `glassOfWater` is used outside of `answers()`. A `map` of pointers is used to force the compiler to allocate its values in the heap. Figure 2 illustrates how the GC would mark variables in use and those that can be deallocated.

```
func main() {
        glassOfWater := answers()
        // fortyTwo can be deleted
        fmt.Println(*glassOfWater)
}

func answers() *string {
        answerMap := make(map[string]*string)
        fortyTwo := "42"
        answerMap["What do you get if you multiply six by nine?"] = &fortyTwo
        glassOfWater := "Ask a glass of water."
        answerMap["What's so unpleasant about being drunk?"] = &glassOfWater
        return &glassOfWater
}
```

Although Figure 2 is a very simplified form of Go's GC model, it shows the basic idea of how all

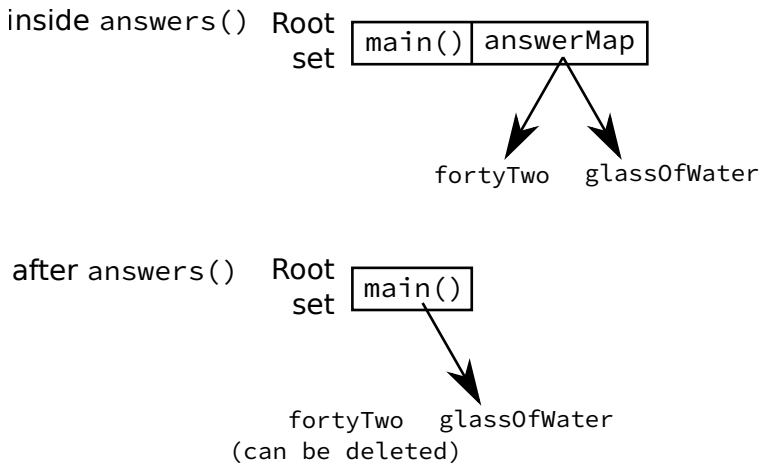

Figure 2: Graphs of the heap at different points in the code

mark-and-sweep garbage collectors are implemented. [2] The root set contains all known active variables, whether those are static, local, or other necessary variables in the current context. [3] The directed edges in the graph represent references to values. The vertices at the head of each edge are the memory blocks that contain some data.

During a GC cycle, the marking stage starts by traversing each graph, starting from the root vertices of the root set. Each vertex that is connected to the root is marked as in use. The remaining vertices remain unmarked. The sweeping stage occurs next, which goes through the memory array and deleting blocks that are not marked. All other blocks are cleared of the in-use flag in preparation for the next cycle. [1]

The traversal algorithm and the need to pause all threads during a GC cycle to avoid race conditions make it a time-expensive operation compared to the simple pointer adjustments to the stack and frame pointers in stack-based allocation. If there are clear ways to define the boundaries of variable lifetimes, they should be allocated on the stack. Otherwise, they can be stored in the heap in the form of a graph which the garbage collector will use to keep track of their lifetimes.

[1] URL: https://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/.
[2] URL: https://blog.golang.org/go15gc.
[3] URL: https://www.dynatrace.com/resources/ebooks/javabook/how-garbage-collection-works/.