GoFree: Reducing Garbage Collection via Compiler-inserted Freeing

Anonymous Author(s)

Abstract

In a memory-managed programming language, programmers allocate memory by creating new objects, but programmers never free memory. A garbage collector periodically reclaims memory used by unreachable objects. As an optimization based on escape analysis, some memory can be freed explicitly by instructions inserted by the compiler. This optimization reduces the cost of garbage collection, without changing the programming model.

We designed and implemented this explicit freeing optimization for the Go language. We devised a new escape analysis that is both powerful and fast ($O(N^2)$ time). Our escape analysis identifies short-lived heap objects that can be safely explicitly deallocated. We also implemented a freeing primitive that is safe for use in concurrent environments.

We evaluated our system, GoFree, on 6 open-source Go programs. GoFree did not observably slow down compilation. At run time, GoFree deallocated on average 14.1% of allocated heap memory. It reduced GC frequency by 7.3%, GC time by 13.0%, wall-clock time by 2.0%, and heap size by 3.6%.

CCS Concepts: • **Software and its engineering → Runtime environments**; **Automated static analysis**.

ACM Reference Format:
Anonymous Author(s). 2018. GoFree: Reducing Garbage Collection via Compiler-inserted Freeing. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2018), 13 pages.

1 Introduction

Garbage collection (GC) frees developers from manual memory management, reducing the risk of memory-related bugs and simplifying coding. However, GC incurs the cost of extra memory usage and scanning time for dead objects.

Go provides a lightweight escape analysis that supports the *stack allocation* optimization but not *explicit deallocation* (Section 2.1.2 explains these optimizations). Our system GoFree implements the explicit deallocation optimization. It supports all of Go's features.

Go's built-in escape analysis runs in $O(N^2)$ time, where N is the program's size. (Throughout this paper, big O notation indicates the average case time complexity.) To keep compilation fast, Go's escape analysis is *field-insensitive* and *flow-insensitive*. It conservatively assumes simplification that anything whose address is assigned to an indirection may escape. For example, `*ppd = pc` on line 24 of fig. 1 causes an edge in Go's escape graph (fig. 1) from `pc`'s node to a virtual "heap location" — which means "this address escapes to the heap" — rather than an edge connecting `ppd` and `pc`.

2018. ACM 2475-1421/2018/1-ART1
https://doi.org/

```
1  type BigType struct {
2    fat [10000000]int // Too
         big for stack
3    p   *int
4  }
5
6  func escapeGraphDemo() *int
       {
7    var s []int
8    for i := 0; i < 10; i++ {
9      s = make([]int, 3)
10     // make() outlived by s
11   }
12
13   var b int
14   // b's address is stored
       into heap
```

```
15   var bigObj BigType // too
       big
16   bigObj.p = &b
17   var c int
18   // c's address is
       indirectly stored
19   var pc *int = &c
20
21   var d int // d escapes
       from function
22   var pd *int = &d
23   var ppd **int = &pd
24   *ppd = pc
25   // c's address is stored
       into pd
26   var pd2 *int = *ppd
27   return pd2
28 }
```

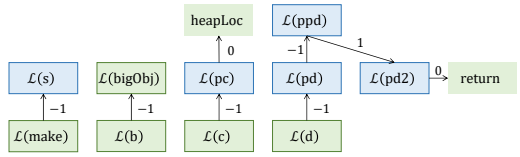

Figure 1: Go code and its escape graph. The escape graph determines whether a location (a node in the graph) is allocated on the stack (blue) or in the heap (green) by solving memory constraints. Virtual locations (nodes without frames) are dummy locations for simplifying analysis that do not stand for specific memory units. Numbers on edges are explained in fig. 5.

The escape analysis could be made more precise by providing it points-to information, which would enable conservative creation of multiple normal edges rather than an edge to the virtual "heap location". Unfortunately, precise pointer analysis is generally $O(N^3)$ [2], which conflicts with Go's goal of fast compilation. Previous studies seldom discuss the complexity of escape analysis in Go. For example, [38] finds an optimization opportunity in Go's handling of interface parameters. Other discussions [28] show code snippets where an obvious improvement can be done, but without acknowledging that fully implementing that would raise Go escape analysis' complexity to $O(N^3)$.

This paper presents GoFree, an $O(N^2)$ explicit deallocation analysis for Go that co-exists with Go's GC and its stack allocation optimization. Unlike approaches that increase precision at the cost of asymptotically greater run time, GoFree maintains $O(N^2)$ complexity by extracting relatively precise information from the original Go escape graph to support the explicit deallocation optimization. Unlike the stack allocation optimization, GoFree enables optimization across nested scopes and function calls.

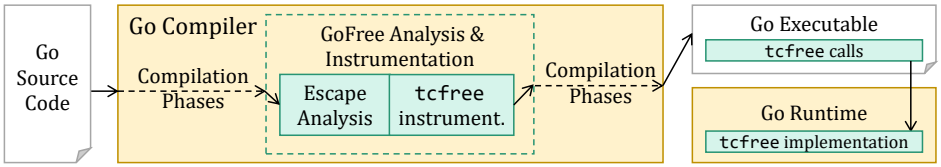

Figure 2: Architecture of GoFree. Blocks in yellow are existing components of the Go compiler and runtime. Blocks in teal are enhancements by GoFree. During compilation, GoFree inserts explicit deallocation (`tcfree` calls) in the executable. During execution, `tcfree` calls into the enhanced Go runtime.

GoFree consists of two main components (fig. 2): the *static analysis and instrumentation* component (section 4) and the *runtime* component for explicit deallocation (section 5).

The static analysis and instrumentation component performs completeness analysis (section 4.2), lifetime analysis (section 4.3), and enhanced inter-procedural analysis (IPA, section 4.4). The completeness analysis reveals which pointers' points-to sets computed from Go escape graph are complete, i.e. contains every variable that it can possibly point to during runtime. The lifetime analysis determines which objects (pointed to by a complete pointer) escape — that is, outlive the pointer's lifetime. The instrumentation (section 4.5) inserts a `tcfree` primitive at the end of the pointer's scope to deallocate the object.

A key part of the runtime support is our new `tcfree` primitive. For small objects allocated in thread-local caches, `tcfree` merely reverts the allocator pointer. For large objects allocated in the global heap, `tcfree` uses a 2-step sweep approach to avoid locking the entire heap (see fig. 10 and section 5). Unlike C's `free` primitive, `tcfree` does not guarantee successful collection. Whenever deallocation would be too costly (e.g., the object is in a non-thread-local place that requires locking to deallocate) or unsafe (e.g., GC is running, leading to a possible deallocation race), `tcfree` returns early leaving the object as if `tcfree` was never called. This strategy is safe: even if `tcfree` does nothing, GC will eventually deallocate the dead object.

GoFree differs from previous escape analysis work in two aspects. First, GoFree sets compilation speed rather than precision as the priority to meet Go's design goal. We implemented a novel, precise deallocation analysis that outperforms Go's escape analysis without increasing its $O(N^2)$ time complexity. Second, GoFree handles Go features. One example is multiple return values: when the returned values have different allocation properties, a single per-method analysis, and summary is inadequate. Another example is Go's *runtime-managed* dynamic data structures, such as slice and map, whose resizing does not appear in any library.

GoFree did not observably slow down compilation. At run time, GoFree deallocated on average 14.1% of allocated heap memory, and it reduced GC frequency by 7.2%, GC time overhead by 14.2%, wall-clock time by 1.9%, and heap size by 3.9%. The speedups apply to GoFree itself: when it is compiled by GoFree, it compiles other Go programs 1.8% faster.

In summary, our key contributions are as follows:

- An $O(N^2)$ static analysis that supports explicit deallocation.
- Runtime support for `tcfree`: explicit deallocation primitives that allow efficient deallocation.
- An evaluation of GoFree.

Our ideas apply to any runtime supporting regions or explicit deallocation. Some generalizable insights that led to GoFree include:

- The static analysis can offload complexity to the runtime system. For example, `tcfree` accommodates double free in certain cases, so the static analysis can use `tcfree` even when it cannot prove no double free would occur.
- In a managed language, the `free` primitive is allowed to fail to free an object since GC will sweep it up in the end. This enables a low-cost best-effort `free` design.
- Identifying conservativeness can indicate where results are precise (not conservative). Sound static analysis makes conservative assumptions; tracking their effect enables later analysis to improve the results or to utilize results that are already precise.

2 Related work

2.1 Escape Analysis

Escape analysis [5, 15, 29, 35, 37] determines the scope and lifetime of objects. If fig. 3 is compiled without escape analysis, then both `make1` and `make2` will be allocated in the heap.

2.1.1 Compiler Optimizations based on Escape Analysis.

Compiler optimizations, such as escape analysis, can determine the scope or lifetime of an object and explicitly free it when no longer accessed, rather than forcing GC to discover when it is no longer accessible.

An optimization may allocate objects on the stack, but they still logically reside in the heap.

Stack Allocation. [10, 11, 19, 39] identifies objects that are local to a method invocation and allocates them on the stack; popping the stack frees them, without any GC.

```
1 func analyses(n int) {
2   s1 := make([]int, 335) // make1
3   // ... use s1 ...
4   for i := 1; i < n; i++ {
5     s2 := make([]int, i) // make2
6     // ... use s2 ...
7   }
8 }
```

Figure 3: Go Code Snippet for Comparing Escape Analyses

The Go compiler implements stack allocation. Each object of constant size that does not live beyond its scope (according to Go's built-in escape analysis) is allocated on the stack to avoid GC. In fig. 3, `make1` can be allocated on the stack, while `make2` must be heap-allocated because its size is not a compile-time constant.

Region-based memory management. [1, 6, 21, 36] allocates and frees memory in regions or arenas. No GC is required for any of the objects in the region, because the entire region is returned to the free list whenever appropriate. However, no partial deallocation within the region is allowed before the whole region is freed.

Explicit Deallocation. [9, 22, 34, 40] identifies the last use of an object and automatically inserts an explicit free primitive after it, similar to what a C programmer would do. This is more fine-grained and flexible than stack or region-based allocation (it can free an object even if the object escapes from the original scope or has variable size), but it typically requires more precise information (points-to sets, flow sensitivity, liveness) to operate.

Go does not support explicit deallocation, while GoFree does. In fig. 3, `make2` can be explicitly freed by inserting a `free` primitive call after line 6.

2.1.2 Precision and Complexity of Escape Analysis.

Fast Escape Analysis [19]. Fast escape analysis is an $O(N)$ algorithm supporting stack allocation. It only propagates escape properties among references and does not distinguish among `new`-ed objects. An object is stack allocated iff the reference it is immediately bound to upon calling `new` does not escape. This simplification sometimes causes unnecessary heap allocation. Because the analysis does not provide any nontrivial points-to information, it is not capable of supporting explicit deallocation.

Connection Graph Based Escape Analysis [2, 10, 11, 22, 37]. A connection graph models the data flow among objects and their addresses, recording the effects of address fetches, assignments, indirect stores, and indirect loads. The algorithm propagates constraints across reference and object nodes. This precision supports better stack allocation than Fast Escape Analysis and also supports explicit deallocation. However, as modeling indirect stores (e.g., `*p = q` in C, or `v.f = u` in Java) may generate up to $O(N)$ edges from a single statement, the analysis costs $O(N^3)$.

2.2 Related Memory Management Techniques

Deferred reference counting. [16] reduces reference counting overhead by not reference-counting the stack. Go does not implement reference counting.

Scalar replacement. [7, 35] decomposes a composite data structure into simpler ones, making them more likely to be eligible for stack allocation. This is especially helpful in languages like Java, where an object and its components can only be accessed via references. Go uses explicit pointers and thus allows access to an object by pointer or value, so scalar replacement is impossible. GoFree implements a similar optimization for multiple return values (section 4.6.3).

3 Background about Go

3.1 Go memory management

Go is a memory-managed language. Over 1.1 million professional developers use Go [26, 41].

All objects are conceptually allocated on the heap [20].

- **Invariant 1**: A pointer to a stack object is not stored in the heap.
- **Invariant 2**: A pointer to a stack object does not outlive that object.

3.2 Go Escape Analysis

Go's escape analysis seeks a balance between compilation speed and execution performance. It simplifies the connection graph approach by removing flow-sensitivity, field-sensitivity, and the tracking of indirect stores (e.g., `*p = q`). Conservative handling of these cases results in some precision loss compared to the connection graph approach, but it still performs better than Fast Escape Analysis. This simplified connection graph is called the *escape graph*.

With indirect stores omitted, each Go statement can generate at most a constant number of nodes and edges in the graph from a statement, so the number of nodes and edges in the graph are both $O(N)$. Go then walks the graph and propagates properties using a modified Bellman-Ford algorithm [3] with $O(N^2)$ time complexity on the sparse graph.

Section 4.1 formalizes Go's escape analysis.

3.3 Go's Heap Allocator and Garbage Collector

Go uses a thread-caching, size-segregated, free-list allocator called TCMalloc [32] to support high-concurrency environments, eliminating global locking for most allocations. After requesting memory pages from the OS, TCMalloc divides them into arena chunks of different small object sizes called *mspans*. Each thread may cache some mspans in its exclusively owned *mcache* so that most allocations from its mspans are lock-free unless the *mspan* runs out and needs to request another *mspan* from the OS.

Go's garbage collector accommodates TCMalloc and concurrent environments. It adopts a non-moving strategy because Go allows the explicit use of pointers by programmers.

4 Static Analysis and Instrumentation

This section details GoFree's explicit free analysis and instrumentation. Section 5 discusses run-time support.

First, section 4.1 introduces the original $O(N^2)$ Go escape analysis framework. The two key analyses of GoFree are completeness analysis (section 4.2) and lifetime analysis (section 4.3); their extra constraints fit in the $O(N^2)$ framework.

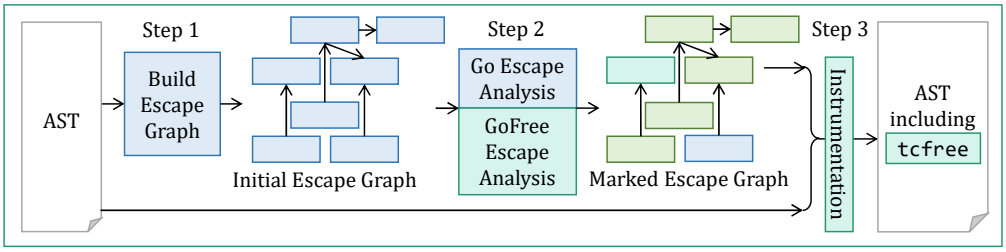

Figure 4: GoFree static analysis and instrumentation. GoFree reuses Go's original escape graph. After escape analysis, nodes in the graph are marked with different properties (such as *HeapAlloc*) that satisfy both Go's and GoFree's constraints. `tcfree` calls are then inserted into the AST according to these properties. Blue and green boxes are components that the original Go system includes. Teal boxes are GoFree-exclusive components.

Table 1: Escape properties used in GoFree

Property[Def#]	Cost[‡]	Description
LoopDepth[4.2†]	$O(N)$	layers of loops that the location is in
HeapAlloc[4.9†]	$O(N^2)$	location must be heap-allocated
Exposes[4.10]	$O(N^2)$	loc. may cause untracked modification to its referent
Incomplete[4.11]	$O(N^2)$	loc.'s value may be changed by untracked data flow
DeclDepth[4.12]	$O(N)$	layers of scopes that the location is in
OutermostRef[4.13]	$O(N^2)$	the smallest scope that covers location's lifetime
Outlived[4.14]	$O(N^2)$	loc. has shorter lifetime than any object it points to
PointsToHeap[4.15]	$O(N^2)$	location may point to at least one heap object
ToFree[4.16]	$O(N^2)$	location is qualified to be deallocated by `tcfree`

† These two properties come from the original Go's escape analysis.
‡ $O(N)$ arises from an AST scan. $O(N^2)$ arises from propagation.

Section 4.4 extends GoFree to support inter-procedural analysis (IPA) to discover more optimization opportunities across function calls. Section 4.5 discusses insertion of `tcfree` calls. Finally, section 4.6 discusses how GoFree supports and utilizes Go's modern language features.

4.1 Formalization of the Go Escape Analysis

This section formally describes the original Go escape analysis, an $O(N^2)$ algorithm built for stack allocation optimization. It has two steps: building an Escape Graph that models program data flow, and solving Escape Properties based on constraints on the graph.

Go's escape analysis yields a memory allocation scheme that is correct (each variable in Go exists as long as there are references to it) and strives for efficiency (put as many objects on the stack as possible). It first builds a directed weighted graph called an escape graph to model the data flow among objects. AST nodes that allocate memory are represented by escape graph nodes called locations, and data flow relations are represented by escape graph edges.

Definition 4.1 (Escape Graph)**.** An escape graph is a directed weighted graph, $EG = (L, E)$, where

- L is a set of locations defined in definition 4.2. l and m are locations.
- E is a set of edges defined in definition 4.3. e and $\langle l, m\rangle$ are edges.

Definition 4.2 (Location)**.** A location $l \in L$ is a vertex in the escape graph, which represents storage space. When we say $l = \mathcal{L}(n)$, we mean that l represents the storage space created (explicitly or implicitly) by a (declaration or expression) node n in the AST. That is, $\mathcal{L}$ maps an AST node to its corresponding location. $LoopDepth(l) \in \mathbb{Z}$ is the loop depth at the declaration of variable n.

code	edge	meaning of edge
`p = *q`	$q \xrightarrow{1} p$	p may equal *q
`p = q`	$q \xrightarrow{0} p$	p may equal q
`p = &q`	$q \xrightarrow{-1} p$	p may equal &q
`*p = q`	$q \xrightarrow{0} \texttt{heapLoc}$	q's value may be in the heap

Figure 5: Go escape graph edges. $Derefs(e)$ is the weight of edge e.

Go's escape analysis is *field-insensitive*. When abstracting an object or an array as a location, all fields of the object or all elements of the array are represented by one single location.

To simplify the escape graph, it contains dummy locations. Location `heapLoc` is a global constant that represents a heap location, while per-function location *return* represents the memory space used to pass the function's return value. Their $LoopDepth()$ is set to the special value $+\infty$ to avoid affecting other non-dummy locations.

Definition 4.3 (Edge)**.** $e = \langle l_1, l_2\rangle \in E$ is a directed weighted edge in the escape graph.

Definition 4.4 (Derefs)**.** $Derefs(e) \in \mathbb{Z}$ is the weight of edge e, representing the dereference count from l_1 to l_2.

Figure 5 shows how edges model assignments. Indirect store `*p = q` is not further tracked to avoid generating $O(N)$ edges for this single statement and degrading Go's escape analysis to $O(N^3)$. This simplification is safe because it conservatively indicates that q's value could go into the heap in the worst case.

Go's escape analysis is *flow-insensitive*. The order and scope of statements have no effect on the escape graph. Since each statement generates a constant number of locations and edges, $|L| = O(N)$ and $|E| = O(N)$.

Using the rules introduced above, the code example in fig. 1 derives the escape graph shown in fig. 1. Once the escape graph is complete, Go creates and solves constraints on Escape Properties.

Definition 4.5 (Holds). $m \in Holds(l)$ means that l can possibly contain the value of m, the address of m, and/or the value of a location that m points to (if m is a pointer). In the context of $Holds(l)$,

- $l \in L$ is called the root location,
- each $m \in Holds(l)$ is called a leaf location.

$|Holds(l)| \leq |L| = O(N)$, so by walking the escape graph in the reverse direction of edges, Go computes $Holds(l)$ in $O(N)$ time. $Holds(l)$ may be *incomplete*, missing some locations that also hold l because the escape graph does not track indirect stores.

Definition 4.6 (TrackDerefs). $TrackDerefs(l_0 l_1 l_2 ... l_n) \in \mathbb{Z}$ is how many times value l_n is dereferenced when it is obtained via track $l_0 l_1 l_2 ... l_n$. A track is a loop-free path. $TrackDerefs(l_0 ... l_n)$ is computed by adding each $Derefs()$ value when walking the track in reverse, maintaining a lower bound 0 before each addition.

- $TrackDerefs(l_{n-1} l_n) = Derefs(\langle l_{n-1}, l_n \rangle)$,
- $TrackDerefs(l_i l_{i+1} ... l_n) = max(0, TrackDerefs(l_{i+1} ... l_n)) + Derefs(\langle l_i, l_{i+1} \rangle)$.

If $TrackDerefs(l_0 ... l_n) = -1$, it means that l_0's address can possibly be obtained by l_n via this track. If $TrackDerefs(l_0 ... l_n) \geq 0$, it means that l_0's value, or any value retrieved by dereferencing it one or more times, can possibly be obtained by l_n via this track.

Definition 4.7 (MinDerefs). $MinDerefs(m, l)$ is the minimum value of $TrackDerefs(t)$ via any track t from m to l. It is defined iff $m \in Holds(l)$.

$$MinDerefs(m, l) = \min_{t=m...l} TrackDerefs(t)$$

Go computes the smallest dereference count because a variable can be an object, a pointer, or even an object containing pointers of different orders. Two different tracks with the same source and destination may have different $TrackDerefs()$. For example, `bigObj` in fig. 1 acts as a 0-order pointer (i.e., a value) with field `fat` (line 2) and a 1-order pointer with field `p` (line 3). Taking the minimum value of dereferences conservatively assumes the leaf object as its highest order pointer, being aware of its ability to pass on other locations' addresses.

Definition 4.8 (PointsTo). $PointsTo(l) \subseteq L$ represents the points-to set of location l, where

$$m \in PointsTo(l) \text{ iff } MinDerefs(m, l) = -1.$$

$m \in PointsTo(l)$ means that l's value may be the address of m, similar to the result of points-to analysis. A queue-optimized Bellman-Ford algorithm [3, 8, 17] runs in $O(N)$ average time on such a sparse graph with limited edge weight. $PointsTo(l)$ may also be incomplete due to the simplification of indirect stores. We will further discuss this in completeness analysis (see section 4.2).

```
1  func walkall(EG = (L, E)):
2    work := UniqueQueue(copy(L))
3    for len(work) > 0: // O(N) repetitions
4      root := work.pop()
5      for leaf in Holds(root): // O(N) repetitions
6        leafUpdated := applyConstraints(root, leaf)
7        if leafUpdated:
8          work.push(leaf)
9        // Extension from GoFree.
10       // rootUpdated := applyConstraints(leaf, root)
11       // if rootUpdated:
12       //   work.push(root)
13       //   break
```

Figure 6: Go's $O(N^2)$ property propagation algorithm

Definition 4.9 (HeapAlloc). $HeapAlloc(l)$ is true if location l must be heap allocated. $HeapAlloc(l)$ is true if

- l = `heapLoc`, or
- l = *return*, or
- $\exists m \,.\, l \in PointsTo(m) \wedge HeapAlloc(m)$, or
- $\exists m \,.\, l \in PointsTo(m) \wedge LoopDepth(m) < LoopDepth(l)$.

where l is in the same function as m.

Go's escape analysis (fig. 6) finds a minimum solution for constraints. Properties propagate only from root locations to leaf locations, similar to the slack operation in the Bellman-Ford algorithm. Inspired by this, Go's escape analysis keeps a work queue for newly updated locations, and takes one root location from the queue to update all other leaf locations until the queue is empty. Since each root takes $O(N)$ time to update others, and one location can be updated and queued at most a constant number of times, this algorithm has $O(N^2)$ time complexity.

4.2 Completeness Analysis

Explicit deallocation requires a complete points-to set to ensure that every possibly-pointed-to object can be freed before calling `tcfree` on a pointer. That is, the lifetime of an object cannot exceed the pointer pointing to it. Stack allocation can be safe without a complete points-to set because any indirect stores have been conservatively modeled as storing into heap in the worst case. If the estimate of an address of an object includes the heap, the object is heap-allocated.

Go's escape analysis does not guarantee a complete points-to set due to its simplification on indirect stores. For example, in fig. 1, `c`'s address is held by `pd2` due to the indirect store on line 19. `c` is safely heap-allocated according to the conservative $\langle \mathcal{L}(\mathsf{pc}), \mathsf{heapLoc} \rangle$ edge, but this does not tell `pd2` anything about `c`. $PointsTo(\mathcal{L}(\mathsf{pd2}))$ lacks `c`, which means it is incomplete.

Table 2 compares $PointsTo(\mathcal{L}(\mathsf{pd2}))$ in different kinds of escape analysis The faster the algorithm is, the more data flow information it omits. Among the three algorithms, only the connection graph provides complete points-to sets. Whenever encountering dereferencing or field accessing, Fast Escape Analysis gives incomplete points-to sets. Whenever

Table 2: Points-to sets in different escape analyses

Method	Fast Esc. Analysis	Go esc. graph	Conn. graph
Time complexity	$O(N)$	$O(N^2)$	$O(N^3)$
Omitted dataflow	`*ppd = pc;` `pd2 = *ppd`	`*ppd = pc`	none
Conservatively modeled as	`{heap} = pc;` `{heap} = pd2`	`{heap} = pc`	none
PointsTo($\mathcal{L}$(pd2))	$\varnothing$	{$\mathcal{L}$(d)}	{$\mathcal{L}$(c), $\mathcal{L}$(d)}

encountering an indirect store, the Go escape graph gives incomplete points-to sets.

The part of the escape graph not affected by indirect stores can contain complete, precise points-to sets. For example, in fig. 1, *PointsTo*($\mathcal{L}$(s)), *PointsTo*($\mathcal{L}$(bigObj)), and *PointsTo*($\mathcal{L}$(pc)) are complete and precise.

To identify which locations are unaffected by indirect stores and thus have complete points-to sets, we introduce two new properties for locations and their constraints.

Definition 4.10 (Exposes)**.** *Exposes*(l) is true if untracked modifications to locations in *PointsTo*(l) may have been made by storing indirectly into l.[1] *Exposes*(l) is true if

- l = heapLoc, or
- l = *return*, or
- $l = \mathcal{L}(n)$ and n is the destination of an indirect store (`*n = ...`), or
- $\exists m \,.\, l \in \mathit{Holds}(m) \wedge \mathit{MinDerefs}(l, m) \leq 0 \wedge \mathit{Exposes}(m)$.

Exposes(l) does not mean l has an incomplete points-to set. It means that l exposes the addresses of locations in *PointsTo*(l) to an under-tracked place, so their values might be changed by indirect stores elsewhere and are thus incomplete. For example, in fig. 1, *Exposes*($\mathcal{L}$(pc)) is true because it exposes c's address, but $\mathcal{L}$(pc) itself is complete as all changes to its value are tracked, and it only fetches c's address.

Definition 4.11 (Incomplete)**.** *Incomplete*(l) is true if l can point to locations not in *PointsTo*(l) at run time. *Incomplete*(l) is true if

- l is a formal parameter, or
- $\exists m \,.\, l \in \mathit{PointsTo}(m) \wedge \mathit{Exposes}(m)$, or
- $\exists m \,.\, m \in \mathit{Holds}(l) \wedge \mathit{Incomplete}(m)$.

We enhance the property propagation algorithm in fig. 6 to support our new constraints. The last constraint in definition 4.11 propagates properties from leaf locations to root locations, which is in the reversed direction of previous constraints. Lines 10–13 in fig. 6 support back-propagation. It is still an $O(N^2)$ algorithm. Previously, only leaf locations could be updated and re-queued on each walk from the root, but now we also allow the updating and re-queuing of the root itself. The root still has only a constant number of properties to be updated, so it can only be re-queued at most constant times. So, the $O(N^2)$ complexity is not increased.

As of now, we can compute a location's points-to set and decide its completeness in $O(N^2)$ time.

[1] *Exposes*() and *Incomplete*() need not be computed for data types not containing pointers, such as scalars, scalar arrays, and objects containing only these types.

4.3 Lifetime Analysis

The lifetime analysis in GoFree collects lifetime information and instruments `tcfree` calls.

The stack allocation optimization determines whether or not an object lives beyond its scope (the brace pair in which it is allocated). Stack allocation has one opportunity to free an object, which is at the end of its scope. By contrast, GoFree's lifetime analysis can find out how many layers of scopes the object has escaped from and determine the exact scope where it is safe to free the object. With this information, GoFree can free objects that escape from several scopes and even the function (with inter-procedural analysis, section 4.4).

Definition 4.12 (DeclDepth)**.** *DeclDepth*(l) $\in \mathbb{Z}$ records the scope depth at the declaration of variable n, where $l = \mathcal{L}$(n).

Definition 4.13 (OutermostRef)**.** *OutermostRef*(l) $\in \mathbb{Z}$ is the smallest scope in a function that covers l's lifetime. It takes the greatest value satisfying the following two constraints:

- $\mathit{OutermostRef}(l) \leq \mathit{DeclDepth}(l)$,
- $\forall m \,.\, l \in \mathit{PointsTo}(m) \Rightarrow \mathit{OutermostRef}(l) \leq \mathit{DeclDepth}(m)$.

OutermostRef's value is always taken from a location's *DeclDepth* and does not further propagate. As a result, despite being an integer, it will not add complexity to the $O(N^2)$ propagation algorithm.

If a pointer's lifetime ends before the lifetime of the object it points to, then we say the pointer it has been "outlived" and is not safe to free.

Definition 4.14 (Outlived)**.** *Outlived*(l) is true if l has a shorter lifetime than any object that it points to.

- $\mathit{Outlived}(l) = l \in \mathit{PointsTo}(m) \wedge \mathit{OutermostRef}(l) < \mathit{DeclDepth}(m)$

As explicit deallocation applies only to heap objects, there is no need to call `tcfree` on a pointer that only points to stack objects. (Such a pointer will be stack-allocated.) However, such a call is safe because `tcfree` ignores stack objects.

Definition 4.15 (PointsToHeap)**.** *PointsToHeap*(l) is true if l might point to a heap object.

- $\mathit{PointsToHeap}(l) = \exists m \,.\, m \in \mathit{PointsTo}(l) \wedge \mathit{HeapAlloc}(m)$.

The final step is to determine which location to free using `tcfree`. This location must be both safe to free and worthy of being freed.

Definition 4.16 (ToFree)**.** *ToFree*(l) is true if l is qualified to be deallocated by `tcfree`.

$$\mathit{ToFree}(l) = \neg\mathit{Incomplete}(l) \wedge \neg\mathit{Outlived}(l) \wedge \mathit{PointsToHeap}(l)$$

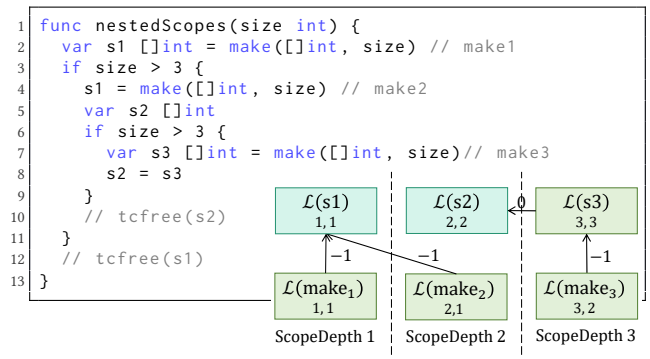

Figure 7: Nested scopes. Numbers under each node are their *DeclDepth*() and *OutermostRef* () values.

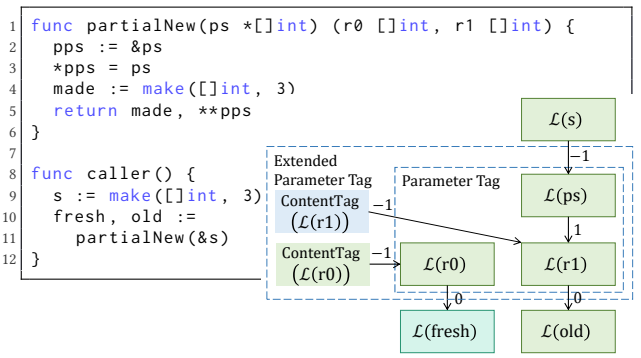

Figure 8: Inter-procedural analysis. Content tags summarize the escape properties of what a return value points to.

We illustrate lifetime analysis with an example of nested scopes as shown in fig. 7. The two numbers shown under each location in fig. 7 represent its *DeclDepth*() and *OutermostRef* () values. All three slices in fig. 7 are heap-allocated because of non-constant size. The lifetime analysis identifies both `s1` and `s2` have complete points-to sets and are not outlived by the objects they point to. Therefore, two `tcfree`s are inserted to free them. However, `s3` has passed the address of its underlying object to an outer scope, making it outlived, so it is not safe to free it within its scope.

4.4 Inter-procedural Analysis

Go's escape analysis provides an inter-procedural analysis framework called *parameter tagging* based on the escape graph, which can be seen as a generalization of the function summary technique. After intra-procedural analysis, a function is abstracted into a *parameter tag*, a compressed escape graph of the function. A parameter tag's locations include only the function's parameter and return values. The detailed data flow within the function is compressed into edges directly from parameters to return values. The edges' *Derefs*() are taken from *MinDerefs*() on the full escape graph.

At a call site, a copy of the callee's parameter tag is embedded into the caller as a subgraph. Go tries to order the intra-procedural analysis of functions inner-to-outer so that more call sites will find known parameter tags. If it is unknown (possibly due to recursion or closures), Go uses a conservative subgraph where all parameters flow to the heap and all return values come from the heap.

Parameter tagging does not support explicit deallocation because it does not include information about objects pointed to by return values, thus losing the completeness of their points-to sets. As shown in fig. 8, `fresh` cannot find the inner `make` (or any of its abstraction) from $PointsTo(\mathcal{L}(\texttt{r0}))$, so misses the opportunity to deallocate it.

GoFree uses a new approach. *Content tagging* summarizes return values' points-to sets to provide them to the caller so that newly allocated objects in the callee could be explicitly deallocated in the caller. For each return value location l, GoFree adds a dummy *content tag* location $m = ContentTag(l)$ to summarize its points-to set, and an edge $e = \langle m, l \rangle$ with $Derefs(e) = -1$. After intra-procedural analysis, GoFree adjusts a few of l's and m's properties before adding them both to the extended parameter tag:

- $LoopDepth(l) = DeclDepth(l) = LoopDepth(m) = DeclDepth(m) = +\infty$.
- $HeapAlloc(m) = PointsToHeap(l)$.
- $Incomplete(l) = Incomplete(m)$.

The first rule sets the depths to a large enough value so that they do not appear in the caller as if used by an outer scope. The second rule summarizes into $HeapAlloc(m)$ whether any location in $PointsTo(l)$ is heap allocated. The third rule tells l to use the incompleteness property of m rather than its own. This is because $Incomplete(l)$ may be propagated from a formal parameter of the callee. As we have conservatively set $Incomplete(param) = \texttt{true}$ in case we have no information about the caller, this may now be a false positive since we have information from the caller in inter-procedural analysis. In contrast, $Incomplete(m)$ could only come from indirect stores within the callee, which must be recorded for safety.

As shown in fig. 8, `fresh` is informed of a deallocation opportunity by $ContentTag(\mathcal{L}(\texttt{r0}))$, which improves analysis accuracy, and $\mathcal{L}(\texttt{old})$ is informed of the incompleteness of $ContentTag(\mathcal{L}(\texttt{r1}))$, so it will not be freed as it is aware of the existence of an indirect store inside the callee.

4.5 `tcfree` Instrumentation

GoFree's runtime includes APIs shown in table 3 that deallocate objects of different types and sizes. Figure 9 shows their calling relationships.

`TcfreeSlice` and `TcfreeMap` are specialized variants of `tcfree` to deallocate built-in data structures of Go (section 4.6).

`Tcfree` receives an address of an object, either from a raw pointer or an unwrapped slice or map, and forwards the address to `TcfreeSmall` or `TcfreeLarge` according to its size.

`TcfreeSmall` and `TcfreeLarge` adopt different strategies to deallocate objects of different sizes efficiently. For their implementation, see section 5.

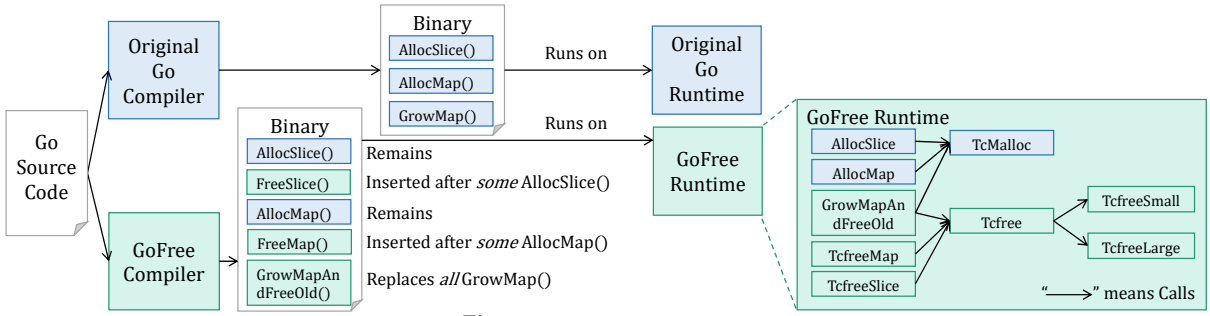

Figure 9: `tcfree` components.

Table 3: The `tcfree` family. Parameters are always addresses.

Runtime API	Parameter	Functionality
`TcfreeSlice`	slice	unwraps the address of underlying array, calls `Tcfree`
`TcfreeMap`	map	unwraps the address of underlying buckets, calls `Tcfree`
`Tcfree`	object	calls `TcfreeSmall` or `TcfreeLarge`
`TcfreeSmall`	small obj.	deallocates a small object from mcache
`TcfreeLarge`	large obj.	deallocates a large object from mcentral

For each location whose *ToFree*() is `true`, the GoFree compiler inserts a corresponding variant of `tcfree` as the last statement (excepting `return` and `goto`, so the `tcfree` is live) of the scope where it is declared. In most cases, the program executes to the last line of a scope, so `tcfree` will be reached. In cases where the function returns from the middle, `tcfree` is not reached, but it is still safe to leave the deallocation to GC.

4.6 Support for Go Language Features

4.6.1 Slice. Slice is a built-in implementation for the linear list in some modern languages, such as Go, Python, and Rust. A slice is typically a fat pointer composed of the address of the underlying array and its length and capacity.

A slice is challenging for escape analysis because its memory is runtime-managed and does not behave as an ordinary object. In an escape graph, slices are equivalent to pointers to their underlying arrays, but these arrays are not always explicitly allocated. When appending to a slice that does not already have an underlying array, one is implicitly allocated. When appending to a full slice, the runtime reallocates a larger space to extend it. These implicit allocations and data flows are not reflected in the escape graph and can cause missing optimizations or even safety problems.

GoFree supports slices by adding dummy content locations m and setting their *HeapAlloc*() = `true` upon each slice appending, conservatively modeling the possibility of implicit allocation. We connect it to the slice location l with edge $e = m \xrightarrow{-1} l$. Slice appending in a loop usually causes implicit heap allocation.

Slices also provide great opportunities for GoFree optimization. They are usually large and do not have a fixed size, making them hard to stack-allocate. Slices are used a lot as temporal buffers with relatively simple data flow, usually local to a scope. GoFree has a variant of `tcfree` for slices called `TcfreeSlice`. When a slice is passed to it, this runtime API unwraps the address of its underlying array and forwards it to deallocation implementations.

We observe that slices may cause a heavy GC burden because of their dynamic and unpredictable memory management, which can outweigh the benefits of reduced copying. Programming language designers should consider more lightweight ways of memory management, such as reference counting, as a supplement to GC for slices.

4.6.2 Map. Map is Go's built-in implementation of hash table. The Go runtime manages its memory. Every access of a map is compiled into a runtime call, which may cause implicit allocation. The biggest part of a map is its *buckets*, a continuous array. When the map's load factor reaches a certain constant, a bigger new bucket array is allocated in the heap to replace the old one.

Go maps provide GoFree two optimization opportunities.

First, when a map grows, its old bucket array is evacuated and abandoned. Since different maps do not share the same bucket array, this abandoned array is in the growing map's exclusive ownership. GoFree deallocates it. This is essentially a runtime optimization that needs no static analysis, but does need `tcfree`. We observe that, even in a managed language, an explicit free primitive can improve memory efficiency.

Second, a map will likely hold an underlying bucket array when it dies. GoFree detects this case similarly to slices. It uses a specialized `TcfreeMap` primitive to unwrap the address of a map's underlying bucket array and forward it to deallocation implementations.

4.6.3 Multiple Return Values. Go provides native language support for multiple return values. Unlike Python or Modern C++, which wrap return values into a tuple and provide syntactic sugar to simulate multiple return values, Go regards each return value as an independent object.

Previous allocation analysis for single-return-value functions typically identifies factory methods [19], which means that the object returned is allocated by the callee. The analysis treats a factory method as a `new` expression in the caller.

Unfortunately, this technique does not work on Go. Each Go function may return many objects (fig. 8), both newly allocated ones $\mathcal{L}$(`r0`) and received ones $\mathcal{L}$(`r1`). A Go function

may be a factory with respect to some return values but not with respect to other return values.

GoFree's inter-procedural analysis generalizes factory method identification with extended parameter tagging and provides enough information to handle multiple return values precisely. Essentially, a factory method is a function that returns something worthy of freeing. GoFree records the *PointsToHeap* property of each return value into the *HeapAlloc* property of its content tag. GoFree also records the completeness property for safety.

4.6.4 Function Inlining.

Function inlining is a common compiler optimization to reduce the cost of calling short functions. Inlined functions are embedded into the caller as part of its code, reducing expensive frame maintenance operations.

Go's escape analysis benefits from inlining. Go stack allocation does not track an object once it escapes from the function and heap allocates it. However, if the callee is inlined, more scopes in the caller are visible to the callee object. If an object escapes from the callee but does not escape from the caller, it can be stack-allocated instead.

GoFree does not benefit from inlining because its inter-procedural analysis provides enough information to analyze the caller function as precisely as intra-procedural analysis. Our extended parameter tags model data flow from parameters to return values, and provide both *HeapAlloc*() and *Incomplete*() properties for return values to analyze them safely and precisely.

5 Runtime support for tcfree

`tcfree` represents a family of explicit free primitives for objects of various types and sizes, which aims to enable safe and low-cost explicit deallocation of heap objects. In cases where explicit deallocation is either unsafe or too expensive, the `tcfree` primitive simply returns without making any changes. This means that the `tcfree` primitive emphasizes safety and efficiency first, and does not always guarantee successful explicit deallocation. Our deallocation strategy is tightly associated with Go's allocation strategy as implemented by TCMalloc, which is dependent on the size of objects.

`tcfree` gives up deallocating and returns with nothing done in the following cases. When GC has been triggered and is already running concurrently with the user program, `tcfree` does not race GC to deallocate any object. If an mspan's ownership has shifted due to Go's runtime thread scheduling between allocation and `tcfree`, `tcfree` gives up because it's unsafe to operate on another thread's mspan.

Figure 9 illustrates the components breakdown of the original Go compiler and our GoFree compiler. Go programs that use built-in data structures like slices and maps are compiled into executable binaries containing corresponding runtime calls for object allocation (and explicit deallocation, for GoFree). Type-specific primitives are inserted into user programs to support different built-in data structures. Calls

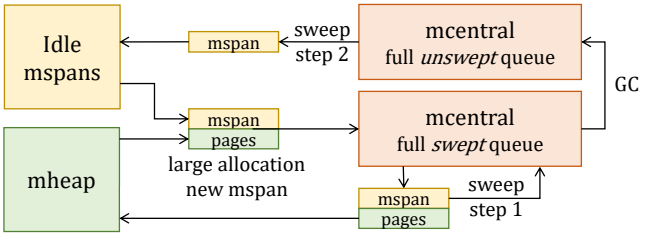

Figure 10: `tcfree` for large objects is implemented with a 2-step sweeping strategy. Yellow blocks are mspan control blocks. Green blocks are memory pages where large objects reside.

to them will be classified by `tcfree` and routed to `TcfreeSmall` or `TcfreeLarge` depending on the size of the object being deallocated.

Explicit Deallocation of Small Objects. Most heap objects are allocated in thread-local mspans, enabling efficient deallocation without locking. When an object of a given size can fit in an mspan, TCMalloc is able to allocate them using a lock-free process. This is accomplished by bumping the mspan's free index and setting an allocation bit, because the allocating thread is in full ownership of the mspan. `tcfree` assumes that it is not called on a long-living object, which is likely still located in the same mspan. If so, deallocation is performed by simply reverting the free index and clearing the allocation bit. If the mcache has been filled and swapped out, `tcfree` does not risk the safety and efficiency of the deallocation process; it just returns without changing anything.

Explicit Deallocation of Large Objects. When objects are too large to fit into a thread-local mspan, TCMalloc allocates them in an exclusive mspan by assigning memory pages to a distinct mspan control block. After allocation, the large object's mspan is pushed into the mcentral, the central cache for mspans not owned by any single thread. `tcfree` performs a 2-step sweeping to minimize its effect on GC, as shown in fig. 10. To deallocate the object, `tcfree` locks the mspan and returns the memory pages it owns to mheap, the central cache for unused memory pages. The mspan is then marked as dangling and to be swept in a normal GC cycle. Although GC is still necessary to deallocate the mspan, most of the space is returned before GC.

Double-free handing. Under the current implementation, a double-free will occur if an object is pointed to by more than one pointer (slice or map control blocks), all of whom are in the same scope and eligible for `tcfree`. However, this will not cause trouble because 1) `tcfree` will ignore any already-freed memory, 2) these `tcfree`s are inserted adjacently and no preemption or new allocation can happen in the middle, and 3) in TCMalloc's policy, no other thread can access the part of memory cached by the current thread.

Table 4: Experimental metrics

Metric	Description
time	time of one execution
GCs	count of GC cycles triggered in one execution
alloced	allocation size: total amount of heap allocation in one execution
freed	free size: total heap memory freed by `tcfree` in one execution
free ratio	free ratio = freed / alloced
maxheap	maximum heap size during one execution

6 Evaluation

This section validates the proposed approach in five aspects.

1. *Effectiveness.* How does explicit deallocation reduce GC time and heap memory occupation?
2. *Efficiency.* How well does GoFree perform on real-world applications?
3. *Deallocation Target Selection.* Why does GoFree choose to free only slices and maps?
4. *Ablation Study.* How much does each category of objects contribute to explicitly deallocated space?
5. *Compilation Speed.* Can GoFree retain fast compiling?
6. *Robustness.* Does GoFree corrupt memory?

We designed specific experiments to answer these questions.

6.1 Experimental Setup

All experiments were conducted on an Ubuntu server with two Intel(R) Xeon(R) Gold 6248R CPUs, each with 3.00GHz processors (48 cores, 96 threads in total) and 503GB memory. A single Go process is allowed to create up to 32 threads.

We implemented GoFree by modifying the official Go 1.17.7 compiler's source code.

Our metrics are described in table 4. Our profiling tool is implemented by hooking Go's runtime library and collecting information throughout the program's execution. The metrics are collected upon specific runtime calls and events, such as heap allocation, `tcfree`, and triggering of a new GC cycle. Our profiling tool has no observable effect on the program's performance.

6.2 Subject programs

There is no standard benchmark suite for Go. We chose 6 open-source programs from GitHub. Each one had more than 5k stars and is a program rather than a service. (It is harder to fairly measure the run time and memory consumption of a service.) `json` is the standard Go library's json parser, which also belongs to the `golang/go` repository. `staticcheck` (scheck) and `structlayout` (slayout) are two different Go programming tools from the same repository `dominikh/go-tools`.

6.3 Effectiveness

Go's TCMalloc is a complex memory allocator with multiple levels of caches. The effect of deallocation goes through these cache layers and the Go GC mechanism before finally being propagated to overall performance. Through controlled experiments illustrated in fig. 11, we observe that

```
func mapPopulate() {
  for n := 0; n < 10000000/C; n++ {
    m := make(map[int]int)
    for i := 0; i < C; i++ {
      m[i] = i
    }
  }
}
```

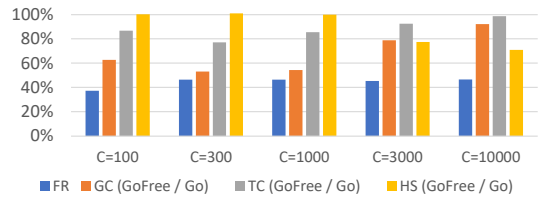

Figure 11: Microbenchmark map experiment. A bigger `C` value (used in the code above) means that the average size of the deallocated objects is bigger.

the benefit of explicit deallocation is reflected in the reduction of either time or memory cost, or both. Which of the two benefits more is more related to the average size of deallocated objects.

As shown in fig. 11, each value of `C` results in a similar free ratio, indicating comparable amounts of deallocation. However, as `C` grows larger, the mean size of deallocated objects increases, resulting in greater reductions in heap size and less significant reductions in GC frequency and time consumption.

6.4 Efficiency

Our metrics behave as a random distribution across different runs (typically fig. 12), so we ran each program under 3 settings, 99 times for each setting, and take the average. The settings are: (1) compile with Go, (2) compile with GoFree, and (3) compile with Go but turn off GC at runtime (`Go-GCOff`). We compare time, GC and maxheap between settings (1) and

Table 5: Effect of GoFree's optimizations

Project	time			GC time	GCs			free	maxheap		
	ratio	stdev	*p*-value	ratio	ratio	stdev	*p*-value	ratio	ratio	stdev	*p*-value
Go	98.6%	1.6%	< 0.0001	91.7%	95.2%	2.7%	< 0.0001	12.0%	98.5%	2.5%	< 0.0001
hugo	99.9%	10.6%	0.4603	99.6%	97.5%	2.1%	< 0.0001	5.5%	99.4%	7.2%	0.2883
badger	99.8%	2.8%	0.2527	98.0%	94.9%	9.2%	0.0003	4.0%	100.2%	9.0%	0.4197
json	93.6%	1.4%	< 0.0001	55.0%	77.0%	0.0%	< 0.0001	22.8%	95.7%	0.2%	< 0.0001
scheck	97.5%	1.9%	< 0.0001	82.9%	93.7%	4.1%	< 0.0001	15.2%	95.5%	4.7%	< 0.0001
slayout	99.0%	5.5%	0.0975	5.1%	97.9%	2.7%	< 0.0001	24.9%	89.1%	2.3%	< 0.0001

Data in grey are not significant at $p = 0.01$, i.e., neither GoFree nor Go is better.
Ratios (%) are GoFree / Go.

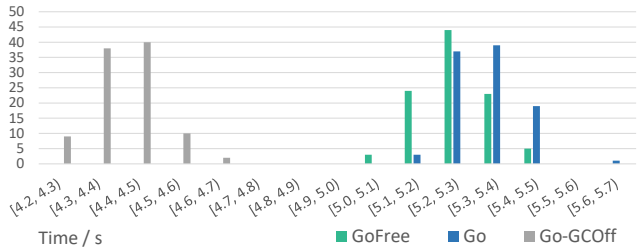

Figure 12: Time consumption distribution of the Go compiler

Table 6: Stack/heap allocation decisions of slices, maps, and other data structures.

Scanned Project	Stack others	Heap GC others	Stack slices	Heap tcfree slices	Heap GC slices	Heap tcfree slices	Stack maps	Heap tcfree maps	Heap GC maps	Heap tcfree maps
Go	461583	1021	2288	660	4692	8.6%	142	304	431	34.7%
hugo	476897	3698	2521	728	7314	6.9%	373	159	1271	8.8%
badger	145569	446	1323	156	2234	4.2%	89	24	100	11.3%
Go/json	45244	45	241	34	280	6.1%	4	3	1	37.5%
scheck	166269	455	1123	263	2417	6.9%	77	84	201	23.2%
slayout	78249	167	364	94	822	7.3%	22	32	84	23.2%

Column 7 is equal to column 5 divided by the sum of columns 5 and 6.
Column 11 is equal to column 9 divided by the sum of columns 9 and 10.

(2) to get the "ratio" columns in table 5. We compute GC time overhead as $\Delta time_{GC} = time_{Go} - time_{GoGCOff}$, and GC time reduction as $(time_{Go} - time_{GoFree})/\Delta T_{GC}$. We also compute standard deviations and p-values for time, GC, and maxheap to examine whether GoFree's effect is significant ($p < 0.01$).

In each run, we did not set a heap size limit because the Go runtime has its own algorithm to balance between space-time tradeoff. The "GOGC" percentage sets a soft goal for the Go GC pacing mechanism, indicating the desired heap growth percentage relative to the heap size at the end of the last GC cycle before triggering the next GC cycle. Go will never trade a stop-the-world GC for enforcing this soft goal. More recent versions of Go do provide a way to set a memory limit, but that is also a similar soft goal mechanism.

On average of the 6 programs, GoFree deallocated 14.1% of allocated heap memory, reduced GC frequency by 7.3%, GC time overhead by 13.0%, wall-clock time by 2.0%, and heap size by 3.6%.

In significant results, GoFree always reduces GC frequency, sometimes reduces time and maxheap, and never increases time or maxheap. GoFree reduces most time from `json`, by 6.4%, and most maxheap from `structlayout`, by 10.9%. Typically, programs with higher free ratio benefit more from GoFree. Apart from the programs listed, we also briefly tested on `protobuf-go`, `fastjson`, `fzf`, and `gods`. They have too low free ratio (< 5%) so we assume GoFree will not optimize them well, but we still want to report this fact.

The Go compiler is written in Go, so it can benefit from GoFree. When GoFree is compiled by GoFree, it compiles other Go programs faster by 1.4%.

6.5 Choice of Deallocation Target

Table 6 reports how often Go stack-allocates slices, maps, and all other data structures including user-defined ones. Columns 2 and 3 show that Go's stack allocation is very effective for the "other" category. Based on this, GoFree operates only on stacks and maps, to avoid adding overhead but little benefit for other types. Other columns of table 6 show that GoFree achieves significant savings for slices and maps.

6.6 Ablation Study

`tcfree` can reclaim memory from three sources: when a slice dies, when a map dies, and the old bucket when a map grows.

Table 7: Ablation study of the contribution breakdown of the three deallocation categories

Project	`FreeSlice()`	`FreeMap()`	`GrowMapAndFreeOld()`
Go	56.0%	13.9%	30.1%
hugo	56.0%	13.9%	30.1%
badger	0.3%	0.0%	99.7%
json	0.0%	0.0%	100.0%
scheck	2.6%	49.7%	47.7%
slayout	0.5%	0.1%	99.4%

The breakdown of their contribution in terms of total space reclaimed is shown in table 7. Their contribution vary a lot depending on the application. For example, the Go compiler uses a lot of slices to hold basic blocks temporarily during compilation, so it benefits a lot from `FreeSlice()`. Applications that avoid a certain data structure cannot benefit from it.

6.7 Compilation Speed

To show that GoFree meets its design goal of maintaining Go's compilation speed, we compiled the `ssa` package, a relatively large package of the Go compiler, 99 times with the original Go compiler and GoFree. The values are similar and the p-value is 0.4963.

6.8 Robustness

To test whether GoFree corrupts the memory i.e., deallocates any living object, we ran Go's official package tests using a mock implementation of `tcfree`. Instead of deallocating the memory according to the strategies described in section 5, this mock implementation sets the memory to zero, or flips all the bits. This reveals wrong deallocation faster because it causes any later usage of the freed space to get an incorrect value instead of possibly getting the correct value if the slot is not immediately re-allocated. In multiple runs, GoFree with the mock `tcfree` implementation successfully passed all the tests, showing the robustness of our approach.

⟨Mike: **How much overhead does GoFree impose compared to manual deallocation? Cite these papers: [25, 33, 42].**⟩

7 Conclusion

This paper presented GoFree, an $O(N^2)$ static analysis that identifies short-lived heap objects that are capable of bypassing GC via explicit freeing, without increasing the time complexity of escape analysis. The approach identifies complete points-to and performs lifetime analysis upon them. Key ideas include offload complexity from static analysis to the runtime system, allowing tentative free primitives for optimization and concurrency, and identifying when a conservative analysis performs precisely. These ideas are applicable to any language that supports garbage collection and either region allocation or explicit deallocation [4, 12–14, 18, 23, 24, 27, 30, 31]. We implemented the approach for Go and examined how its modern language features pose both challenges and advantages for explicit free analysis. Our evaluation of

GoFree performs static scanning and runtime profiling on a set of Go programs, illustrating its effectiveness.

References

[1] Alex Aiken, Manuel Fähndrich, and Raph Levien. 1995. Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages. In *PLDI '95: Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*. La Jolla, CA, USA, 174–185.

[2] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language*. Ph. D. Dissertation. Citeseer.

[3] Richard Bellman. 1958. On a routing problem. *Quarterly of applied mathematics* 16, 1 (1958), 87–90.

[4] Daniil Berezun and Dmitri Boulytchev. 2014. Precise garbage collection for C++ with a non-cooperative compiler. In *CEE-SECR 2014: Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia*. Moscow, Russia, Article 15.

[5] Bruno Blanchet. 1999. Escape analysis for object-oriented languages: application to Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, USA). 20–34.

[6] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin Rinard. 2003. Ownership types for safe region-based memory management in real-time Java. In *PLDI 2003: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. San Diego, CA, USA, 324–337.

[7] Steve Carr and Ken Kennedy. 1994. Scalar replacement in the presence of conditional control flow. *Software: Practice and Experience* 24, 1 (1994), 51–77.

[8] Hao Chen and Hee-Jong Suh. 2012. An improved Bellman-Ford algorithm based on SPFA. *The Journal of the Korea institute of electronic communication sciences* 7, 4 (2012), 721–726.

[9] Sigmund Cherem and Radu Rugina. 2007. Uniqueness inference for compile-time object deallocation. In *Proceedings of the 6th International Symposium on Memory Management*. 117–128.

[10] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C Sreedhar, and Sam Midkiff. 1999. Escape analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, USA). 1–19.

[11] Jong-Deok Choi, Manish Gupta, Mauricio J Serrano, Vugranam C Sreedhar, and Samuel P Midkiff. 2003. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems* 25, 6 (2003), 876–910.

[12] Matthew Davis, Peter Schachte, Zoltan Somogyi, and Harald Søndergaard. 2013. A low overhead method for recovering unused memory inside regions. In *MSPC 2013: Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. Seattle, WA, USA, Article 4.

[13] Ulan Degenbaev, Jochen Eisinger, Kentaro Hara, Marcel Hlopko, Michael Lippautz, and Hannes Payer. 2018. Cross-component garbage collection. In *OOPSLA 2018, Object-Oriented Programming Systems, Languages, and Applications*. Boston, MA, USA, Article 151.

[14] Ulan Degenbaev, Michael Lippautz, and Hannes Payer. 2019. Garbage collection as a joint venture. *CACM* 62, 6 (June 2019), 36–41.

[15] Alain Deutsch. 1997. On the complexity of escape analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 358–371.

[16] L Peter Deutsch and Daniel G Bobrow. 1976. An efficient, incremental, automatic garbage collector. *Commun. ACM* 19, 9 (1976), 522–526.

[17] Fanding Duan. 1994. A faster algorithm for shortest-path—SPFA. *Journal of Southwest Jiaotong University* 29, 2 (1994), 207–212.

[18] Martin Elsman. 2003. Garbage collection safety for region-based memory management. In *TLDI 2003: The ACM SIGPLAN Workshop on Types in Language Design and Implementation*. New Orleans, LA, USA, 123–134.

[19] David Gay and Bjarne Steensgaard. 2000. Fast escape analysis and stack allocation for object-based programs. In *Proceedings of the International Conference on Compiler Construction*. Springer, 82–93.

[20] Google. 2022. Go Official Frequently Asked Questions. https://go.dev/doc/faq#stack_or_heap.

[21] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. Berlin, Germany, 282–293.

[22] Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. 2006. Free-Me: A Static Analysis for Automatic Individual Object Reclamation. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada). ACM, New York, NY, USA, 364–375. https://doi.org/10.1145/1133981.1134024

[23] Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. 2006. Free-Me: a static analysis for automatic individual object reclamation. In *PLDI 2006: Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*. Ottawa, Canada, 364–375.

[24] Niels Hallenberg, Martin Elsman, and Mads Tofte. 2002. Combining region inference and garbage collection. In *PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. Berlin, Germany, 141–152.

[25] Matthew Hertz and Emery D. Berger. 2005. Quantifying the performance of garbage collection vs. explicit memory management. In *OOPSLA 2005, Object-Oriented Programming Systems, Languages, and Applications*. San Diego, CA, USA, 313–326.

[26] JetBrains. 2021. Go Programming - The State of Developer Ecosystem 2021. https://www.jetbrains.com/lp/devecosystem-2021/go/.

[27] Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang. 2023. Modular verification of safe memory reclamation in concurrent separation logic. In *OOPSLA 2023, Object-Oriented Programming Systems, Languages, and Applications*. Cascais, Portugal, Article 251.

[28] William Kennedy. 2018. Escape-Analysis Flaws. https://www.ardanlabs.com/blog/2018/01/escape-analysis-flaws.html.

[29] Kyungwoo Lee and Samuel P Midkiff. 2006. A two-phase escape analysis for parallel Java programs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 53–62.

[30] Maged M Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *TPDS* 15, 6 (June 2004), 491–504.

[31] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. 2009. Precise garbage collection for C. In *ISMM 2009: International Symposium on Memory Management*. Dublin, Ireland, 39–48.

[32] Paul Menage Sanjay Ghemawat. 2007. TCMalloc: Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[33] Kunal Sareen and Stephen Michael Blackburn. 2022. Better understanding the costs and benefits of automatic memory management. In *MPLR 2022: 19th International Conference on Managed Programming Languages & Runtimes*. Brussels, Belgium, 29–44.

[34] Ran Shaham, Eran Yahav, Elliot K Kolodner, and Mooly Sagiv. 2003. Establishing local temporal heap safety properties with applications to compile-time memory management. In *Proceedings of the International Static Analysis Symposium*. Springer, 483–503.

[35] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial escape analysis and scalar replacement for Java. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*. 165–174.

[36] Mads Tofte and Jean-Pierre Talpin. 1994. Implementation of the typed call-by-value *lambda*-calculus using a stack of regions. In *POPL '94: Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Portland, OR, 188–201.

[37] Frédéric Vivien and Martin Rinard. 2001. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. 35–46.

[38] Cong Wang, Mingrui Zhang, Yu Jiang, Huafeng Zhang, Zhenchang Xing, and Ming Gu. 2020. Escape from escape analysis of Golang. In *Proceedings of the 42nd Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*. 142–151.

[39] John Whaley and Martin Rinard. 1999. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*. 187–206.

[40] Yu Zhang, Lina Yuan, Tingpeng Wu, Wen Peng, and Quanlong Li. 2010. Just-in-Time Compiler Assisted Object Reclamation and Space Reuse. In *Proceedings of the 2010 IFIP International Conference on Network and Parallel Computing*. Springer-Verlag, Berlin, Heidelberg, 18–34.

[41] Ekaterina Zharova. 2021. The state of Go. https://blog.jetbrains.com/go/2021/02/03/the-state-of-go/.

[42] Benjamin Zorn. 1993. The measured cost of conservative garbage collection. *Software: Practice and Experience* 23, 7 (1993), 733–756.