A Low Overhead Method for Recovering Unused Memory Inside Regions

Matthew Davis

NICTA Victoria Laboratories and
Department of Computing and Information Systems,
The University of Melbourne, Victoria 3010, Australia
davis@student.unimelb.edu.au

Peter Schachte, Zoltan Somogyi,
and Harald Søndergaard

Department of Computing and Information Systems
The University of Melbourne, Victoria 3010, Australia
{schachte,zs,harald}@unimelb.edu.au

Abstract

Automating memory management improves both resource safety and programmer productivity. One approach, region-based memory management [9] (RBMM), applies compile-time reasoning to identify points in a program at which memory can be safely reclaimed. The main advantage of RBMM over traditional garbage collection (GC) is the avoidance of expensive runtime analysis, which makes reclaiming memory much faster. On the other hand, GC requires no static analysis, and, operating at runtime, can have significantly more accurate information about object lifetimes. In this paper we propose a hybrid system that seeks to combine the advantages of both methods while avoiding the overheads that previous hybrid systems incurred. Our system can also reclaim array segments whose elements are no longer reachable.

Categories and Subject Descriptors D.3.4 [*Processors*]: Memory management (garbage collection)

General Terms Algorithms, design, languages, performance

Keywords Region-based memory management, garbage collection

1. Introduction

Many programming languages, such as C and C++, require that the programmer manage memory manually by explicitly reclaiming storage for data that are no longer needed. This requirement demands that programmers maintain a global and detailed awareness of their usage of data so they can reclaim each memory item when no longer needed. Programmers frequently make mistakes in this, either failing to reclaim an item that is no longer used, causing a *memory leak*, or reclaiming an item that is still needed, causing memory corruption. Automatic memory management aims to improve resource safety and improve programmer productivity by relieving the programmer of the burden of explicitly managing memory, and removing the possibility of human error.

Automated memory management systems can be implemented using either region-based memory management (RBMM) or garbage collection (GC). RBMM works by grouping allocated items together into *regions* and then freeing entire regions in a single quick operation. A set of items is grouped together into a region if they refer to each other and/or have similar lifetimes. This potentially improves locality and hence cache performance. Garbage collection works by periodically recursively tracing through all items reachable from any of the live variables in the program. All unreachable items can safely be reclaimed.

A comparative analysis of the two approaches is complex, but the big picture is that there is a time/space trade-off between them. Since an RBMM system does not require a scan of the program's memory at runtime, it can result in a faster running executable than a GC system. However, since RBMM groups items into regions, and it can reclaim only whole regions and not individual items, an RBMM system will be forced to keep inaccessible (dead) items resident in memory if they reside in the same region as any accessible (live) items. In certain cases, a region can grow very large even though it contains few live items, resulting in the program consuming much more memory than needed.

However, the situation is more complicated than that. *Ideal* RBMM should not only produce smaller execution times, but also realise a potential to use less memory than GC. This potential is due to two facts: (1) RBMM needs considerably less memory for its own bookkeeping, and (2) RBMM can decide what memory to free based on what the program will need in the future, rather than simply on what it can currently access. In principle, RBMM should be able to release memory in relatively small chunks, resulting in a flatter memory usage profile. However, there will always be programs exhibiting behaviours that favour garbage collection.

In this paper we consider a way of getting the benefits of both approaches by combining them. To achieve the performance benefits of RBMM, while also avoiding the problem of memory bloat, we garbage collect inaccessible items within regions. We make three contributions:

1. We propose a *new way of combining GC and RBMM*, with *less overhead* than similar systems [7].
2. Our algorithms support *partial* collections: recovering memory from *some* regions, but not all.
3. We enable the *collection of segments of arrays*. In languages that support *slices*, it can happen that some elements of an array are live and others are not. We show how to recover the memory occupied by the dead elements, at a low cost.

We are in the process of implementing our algorithms in a compiler for Go. Some of the paper's discussions, especially about arrays and slices, are conducted with a Go implementation in mind, but all of the techniques we discuss should be applicable to all statically typed languages, possibly with some minor modifications.

MSPC'13 June, 2013, Seattle, Washington

The rest of the paper is structured as follows. Section 2 presents the background for the paper, discussing GC and RBMM in more depth. Section 3 presents the trade-offs between GC and RBMM in more depth, motivating the present work. Sections 4, 5, and 6 lay out our proposed hybrid system. Finally, Section 7 discusses related work, and Section 8 concludes.

2. Background

Understanding how both approaches to automated memory management work is critical in understanding how we can mix the two. Both approaches try to approximate the ideal of an automatic system that reclaims each memory item *immediately* after it is last accessed, with *zero* time overhead. This ideal is not achievable, because which access to each memory item will be the last is in general unpredictable, and because reclaiming items from the heap will always have a nonzero cost.

2.1 Garbage collection

A GC system conservatively approximates the set of items which will not be accessed again as the set of items which are inaccessible to the program *now*. It determines this by scanning all accessible items, beginning with the *root-set*, the set of all variables reachable on continued execution of the program, and recursively following all pointers to other items. Any items not reached by this scan are considered dead, *i.e.* garbage, and their memory is reclaimed. Runtime memory scans can be time-consuming, and often require a pause of program execution to ensure that no state change takes place during the scan.

2.1.1 Non-moving and moving garbage collectors

There are two approaches to reclaiming items determined to be garbage. A *non-moving* collector links all garbage items together in a freelist; future allocations are then taken from this list. In contrast, a *moving* collector consolidates all the non-garbage items, typically into a small number of contiguous extents; the remaining memory is then free to be allocated later.

Moving collectors have several advantages. First, they allow memory to be quickly allocated by simply advancing a pointer. Second, they give greater locality of reference. Third, they naturally defragment free memory as they consolidate items; non-moving collectors must explicitly do this as a separate operation. Finally, the time taken to consolidate non-garbage items is proportional to the amount of non-garbage, while the time to link together the garbage items is proportional to the amount of garbage. In a well-tuned GC system, the amount of non-garbage will *usually* be small compared to the amount of garbage.

2.1.2 Conservative and type-accurate collectors

As it scans memory items, the GC system must determine which values are pointers to memory items and which are something else (such as primitive values). A *conservative* collector makes this decision by looking at the value. Since a bit pattern that matches an address in a part of the heap managed by the GC system *could* be a pointer, conservative collectors treat it as a pointer, and keep alive the item it points to. A *type-accurate* collector maintains type information about every variable and every structure type, and uses *this* to decide which values are pointers.

Conservative collectors are generally somewhat simpler, since they do not need to consider types, and therefore do not need the cooperation of a compiler to provide type information. They are also applicable to weakly typed languages, such as C, in which (due to casts) values present at runtime may not reflect the declared types of the variables holding those values. However, conservative collectors can mistake for pointers integers and other non-pointer values whose bit patterns, when viewed as pointers, happen to represent pointers into the heap. Such mistakes can accidentally preserve an item, *and all the other items reachable from it*, which may collectively represent a large amount of memory. Another important drawback is that since they cannot be certain whether a bit pattern they treat as a pointer *actually is* a pointer, they cannot update the bit pattern, which means that they cannot be moving collectors.

2.2 Region-based memory management

RBMM reduces the cost of automatic memory management in two ways. First, it eliminates the need to scan memory to determine which items to reclaim. This decision is made at compile-time by analysing the program to determine when items will no longer be accessed. Second, it reduces the cost of reclaiming items by grouping multiple items together in a single region to be reclaimed in a single, quick operation.

The analyses to determine when each region should be created, when it should be reclaimed, and from which region to make each allocation, are the most complex part of an RBMM system. The results of the analysis are then given to a program transformation that modifies the program to actually insert the operations to create and to destroy regions at the points indicated by the analysis, and to set up all the memory allocations to happen from the indicated regions. Since a function may need to work with different regions when called from different places, these modifications usually also require the addition to most functions of parameters identifying the regions to be allocated from and to be reclaimed.

The runtime part of an RBMM system consists of functions that create a region, allocate memory for an item from a region, and destroy a region. Since the amount of memory to be stored in a region is rarely known in advance, RBMM systems typically store each region as a linked list of contiguous chunks of memory. When the region is created, the RBMM system allocates a chunk of memory for it. Each allocation from the region hands out some of this memory to the requestor. When the amount of memory left in the current chunk is not enough for a request, the RBMM system gets another chunk of memory, and adds it to the list of chunks in the region. There are two sources of these chunks of memory: the OS, and the freelist. The freelist contains chunks of memory that were parts of regions that have since been deallocated, and whose memory is therefore available for reuse.

Most RBMM systems prefer to make all chunks the same size, since this makes such reuse as simple as possible, by avoiding external fragmentation. The typical size of a chunk is the size of a page in the hardware's virtual memory system, which is typically 4 or 8 KB, so these chunks are usually called *region pages*. However, some items may be bigger than a page. (This happens very rarely when the item is a single structure, but is reasonably common when it is an array.) Most RBMM systems therefore allow chunks to have any size that is a multiple of the system page size. The restriction to integer multiples allows them to require that the start address of every chunk be a multiple of the page size, which considerably reduces (though it does not eliminate) the problem of external fragmentation in the free memory, To handle allocations that are bigger than a page, these systems allocate a new chunk whose size is the requested size rounded up to the next multiple of the page size. In systems like this, each chunk is either a single page or a contiguous sequence of pages, so we call the chunks *flexipages*.

Each region needs a small amount of housekeeping information, such as a pointer to the current flexipage and a pointer to the start of the free memory in that flexipage. This information is kept in a *region header*, which is typically stored at the start of the first flexipage of the region. At runtime, each region is identified by a pointer to its header. Each flexipage also needs a small header

which contains its size and a pointer to the previous flexipage in the region (if any).

Some RBMM systems, including the one presented in [9], impose a stack discipline on regions: the last region allocated must be the first one freed. Our system does not impose this requirement.

3. Our motivation

Because the lifetime of a memory item is in general undecidable, region analysis must conservatively approximate it. In some cases, the approximation is too conservative, creating long-lived regions in which many items are no longer needed. In particular, items referred to by global variables are placed in a region which will be kept until the program exits. Several previous studies [2, 6] have shown that such long-lived regions can accumulate large numbers of now-dead objects beside some live ones, increasing the program's memory footprint significantly, and in some cases beyond the limit of acceptability.

In earlier work [3], we treated the region containing global variables specially, managing it with Go's existing built-in GC system, but this approach has two shortcomings. First, it does nothing to reclaim unused items in other long-lived regions, and second, it leaves us with two non-interoperable memory management systems. The reason why the second point matters is that it prevents us from implementing an optimization we believe may be important. In certain cases, one code path may permit two regions to be kept separate, while a less common code path may require the analysis to consider them to be the same region. Keeping them separate may permit one region to be reclaimed much earlier than the other, so we would prefer to do this. However, we cannot do this if we cannot merge the two regions at runtime, and indeed that is the case when one of the "regions" is managed by Go's builtin GC system.

Thus the goal of this work is to create a RBMM system that allows the contents of regions (especially long-lived regions) to be garbage collected. We still expect that most regions will be short lived, and that most memory cells will be recovered without GC, when the regions containing them are removed. Since we expect GC operations to be the exception and not the rule, we want region operations (the creation and destruction of regions, and allocation of memory from regions) to be as fast as they are in our current RBMM system; the existence of the GC system should not have a significant impact on the performance characteristics of regions that are never garbage collected. A secondary goal is to make our GC system a moving collector, to improve locality of reference.

Section 4 presents our basic approach. To allow us to explain it clearly, we simplify the problem by assuming the absence of several Go language constructs, namely arrays, interface types, and maps. This assumption gives us two restrictions. The first ensures that all values of the same type have the same size, and the second lets us know the type of each variable at compile time. In Section 5, we extend our system to handle arrays, lifting the first restriction. Section 6 briefly discusses interface types and maps, lifting the second restriction.

4. Managing ordinary structures

As mentioned above, we want our garbage collector to be a moving collector. Such a collector needs to know which parts of each item are pointers and which are not. The simplest way to give it this information is to include type information next to every item in every region [7]. Objects already have this information, but other items (such as ints, floats or non-object structures) do not. Adding a type description next to every item in a region would significantly increase memory consumption. Since each region will contain values from only a limited set of types, we can greatly reduce the space overhead of type information by storing the description of each type

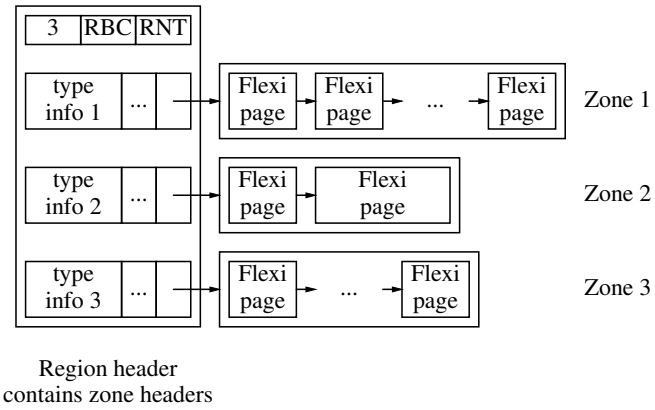

Figure 1. Region data structure

that can occur in the region just once, and associating all values of that type in the region with that description. In other words, we split each region into a set of *zones*, with one zone per type that can appear in the region.

If there are N types that can appear in a region, then this scheme costs us the memory occupied by N zone headers, each of which is 72 bytes in size. Typical values of N (from our test cases) range from 1 to 25, so the typical overhead ranges from 72 to 1,800 bytes per region. This is a fixed cost. On the other hand, the amount of memory that this schemes allows us to save scales with the amount of data in the region. For example, if a region contains 100,000 8 byte items (800 KB total) and 200,000 24 byte items (4.8 MB total), then, by not having to identify the type of each item with an 8 byte pointer to type information, we save 300,000 * 8 = 2.4 MB, which in this case represents 50% overhead. On other cases, the percentage will be different. However, it should be clear that our scheme saves not just significant amounts of memory, but also the time needed to fill in this memory.

While garbage collection systems have used type-specific zones before, to the best of our knowledge, this is the first time they have been applied to regions in a system using RBMM.

Region analysis can give us, for each of the regions it creates, the set of types whose values may appear in the region [3]. In fact, it is guaranteed to do so, unless the program uses language constructs (such as interfaces) that introduce polymorphism and thus hide the actual types of some values from the compiler. Until Section 6, we will assume that such constructs are absent, and that we *do* know the set of types in each region.

Figure 1 shows the effect of splitting a region into a set of zones, one for each type in the region, each zone holding all the items of that type in the region. Each zone consists of a list of flexipages, and has a header that contains the following slots:

- A pointer to the header for the whole region.
- A pointer to a description of the type of the items in the zone, what we call a *typeinfo*. These typeinfos are read-only data structures created by the compiler. Each typeinfo contains the size of items of that type, an indication of whether the type is a builtin type, if it is, which one, and if it is not, then whatever information our garbage collector needs to know about its components. For structures, this includes the number of fields, and for each field, its offset and a pointer to the typeinfo for its type.
- The number of items of this type that fit in a single page, *i.e.* in a flexipage of the minimum size. This is calculated from the page size, the size of flexipage headers, and the size of each item. We will show the exact formula later.

- A pointer to the start of the most recently allocated flexipage in the zone.
- During a collection, the pointer above defines the list of flexipages that act as the from-space. We also have a corresponding pointer that serves to define the to-space. This second pointer is used only during collections.

The region header contains:

- The number of zones in the whole region.
- Two bits that are needed only during collections. The REGIONBEINGCOLLECTED bit is set iff the collection is attempting to recover memory from the region, while the REGIONNEEDSTRACING bit is set iff the GC algorithm needs to traverse the contents of the region in order to find live items in the regions being collected.
- An array of the headers of the zones.

4.1 Creating a region

One of the jobs of the region transformation is to insert code to create regions *just before* the points in the program where the region analysis determines that those regions are first needed. The tasks of the code to be inserted are:

- to allocate memory for the header of the new region,
- to initialize all the components of the region header, and
- to return the address of the header.

As shown in Figure 1, the size of the region header is a simple function of the number of zones in the region.

Most RBMM systems put the region header at the start of the first flexipage of the region. We cannot do that, because a region with n zones effectively has n "first" flexipages. We could pick one, but we would have to treat that one differently from the others (for example, because that flexipage would have room for fewer items than all other flexipages in that zone). We sidestep these problems by allocating the region headers from a memory pool (Pool 2) that is *separate* from the pool that supplies the flexipages for zones (Pool 1).

To fill in the newly allocated region header, we copy into it the contents of a compiler-generated static data structure. The fields of region and zone headers must be filled in either with a fixed value (such as NULL for all the flexipage pointers) or with information that is known statically, such as the number of zones in the region. The trickiest fields are the typeinfo pointers. After the compiler picks a standard order for the types in the region, it knows the symbolic address of the typeinfo that the typeinfo-pointer field of each zone header should contain, because the compiler also generates exactly one typeinfo for each type. It can just put that symbolic address in the region header template, and pass the symbolic address of that template to the memory copy function. The linker will convert the symbolic addresses of both the typeinfo and the template into absolute addresses. Therefore the code that the region transformation inserts into the program to create a region looks like this:

```
rgn_hdr_ptr = create_region(template_ptr, size);
```

and the function that this calls looks like this:

```
create_region(template_ptr, size) {
    rgn_hdr_ptr = checked_malloc_pool_2(size);
    memcpy(rgn_hdr_ptr, template_ptr, size);
    return rgn_hdr_ptr;
}
```

4.2 Allocating from a region

In traditional RBMM systems, each allocation (the equivalent of a call to *malloc*) specifies in what region the new item should be allocated, by providing a pointer to the header of that region. In our system, the parameter list of the allocation function includes not just a pointer to the region header, but also the zone number that corresponds to the allocated item's type in that region. From that, the allocation function can look up the zone's header, and the typeinfo for the type, which gives the size of the item and thus the number of bytes to be allocated. The allocation function gets this number of bytes from the last allocated flexipage of the zone if it has room; if it does not, or if the zone has no allocated flexipages yet, it allocates a new flexipage and adds it to the zone first.

Determining which zones each region must have is an added responsibility of the compile-time region analysis. Note that this must be a global analysis, since different modules may require the inclusion of different types in each region. Further complicating matters, each allocation must specify a single offset in the region structure to find the appropriate zone for that allocation. This offset must be correct for every region that may be used for that allocation, so the offset for each type allocated in a function must be consistent among all regions that may be used for that allocation in that function. Ensuring this, while minimizing the number of zones in each region, is a complex optimization problem.

4.3 Reclaiming a region

Reclaiming a region is simple: we release every flexipage in every one of the region's zones back to Pool 1, and we release the region header back to Pool 2.

4.4 Finding typeinfos

Since we want to use a type-accurate (non-conservative) collector, we need to be able to find the type of an item from its address. To this end, we maintain a data structure we call the *zone-finder*, which is a variant of the *BIBOP* or *big bag of pages* idea [8]:

- Every item the collector needs to trace on the heap is stored in a flexipage of a zone of a region.
- Both the size and the starting address of every flexipage is an integer multiple of the standard page size. (That is, flexipages are aligned on page boundaries.)
- Conceptually, Pool 1, the pool from which flexipages are allocated, is a contiguous sequence of pages.
- We pair every page in Pool 1 with a *shadow* word in a new pool, Pool 3, which is the zone-finder.
  - If a page in Pool 1 is not currently in use, then the shadow word corresponding to it will be NULL.
  - If a specific page in Pool 1 is currently in use as the first page of a flexipage in zone z in region r, then its shadow word will point to the zone header for z. From there, we can reach both the header of region r and the typeinfo describing the type of the items stored in zone z of region r.
  - If a specific page in Pool 1 is currently in use as the non-first page of a flexipage in zone z in region r, then its shadow word will be a pointer to the shadow word corresponding to the first page of that flexipage, but tagged to indicate that it points to a shadow word rather than a zone header.

  Since zone headers and shadow words are both always stored at aligned addresses, we use the least significant bit as a tag to distinguish between the last two cases.

Conceptually, Pool 1 and Pool 3 are arrays with corresponding elements. However, if we want the pools to grow beyond their initially

Algorithm 1 Preserve data in an item

Require: *base*: The address of the start of an item to preserve
Require: *type*: The type of that item
Require: *fpp*: Points to the flexipage containing that item
Require: *zhp*: Points to header of the zone containing that item
  function PRESERVE(*base*, *type*, *fpp*, *zhp*)
    size ← SIZEOF(*type*)
    newbase ← ALLOCFROM(TOSPACE(*zhp*), *size*)
    COPYMEMORY(*newbase*, *base*, *size*)
    REDIRECTBIT(*fpp*, *base*) ← `True`
    ∗*base* ← *newbase* ▷ Set redirect pointer
    return *newbase*

allocated sizes, we must allow them to be stored noncontiguously. For our purposes, pretty much any of the many possible ways of simulating contiguous memory will do. Our implementation represents both Pool 1 and Pool 3 as a list of one *or more* contiguous sequences of pages, with a contiguous sequence of pages in Pool 3 for each contiguous sequence of pages in Pool 1.

4.5 Managing redirections

We garbage collect each zone using a semispace algorithm [5]; that is, we copy every live item out of the flexipages currently allocated to the zone (the from-space), into a fresh new set of flexipages (the to-space). When this traversal of live items arrives at an item, it needs to know whether that item has been copied to the to-space yet. (Copying a live item to the to-space several times would change the aliasing between items, which would be incorrect.) We need one bit per item for this information. These bits are required only during GC, and could thus be kept in temporary data structures, but the management of these data structures would take extra time. To avoid this and to keep the algorithm simple, we reserve space for these bits in each flexipage. The space cost is usually quite small, 1% or less: one bit per item, whose size is virtually always at least 64 bits, and most often 128 bits or more. Therefore the structure of each flexipage is:

- a fixed size flexipage header, which includes, amongst other things, the size of this flexipage and n, the number of items it contains,
- an array of n redirection bits, one bit per item,
- any padding required to align the following items, and
- an array of n items.

The formula for computing n and the number of bytes before the first item bi is:

$$n = \left\lfloor \frac{(bytes\_per\_flexipage - bytes\_per\_header) * 8}{1 + (bytes\_per\_item * 8)} \right\rfloor$$

$$bi = \left\lceil \frac{bytes\_per\_header + \lceil \frac{n}{8} \rceil}{alignment} \right\rceil * alignment$$

where all items begin at an address divisible by $alignment$.

Given the start address of a flexipage, address arithmetic can compute the location of the REDIRECTBIT for an item in that flexipage, and vice versa.

Between two collections, each REDIRECTBIT in each flexipage contains 0. When the traversal encounters a live item whose REDIRECTBIT is 0, it copies the item to the to-space, and sets its REDIRECTBIT to 1. To let later parts of the traversal know not just that the item *has* been copied but also *where* it has been copied to, the traversal also records the address of the item in to-space in the first word of the item. (It is ok to overwrite any part of the user data stored in the old copy of the item, since it will not be referred to anymore.) All this is shown in Algorithm 1.

Algorithm 2 Garbage collect from regions

Require: *Roots*: The set of root variables
Require: *GC_regions*: The set of regions to collect
  function GC(*Roots*, *GC_regions*)
    for all *rhp* ∈ *all_regions* **do**
      REGIONBEINGCOLLECTED(*rhp*) ←
        rhp ∈ *GC_regions*
      REGIONNEEDSTRACING(*rhp*) ←
        some region in *GC_regions* is reachable from *rhp*
    for all *root* ∈ *Roots* **do**
      PRESERVEANDTRACE(*root*, `True`)
    for all *rhp* ∈ *GC_regions* **do**
      for all *zhp* ∈ ZONESOF(*rhp*) **do**
        FREE(FROMSPACE(*zhp*))
        FROMSPACE(*zhp*) ← TOSPACE(*zhp*)
        TOSPACE(*zhp*) ← `nil`

Of course, this assumes that all items are big enough to hold a pointer. This is why our system allocates a word (the size of a pointer) even for requests that ask for less memory than that. It is not alone in this; virtually all other memory management systems do the same, including the usual implementations of malloc.

When a garbage collection cycle is complete, all the pages of all the flexipages of the collected regions are returned to the freelist of Pool 1. Before any flexipage is reused, all its bits will be set to zero, including its redirection bits.

4.6 Collecting garbage

The top level of our garbage collection algorithm is shown in Algorithm 2. Its first parameter is the root set, *i.e.* the set of all the registers, stack slots and global variables that may contain pointers to items in regions. (We start by making copies of the original register values in memory, and copy the possibly-redirected values back to the registers when we are done.) The second parameter specifies the set of regions from which this invocation of the collector should recover memory. This set *need not* be the set of all regions. If the entity controlling the collection process expects that some regions have very little garbage, it can omit them from *GC_regions*. A region left out of *GC_regions* will still be traversed (traced) by our algorithm if such traversal may lead to live items in regions which *are* in *GC_regions*, but

- the collector will not need space to store copies of all the live items in those regions, reducing memory requirements when those requirements are otherwise at their peak, and
- the collector will not need to spend any time copying all the live items in the regions to the to-space, and updating all the pointers to the moved items.

The algorithm starts by recording, in each region header, whether the region is being collected in this collection, and whether it needs to be traced.

After that, Algorithm 2 finds all items in the collected regions that are reachable from the roots, using Algorithms 3 and 4, which we discuss below. Together these algorithms preserve each live item in a collected region by copying it from its original location in a from-space flexipage of one of the region's zones to the to-space of that zone, which consists of its own list of flexipages.

Once all reachable items have been so copied, and the pointers to them updated to point to the copies, the algorithm releases the memory occupied by the zones' original flexipages (the from-space) and replaces the pointer to the from-space with the pointer to the to-space flexipage list.

Finding live items in the regions being collected and copying them to the to-space of their zone is done by Algorithm 3. The

Algorithm 3 Preserve an item and everything it keeps alive.

```
Require: addrptr: Pointer to the address of an item
Require: toplevel: Is the call coming from Algorithm 2?
  function PRESERVEANDTRACE(addrptr, toplevel)
    addr ← *addrptr
    if addr is in the heap then
      ⟨fpp, base, zhp⟩ ← LOOKUPHEAP(addr)
      type ← TYPEIN(zhp)
      offset ← addr − base
      rhp ← CONTAININGREGION(zhp)
      if ¬REGIONBEINGCOLLECTED(rhp) then
        newbase ← base                          ▷ Item is not moved
        needstrace ← REGIONNEEDSTRACING(rhp)
      else
        if ¬REDIRECTBIT(fpp, base) then
          newbase ← PRESERVE(base, type, fpp, zhp)
          *addrptr ← newbase + offset
          needstrace ← REGIONNEEDSTRACING(rhp)
        else
          newbase ← *base                       ▷ Get redirect pointer
          *addrptr ← newbase + offset
          needstrace ← False                    ▷ Has been traced already
    else if addr is not null then
                ▷ if addr is not in the heap, it must refer to a root
      if toplevel then
        newbase ← addr    ▷ Top level refs point to the start
        type ← TYPEOF(addr)
        needstrace ← True
      else
        needstrace ← False    ▷ A top level call will trace it
    if needstrace then
      TRACE(newbase, type)
```

PRESERVEANDTRACE function is invoked not with the address of the item it is to preserve and trace, but with a pointer to that address, so that if and when it needs to move the item, it can update the address that pointed to it. When it is invoked, *addrptr* will point either to a root (such as a global variable or a stack slot containing a pointer), or to a part of the heap that itself contains a pointer. The pointers to roots supplied by Algorithm 2 always point to the start of a root item, as promised by the *toplevel* = `True`; pointers supplied by tracing may point inside (*i.e.* not at the start of) items, as allowed by *toplevel* = `False`.

The value of *addr* may or may not point into the heap, which in our case means "into one of the regions". If it does, then we can use the data structures described in Section 4.4, represented here by the function LOOKUPHEAP, to find out the address of the flexipage containing the item at *addr*. From that, the function can use address arithmetic to compute *base*, the address of the start of the item (*addr* may point into the *middle* of the item). If s is the size of the items in the flexipage, then

$$base = fpp + bi + s * \left\lfloor \frac{addr - (fpp + bi)}{s} \right\rfloor$$

We need to know *base* because if we copy the item, we must copy *all* of it. If the item ends up moved, the updated pointer must point to the same offset within the item as it did before.

Given the flexipage pointer, LOOKUPHEAP can also use the zone-finder to find the identity of the zone containing the flexipage. We can then follow the pointer in the zone header to the header of the region containing it. If this region is *not* being collected, then the item will survive the collection, at its current address, without us doing anything (though we may still need to trace any pointers inside the item). If this region *is* being collected, then Algorithm 2 will free all the flexipages of all the zones of the region, and we must copy the item to the corresponding to-space, unless this has already been done. If the redirect bit says that it has not yet been done, then we call PRESERVE, the function in Algorithm 1, to copy it to the zone's to-space. PRESERVE returns the new address of the item, and we set the original pointer to the item to refer to the original offset from this new address. PRESERVE also records both the fact that the copying has been done (by setting the redirect bit corresponding to this item in its flexipage) and the address of the new home of the item (in the first word of the item). So the next time the traversal reaches this item, the redirect bit will tell us that we do *not* need to copy the item again, and that we can instead pick up the new address of the item from the first word in its old copy. In this case, the traversal will also have traced all the pointers inside the item, so we need not process them again. We can similarly skip the processing of the pointers inside the item if the item is in a region from which the regions being collected cannot be reached either directly or indirectly.

Algorithm 4 Trace an item and preserve all items it keeps alive

```
Require: base: Pointer to the start of the item
Require: type: The type of the item located at base
  function TRACE(base, type)
    for all aioff ∈ ADDRSINSIDE(type) do
      aiaddr ← base + aioff
      PRESERVEANDTRACE(aiaddr, False)
```

Since we do not garbage collect the places that may contain roots, *i.e.* the stack, the global variables, and the registers, we need not concern ourselves with protecting any item that is not in the heap against being moved. Since Algorithm 2 will eventually invoke Algorithm 3 on every root, we need not trace roots when we reach them by following pointers in items. This is just as well, since those pointers may point *inside* roots that are structures, and finding the starts of those structures would be far from trivial.

The last step of Algorithm 3 is to invoke Algorithm 4 on roots and items on the heap that (a) may contain pointers that lead, directly or indirectly, to live items in the regions being collected, and (b) are not known to have been traced before. The traced items may be pointers, for which the ADDRSINSIDE function should return the offset 0. Or they may be structures containing pointers, for which it should return the offset of all the pointer-valued fields inside the structure, whether they are fields of the structure itself or of its parts. We then traverse the items all these pointers point to.

Note that we trace the pointers in the version of the item in the to-space, not the from-space. That is because in the from-space, the first word of the item will have been overwritten by the redirect pointer (also called the forwarding pointer).

Figure 2 shows an example illustrating these algorithms. Figure 2(a) shows part of the memory as GC begins: the stack contains two pointers, `xptr` and `yptr`, that point to two structures in the same zone, which has one flexipage. These structures each contain one non-pointer, whose contents are irrelevant here, and one pointer, which in this case point to each other, so this is a cyclic data structure. Note the flexipage contains two garbage structs, and the redirection bits are all 0.

After Algorithm 2 determines which regions to collect, we process the root set. We start by calling PRESERVEANDTRACE with `xptr`. The struct this points to ("item 1") is on the heap, in a region to be collected, and the redirect bit for it is clear, so we PRESERVE it: we copy it to to-space, set its redirected bit in from-space, and overwrite the first word of the struct in from-space with a pointer to the new copy in to-space. We then update `xptr` to point to the new copy as well. This is shown in Figure 2(b).

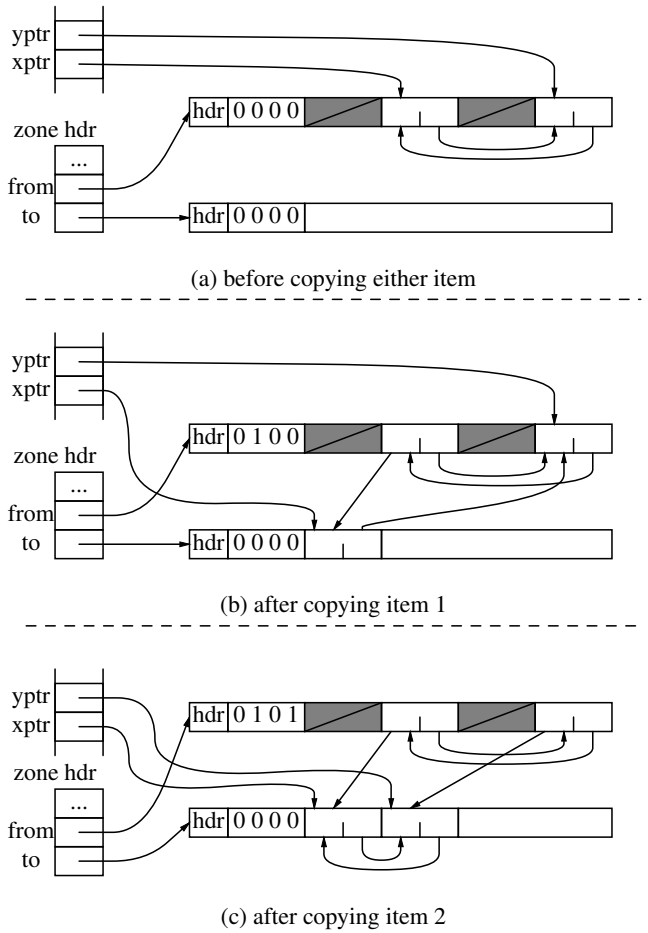

Figure 2. Example: copying two items to to-space

Next we call PRESERVEANDTRACE on the sole pointer in the freshly copied struct. Again this points to a struct ("item 2") on the heap, in a region to be collected, whose redirect bit is clear, so we PRESERVE it, and update the pointer we followed (in item 1) to point to the new copy. Next we TRACE the newly preserved struct, but this time the sole pointer points to a struct (item 1 again) whose redirected bit is now *set*. In this case we do not preserve or trace the struct, we just look up its new location in to-space, so we can use it to overwrite the pointer in item 2 (which used to point to item 1 in from-space).

When Algorithm 2 calls PRESERVEANDTRACE on the second root, `yptr`, it finds that the struct it points to, item 2, already has its redirect bit set. It will therefore update `yptr` to point to the new location of item 2 in to-space, but will not trace item 2 again. This will leave the state shown in Figure 2(c).

5. Managing arrays and slices

The *Go* language provides arrays, pointers to arrays, and slices. Arrays are fixed-size contiguous collections, and array pointers refer to fixed-sized collections as well, since the type of array pointers includes the number of elements in the array as well as the type of the elements. On the other hand, while the compiler knows the type of the elements of a slice, it does not know their number; the size of slices is dynamic. The Go implementation represents each slice as a structure holding a reference to some element of an array, as well as a capacity (the number of elements in the slice, starting at the pointed-to element), and a count (the number of initial elements in the slice that are meaningful). Thus slices are simply Go structures comprising three members, and, except in the optimization we describe in Section 5.2, we treat them as such.

5.1 Finding the start of an array

Some slices will point to elements in the middle of the target array. The PRESERVEANDTRACE function needs to know the start address of the item to be preserved. With scalar items, once we know the start address of a flexipage, we can use address arithmetic to convert the address of any part of an item into the address of the start of the item. However, since we put all arrays of a given type (regardless of length) into a single zone dedicated to arrays of that type, we cannot find the start of an array by address calculation.

Our solution to this problem is to prefix each array with a small header containing just its size. (Most memory management systems do this for every item; we do it only for arrays.) When tracing a pointer to or into an array, we can look up its address in the zone finder, which will give us a pointer to the start of the flexipage containing the array item. The first item in the flexipage starts just after the flexipage header. Given the start address of an item, *i.e.* the address of its header, the size allows us to calculate the address of the start of the next item in the flexipage, if there is one. So we can find the start address of the array that a pointer points into by traversing through the array items on the flexipage. The address we want is the last item start address we encounter in this traversal that is smaller than the pointer's value.

5.2 Preserving only the used parts of arrays

Our PRESERVEANDTRACE algorithm treats any reference to any part of an item as a reference to the entire item. A live variable whose type is an array or array pointer keeps all the elements alive. For slices, however, we can do better, provided live slices refer only to a *part* of the array that the slices were derived from, and there are no live references to the *whole* array. In such situations, we can reclaim the unneeded elements in such arrays, if we can modify the algorithm we use to trace slice headers. First we present how we handle slices; later we will return to discuss how array- and array-pointer-valued variables fit into our scheme.

The Go language semantics requires that if an array and slice, or two slices, shared the memory of some elements *before* a collection, they must also share those same elements *after* the collection. We therefore cannot copy live array elements individually; we must ensure that contiguous sequences of live array elements are copied to a new (possibly smaller) array in which they are still contiguous.

When tracing arrives at a slice header, we know that the array elements referred to by the slice are live. Unfortunately, we cannot know at that time whether the elements before and after these in the array are live or dead: it is possible that they have not yet been visited by the collector, but will be visited later. In general, we can know which elements of an array are live and which are dead only once a garbage collection has finished tracing all live data.

We could add an extra pass to the end of every collection, and defer the copying of array slices until this pass. However, this extra pass would add significant overhead, because not preserving an array when tracing it would reduce locality, and because we would need extra data structures to keep track of the deferred work. These data structures would occupy space during each collection, *exactly when free space is scarcest.* We therefore choose to use a conservative approximation: when we get to an array, we copy to to-space the set of array elements that were live at the end of the *last* collection. This means that an array element that becomes dead will survive one collection, but not two.

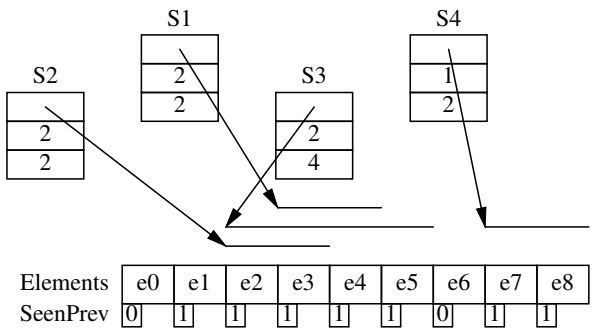

Figure 3. Copying only the previously live parts of arrays

This optimization needs more information attached to each array item than what we would need in its absence, which we just described in Section 5.1. This information consists of:

- ARRAYNUMBYTES, the number of bytes occupied by the item (as before).
- ARRAYNUMELTS, the number of elements in the array; redundant, as it could be computed from ARRAYNUMBYTES, but storing it avoids unnecessary recomputations.
- SEENPREV, an array of ARRAYNUMELTS bits. The bit is 1 iff the corresponding element was live at the end of the last collection. Initialized to 1 when the array is first created.
- SEENCURR, an array of ARRAYNUMELTS bits. The bit is 1 iff the corresponding element has been seen live during the current collection. Always initialized to 0; meaningful only during a collection.
- ELEMENTS, the elements of the array themselves.

Our optimization modifies Algorithm 3 so that when the collector traces a slice, it will invoke Algorithm 5 instead of Algorithm 4. This algorithm has a chance to avoid preserving unneeded elements of the array holding the slice's elements, but only if the array is stored on the heap. If it is stored in the rodata, data or bss section, then the array must be a global variable. This means that we need not take action to preserve its storage, and since that global variable is a root, that root either has already been or will be traced later, as an array (not as a slice). The array cannot be on the stack, because the Go compiler performs escape analysis, and this changes the storage class of any function-local array that a slice may ever refer to, converting it from stack allocated to heap allocated.

If the array holding the slice's data is on the heap, we use the algorithms of Section 5.1 (represented by function LOOKUPHEAPARRAY) to find the address of the flexipage storing the item, and from that, the start of the array item, and the zone containing that flexipage. From the addresses of the first elements of the slice and of the array, we can calculate *fse*, the index of the first slice element in the array. From that and the capacity of the slice, we can calculate *lse*, the index of the last slice element in the array.

Consider the situation when Algorithm 5 is invoked on slice header S1 in the example in Figure 3. This slice has a capacity and count of 2, and its data pointer points to the element at index 3 in the array. We will thus set *fse* to 3 and *lse* to 4. However, we cannot copy to to-space just the subarray containing only elements 3 and 4. For example, if we later see a reference to slice S2, which also has a capacity and count of 2, but its data pointer points to the element at index 2 in the array, the copies of the two slices must share the element corresponding to the index 3 in the original array.

The sequence of elements we may need to have contiguous in the copy is restricted to the neighboring elements that were live at the last collection. In our example, the SEENPREV bit vector for the array has a 1 in every position except the ones at indexes 0 and 6, so the first bit in the contiguous sequence of 1 bits that includes the 1 bits at positions 3 and 4 is at index $fce = 1$, and the last bit in that contiguous sequence of 1 bits is at index $lce = 5$. This is why we want to make sure that there is a copy in to-space of the subarray consisting of elements 1 to 5.

If all of the bits in SEENCURR starting from *fce* to *lce* are 0s, then the subarray has not yet been copied to to-space, so we do the copying then and there. After figuring out the amount of memory needed for the new array item, we reserve memory for it in to-space. We then fill in the array item's header, its SEENPREV and SEENCURR arrays, and finally the elements, which we copy from the original array. The SEENCURR array has all its bits set to 0s: those bits will be meaningful in the *next* collection, not this one. We set the SEENPREV bits in the to-space copy only for the array elements that this slice refers to, since so far these are the only elements that we know are live.

After we copy all the elements of the subarray from from-space to to-space, we overwrite the first word in the first element copied in from-space with the address of the copy in to-space. This ensures that later calls to TRACESLICE arriving at this subarray (*e.g.*, when TRACESLICE is invoked with S2) will know where the copy is.

Once we have preserved the data in the contiguous elements, we need to trace any pointers in the meaningful part of the slice. So we iterate over all those elements, tracing pointers in elements we have not traced before. Note that we set the SEENPREV bit corresponding to such items even if this call to TRACESLICE did not copy the subarray. In our example, this will happen when tracing the copy of element 2 during the invocation of TRACESLICE for slice S2. This will tell the *next* invocation of the collector that the elements at indexes 1, 2 and 3 are live in the copied subarray; these correspond to indexes 2, 3 and 4 in the original array.

The elements in slices that correspond to the difference between the COUNT and the CAPACITY (if there is one) do not contain data that the program may use, but they must be there, contiguous with the earlier elements, in case the program expands the slice. If there is a slice S3 that has a count of 2 but a capacity of 4, and its data pointer points to the element at index 2 in the array, then we must mark index 4 in the copy (index 5 in the original) as seen, because if we did not, then the collection *after* the next one would feel free recover its memory, which would prevent the correct operation of any expansion operations on S3.

Since the body of the function after the initial test can rely on the capacity of the slice being at least one, one of the loops that together iterate i from 0 to *cap* will set the SEENCURR bit for *fse*. Since *fse* is guaranteed to be in the range *fce*..*lce*, all later invocations of TRACESLICE on a slice that fits in that range will know that the subarray in that range has already been copied to to-space.

Just as our optimization must ensure that we call TRACESLICE instead of TRACE when tracing a slice header, we must handle two other cases specially as well. The first is arrays on the heap, or pointers to them. For these, we need to invoke a version of TRACESLICE that acts as if it was tracing a slice whose count and capacity are both the array size (in Go, this is available as part of the type of both arrays and pointers to arrays), the only differences being that (1) the capacity and count come from somewhere else, and (2) relocation must be reflected by an assignment to something other than DATA(*shp*). The second case is pointers to values that happen to point to or inside an array element. We can handle these as if we were looking at an one-element slice, though recording the relocation must be done differently yet again.

Since array elements can be of any type, the criterion that tells Algorithm 3 that it should call not TRACE but TRACESLICE (or its equivalents for arrays and array pointers) should *not* be the type of

Algorithm 5 Trace a slice header, and preserve and trace its slice

Require: *shp*: Address of the slice header
  function TRACESLICE(*shp*)
    if CAPACITY(*shp*) $= 0$ **then**
      return
    slicestart $\leftarrow$ DATA(*shp*)
    if *slicestart* is in the heap **then**
      $\langle base, zhp \rangle \leftarrow$ LOOKUPHEAPARRAY(*slicestart*)
      type $\leftarrow$ ELEMENTTYPE(TYPEIN(*zhp*))
      es $\leftarrow$ SIZEOF(*type*) ▷ Element size
      cap $\leftarrow$ CAPACITY(*shp*)
      fse $\leftarrow$ (*slicestart* $-$ &ELEMENTS(*base*, 0))/*es*
      lse $\leftarrow$ *fse* $+$ *cap* $-$ 1
      fce $\leftarrow$ *fse*
      while $0 \leq$ *fce* $- 1 \wedge$ SEENPREV(*base*, *fce* $- 1$)) **do**
        fce $\leftarrow$ *fce* $- 1$
      lce $\leftarrow$ *lse*
      while *lce* $+ 1 <$ *cap* $\wedge$ SEENPREV(*base*, *lce* $+ 1$)) **do**
        lce $\leftarrow$ *lce* $+ 1$
      copybase $\leftarrow$
        PRESERVEELEMENTS(*slicestart*, *fce*, *lce*, *fse*, *lse*, *es*)
      DATA(*shp*) $\leftarrow$ &ELEMENTS(*copybase*, *fse* $-$ *fce*)
      count $\leftarrow$ COUNT(*shp*)
      for $i \leftarrow 0$ **to** *count* $- 1$ **do**
        if SEENCURR(*base*, *fse* $+ i$) $= 0$ **then**
          SEENCURR(*base*, *fse* $+ i$) $\leftarrow 1$
          SEENPREV(*copybase*, *fse* $-$ *fce* $+ i$) $\leftarrow 1$
          eltbase $\leftarrow$ &ELEMENTS(*copybase*, *fse* $-$ *fce* $+ i$)
          for all *aioff* $\in$ ADDRSINSIDE(*type*) **do**
            aiaddr $\leftarrow$ *eltbase* $+$ *aioff*
            PRESERVEANDTRACE(*aiaddr*, `False`)
      for $i \leftarrow$ *count* **to** *cap* $- 1$ **do**
        SEENCURR(*base*, *fse* $+ i$) $\leftarrow 1$
        SEENPREV(*copybase*, *fse* $-$ *fce* $+ i$) $\leftarrow 1$

Algorithm 6 Preserve slice elements

Require: *slicestart*: Starting address of slice data
Require: *fce*: Index of the first contiguous element
Require: *lce*: Index of the last contiguous element
Require: *fce*: Index of the first slice element
Require: *lce*: Index of the last slice element
Require: *es*: The size of each element
  function PRESERVEELEMENTS(*slicestart*, *fce*, *lce*, *fse*, *lse*, *es*)
    $\langle base, zhp \rangle \leftarrow$ LOOKUPHEAPARRAY(*slicestart*)
    if $\forall i \in$ *fce*..*lce* . $\neg$SEENCURR(*base*, i) **then**
      numelts $\leftarrow$ *lce* $-$ *fce* $+ 1$
      copybytes $\leftarrow \left\lceil \frac{headerbytes + \lceil (2 * numelts)/8 \rceil}{alignment} \right\rceil * alignment$
        $+$ *numelts* $*$ *es*
      copybase $\leftarrow$ ALLOCFROM(TOSPACE(*zhp*), *copybytes*)
      ARRAYNUMBYTES(*copybase*) $\leftarrow$ *copybytes*
      ARRAYNUMELTS(*copybase*) $\leftarrow$ *numelts*
      for $i \in 0..$*numelts* $- 1$ **do**
        SEENPREV(*copybase*, i) $\leftarrow$ *fse* $\leq$ *fce* $+ i <$ *lse*
        SEENCURR(*copybase*, i) $\leftarrow 0$
      COPYMEMORY(&ELEMENTS(*copybase*, 0),
        slicestart, *numelts* $*$ *es*)
      $*$(&ELEMENTS(*base*, 0) $+$ *fce* $* sz$) $\leftarrow$ *copybase*
    else
      copybase $\leftarrow *$(&ELEMENTS(*base*, 0) $+$ *fce* $* sz$)
    return *copybase*

the item being traced, but a property of the zone that contains it. The obvious property to test is "does this zone contain array data". However, the approach we described in this section add both space and time overheads. If arrays in a zone typically die all at once, then we would not want to incur these overheads, because they would not pay for themselves through the earlier recovery of the memories of array elements. If either the programmer or a profiling system can predict which zones fall into which category, they can control whether the algorithms of this section are applied to each zone by including a bit in the headers of zones containing arrays that tells the algorithms operating on the zone's flexipages, including Algorithm 3, which item representation the zone uses, and therefore whether they should call TRACESLICE, or just a version of TRACE adapted to the simpler data structures described in Section 5.1.

6. Handling other Go constructs

Go provides several features we have not yet discussed. Space limitations prevent us from discussing them all in detail, but a few deserve mention.

Interface types in Go might be expected to present something of a problem, since values declared in a function as having an interface type actually have some other type which is not known when the function is compiled. However, our scheme handles interface types without adaptation. User code that deals with the item *without* knowing its actual type, knowing only what interface it implements, never needs to know what zone the item is stored in. However, when an instance of an interface type is created, its actual type must be known, so it will naturally be placed in the correct zone. When tracing an interface type object, our functions will find the flexipage and the zone it occurs in, and will determine its actual type from that before preserving and tracing it.

An interface type in effect stands for all the types that implement all the methods of that interface. In some cases, this may lead to regions with many zones, one for each actual type that is passed to a function or a set of functions expecting an interface type, with many of these zone containing very few items. In such cases, much space will be wasted on flexipages with few inhabitants. An alternative approach would have the region inference algorithm put items that are used as values of interface types into a special zone in each relevant region, a zone in which each item contains a tag. This would trade slower GC and higher per-item memory overhead for a lower per-actual-type overhead. Determining which of these two approaches is better, and under what circumstances, is a matter for future work.

Go's co-routines (*goroutines*) present more of a problem. The static analyses used to control the RBMM system assume that a called function runs to completion before the next function is called. The `go` construct violates that assumption: the function call following the `go` keyword will typically not be complete before the start of the execution of the following construct. This means in some cases, it is necessary to decide *dynamically* when to reclaim a region. Go also supports *channels* for communication between goroutines. These must be augmented to pass regions along with the items they contain, so that the receiver can know which regions to allocate from and which to reclaim. Finally, during GC, any thread that may use or modify any items in any regions being collected must be stopped for the duration of the GC. Note, however, that threads that cannot access any regions being collected *may* be allowed to continue during GC. For further discussion, see Davis *et al*. [3].

There are aspects of the Go implementation, such as strings and maps, that use specialized data representations. The algorithms that we have presented in this paper need minor adaptations to handle these representations.

7. Related work

Tofte and Talpin [9] introduced RBMM for Standard ML. Their seminal idea was to group data allocations into *stacks* of regions based on their lifetimes. The stack discipline requires a region to live at least as long as all other regions created after it, even if the data in it is no longer needed. They found that their RBMM system usually led to a smaller maximum resident memory size, while garbage collection was often faster. Aiken, Fähndrich and Levien [1] observed that significantly better results were possible by liberating region lifetimes from stack discipline.

Cyclone [6] is a safety-enhanced C variant that uses semi-automatic region-based memory management. The programmer can specify from which named region a particular allocation should be made. Cyclone allows the combination of regions and GC the same way our previous system [3] did: by setting aside a special "heap" region that may optionally be managed by the Boehm-Demers-Weiser collector. GC does not take place for other regions, and items on the heap are exclusively managed this way. Grossman *et al*. found that for some benchmark programs, their system suffers from the problem that motivated this paper: that some (non-heap) regions can grow much too large.

Berger, Zorn, and McKinley [2] had observed this same effect in their thorough investigation of manual memory management. This looked at both custom and general-purpose allocators, with some of the custom allocators being based on regions. They also presented a new system called a reap allocator, which combines the ideas of general purpose and region allocators. A reap works as a region until the programmer manually frees some items in it, after which the memory of those items becomes a free-list for the region. Space for new allocations in the region comes from the free-list until it becomes empty, at which point the system reverts to allocating from the end of the region. This is effectively a manual simulation of a system that automatically garbage collects the contents of regions.

Hallenberg, Elsman and Tofte [7] extend a stack based RBMM system with a copying garbage collector using Cheney's algorithm. Unlike our approach, their GC system requires adding a one-word tag to each memory item. Their testing showed that adding tags increased memory usage by as much as 61%, and slowed their RBMM-only system by up to 30%. (Our system's memory overhead should be *much* lower, though we cannot yet say anything about time overheads.) They found that adding RBMM to their original GC system improved execution speed by up to 42%, even though it required adding tag words. This improvement comes from being able to free a significant fraction of allocated memory cells *without* the expensive memory scans required by GC. We expect that our system should be able to get a similar performance advantage over Go's native garbage collector.

Hallenberg, Elsman and Tofte's paper deals with arrays simply by storing them *outside* region pages, in memory areas allocated via `malloc`, which are freed only when the whole region is freed. Their system never copies these areas, though it does have to traverse them during a collection to find the live objects they refer to. Compared to our system, this saves the cost of copying, but also eliminates the possibility of recovering the memory of unused arrays and unused array elements. Without a working system, we cannot say which approach is better, but it is very likely that the answer will depend on the behavior of the program, such as how often it takes partial slices of arrays.

Elsman [4] investigates type safety in the combined RBMM and copy-collector system implemented for Standard ML[7]. His combined system allows pointers between regions. However, having a older region point to an newer one poses a problem. When the newer region is popped off of the region stack, the pointer in the older region will be a dangling pointer, referencing newly-reclaimed memory. The author introduces pointer safety and proves the soundness of their implementation by eliminating dangling pointers in their region typing system. Although our system does not impose a stack discipline on regions, it relies on our region inference algorithm providing a similar guarantee.

8. Conclusion

In this paper we have presented an automatic memory management system combining the fast allocation and deallocation of memory under RBMM with the relatively low peak memory usage of a garbage collector. It requires no programmer annotations. To improve locality of reference, we want to use a copying collector, so we need to know the type of each item. Instead of attaching type tags to individual memory items, as done by Hallenberg, Elsman and Tofte [7], we attach them to pages, similar to the *Big Bag of Pages* approach to GC. The system can decide which region or regions to garbage collect based on runtime memory usage. Using compile-time region points-to information, it can also avoid tracing regions that cannot point to regions being collected. This can reduce GC times.

We have also presented a scheme for garbage collecting unused elements in arrays. These can arise as slices and slices of slices are taken, old arrays and slices go out of use, and new ones continue to be used. Our method reclaims unneeded elements over two successive GCs, with each GC retaining the elements that were live at the time of the previous GC, and determining the elements needed now. This capability costs two bits per slice element and increases runtime overheads slightly, but can potentially reclaim a substantial amount of additional memory. It can be switched off selectively in zones where it proves ineffective.

Future work includes completing the implementation of this system and benchmarking it. We must also develop heuristics to determine when and which regions to GC. When the system is complete, we expect to make it available as open source software.

References

[1] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proc. PLDI'95*, pages 174–185. ACM, 1995.

[2] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *OOPSLA 2002*, pages 1–12, 2002.

[3] M. Davis, P. Schachte, Z. Somogyi, and H. Søndergaard. Towards region-based memory management for Go. In *Proc. MSPC'12*, pages 58–67. ACM, 2012.

[4] M. Elsman. Garbage collection safety for region-based memory management. In *Proceedings of the SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 123–134, New Orleans, Louisiana, 2003.

[5] R. R. Fenichel and J. C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12(11):611–612, 1969.

[6] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proc. PLDI 2002*, pages 282–293, 2002.

[7] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *Proc. PLDI'02*, pages 141–152. ACM, 2002.

[8] G. Steele. *Data Representations in PDP-10 MacLISP.* AI Memo. Artificial Intelligence Laboratory, MIT, 1977.

[9] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proc. POPL'94*, pages 188–201, Portland, Oregon, 1994. ACM.