A Concurrent Garbage Collector for the Glasgow Haskell Compiler

Ben Gamari (Well-Typed LLP)

Contents

We propose a design for an incremental garbage collector for the Glasgow Haskell Compiler. The goal of this design is to reduce maximum pause times for major garbage collections while maintaining throughput similar to the current collector in both serial and parallel programs. Furthermore, the design aims to preserve GHC's existing heap organization to minimize the invasiveness of the implementation.

1 Background

Currently, GHC uses a generational, moving collector [Ungar1984] based on a block-structured heap [Marlow2008]. During collection under this scheme, live heap data is copied from its present location (in an area we will call *from space*) to a newly allocated segment of memory (namely, *to space*), replacing the old object with a *forwarding pointer*. After the copy has been made, the collector adds a pointer to it to a queue, known as the *scavenging queue*. This entire process is known as *evacuation*.

After all roots have been evacuated, collection proceeds by popping an object from the scavenging queue, evacuating its references if they haven't yet been evacuated, and updating its pointers. This continues until the scavenging queue is empty, at which point all live objects have been relocated to *to space* and no references to *from space* remain.

Moving collectors offer excellent data locality characteristics and supports extremely cheap allocation. The collector is typically configured with a small, cache-sized nursery (~1 MByte), ensuring that nursery collections are of bounded size.[1] Furthermore, the fact that the nursery fits in cache permits extremely fast allocation and collection. The older generations, however, may grow in size without limit. As the collector must traverse the entire heap in order to collect, pauses due to oldest-generation collections may be arbitrarily long. This can be problematic for services which must guarantee bounded response time to external events.

1.1 Incremental, moving collection

Incremental collection is one means of mitigating long pause times. By constructing the collector to make progress while traversing only a fraction of the heap in each pause, one can

[1]Note that in a generational collector the fact that a nursery is of bounded size doesn't necessarily guarantee that it needs to do a bounded amount of work: the collector must also traverse the remembered set which may in principle contain the entirety of the old generation. However, in practice the weak generational hypothesis tends to hold quite well and consequently it is usually safe to assume nursery collections will require only $O(\text{nursery size})$ work.

bound pause times. However, due to book-keeping and synchronization overhead, incremental collection typically comes at the expense of throughput.

One common incremental, moving collection scheme is *Baker's algorithm*. Under this scheme, the collector evacuates and scavenges as described above, but may stop collection at any time. To ensure progress, the user program (which we will hencefore refer to as the *mutator*) must ensure that it doesn't introduce any new references to objects still residing in *from space* into the heap. This is accomplished by way of a *read barrier*, where the mutator assesses the location of all pointers it encounters and scavenges those still residing in *from space*.

The sort of naive read barrier used by Baker's algorithm tends to incur a significant runtime cost. [Cheadle2000] describes an incremental collection scheme implement in GHC based on a variant of Baker's algorithm. Their collector exploits a characteristic of GHC's heap structure to efficiently implement the read barrier. While the approach demonstrates promising pause-time reductions, it suffers from a number of issues. Specifically:

a. a dramatic increase in code size with attendant adverse effects on cache effectiveness
b. incompatibility with the *pointer tagging* optimization [Marlow2007] introduced into GHC in the intervening years, which significantly improves runtime efficiency on modern hardware.[2]
c. reduced efficiency in parallel programs due to the need for synchronization during scavenging[3]

These issues, particularly (b), pose a significant challenge to the use of the Cheadle concept in GHC today. Furthermore, without a trick analogous to Cheadle's read barrier, Baker's algorithm has been shown to exhibit poor performance compared to other modern collectors.

Another family of approaches is that of concurrent copying collectors. These generally fall into a few categories:

those requiring virtual memory subsystem support One approach to avoid the need to compact the entire heap in one uninterrupted pass is to use the machine's virtual memory subsystem to trap writes to pages containing unscavenged objects. While these approaches [Kermany2006] [Tene2011] offer outstanding pause times and very good performance, the implementation complexity and platform dependence that they demand is above what GHC can support.

those requiring additional indirection Another approach for avoiding $O(n)$ pauses during copying collection is to simply avoid including pointers in heap objects. Instead, one can use a handles and an auxiliary lookup data structure, ensuring that there is at most one reference to update for each evacuated object.

those requiring tracking of in-references Another means of bounding pause times for copying collection is to collect a data structure to ensure that the references to a relocated object can be found in $O(1)$ time [Detlefs2004]. However, the complexity associated with tracking this metadata introduces its own set of space and time overheads.

[2] [Marlow2007] reports that pointer tagging improved runtime performance by an average of 13%. However, it is likely that its effect is even more pronounced on today's hardware due to the importance of cache utilization and branch prediction on long-pipeline processors.

[3] Specifically, mutators must use atomic memory operations while scavenging to prevent potential memory leaks due to objects being copied multiple times.

1.2 Non-moving collection

In general, incremental collection is significantly easier in a *non-moving* collector. Under this design, each object remains in a single location throughout its lifetime. This frees the collector from the need to update references to moved data, which significantly simplifies incremental collection.

One such non-moving collector is the traditional *mark-and-sweep*. Under this scheme, the collector maintains a liveness flag for each allocated object. At the beginning of collection all such flags are cleared and the *marking phase* begins. In this phase the collector traverses all live objects in the heap starting from a set of roots, setting the liveness flag of every traversed object and adding the object's pointers to a marking queue. This continues until the marking queue is empty, at which point each object's mark bit reflects its liveness state. Upon conclusion of marking the *sweep phase* begins, where the collector walks the entire heap looking for allocation cells containing dead objects; these are returned to a free list from which future allocations can be serviced.

A significant advantage of this approach is that marking can be carried out either incrementally or concurrently with mutator execution. However, this approach makes a few trade-offs relative to a moving collector:

locality It is often true that heap objects which are allocated at the same time will also tend to be accessed together. Copying collectors are helpful in such cases as temporally proximate allocations will generally end up near one another in address space. This help cache utilization, which can improve runtime performance significantly on modern hardware.

By contrast, non-moving allocators will tend to scatter temporally proximate allocations throughout address space.

fragmentation Heap fragmentation is not a concern in copying collectors as they do not preserve interstitial free space.[4] This is not true of non-moving collection, which must track and allocate into the gaps between live objects. However, there are well-known techniques to mitigate the fragmentation problem and many have argued that fragmentation tends not to pose a problem for modern free-list allocators in practice [Johnstone1998].

allocation overhead Moving collectors permit the use of extremely efficient bump-pointer allocation. By contrast, non-moving collectors must make use of some sort of free-list allocator, even the most efficient of which will be significantly more expensive than a bump-pointer allocator.

Despite these trade-offs, the relative ease of concurrent collection makes non-moving collection a natural choice in settings where low pause time is prioritized above throughput.

2 Detailed design

We propose a concurrent mark-and-sweep collector, inspired by [Ueno2016], for collection of the old generation. By retaining a copying collector for the nursery, we avoid imposing

[4]While copying collectors eliminate heap fragmentation, GHC's particular collector builds on top of a block allocator. In principle this allocator may fragment, although this tends not to be a problem in practice.

the high allocation-cost of a free-list allocator on the mutator while still enabling low-pause collection of the old generation.

This design enables concurrent collection by way of snapshotting: the heap representation is constructed in such a way that with a small amount of additional metadata and a means of keeping track of pointer mutation (via a *write barrier*), the collector can compute a conservative approximation of the heap reachability graph as it existed when the snapshot was taken. This ability allows the collector to safely collect while the same heap is mutated by one or more mutator threads.

Under this design, the mutator will allocate via the same fast bump-pointer allocation scheme currently used in GHC. When the nursery fills, the generation 0 collector will move promoted data into the generation 1 heap, managed by a free-list allocator. The non-moving generation 1 heap will be periodically collected via a mark-and-sweep collector.

In the following we elaborate on the heap data structure, allocation, and collection algorithms.

2.1 Non-moving heap structure

In order to avoid heap fragmentation, the allocator consists of a family of sub-allocators $\{H_3, ..., H_n\}$. Each allocator H_m services allocations of size 2^m bytes.

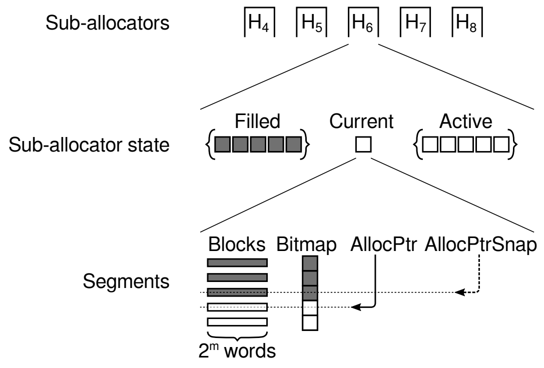

Figure 1: The three levels of hierarchy of the non-moving heap structure consisting of: multiple sub-allocators servicing a logarithmically-spaced selection of allocation sizes; the sub-allocator state tracking containing a set of fill segments, an active segment, and a set of active segments; the segment structure, consisting of a *Blks* array, a liveness bitmap, and two allocation pointers.

The memory allocated into by H_m is divided into *segments* (following the nomenclature of Ueno et al.) each of size N. Each segment consists of

- an array of N/m *blocks* (BLKS) each of size m into which objects are allocated
- a *marking bitmap* (BITMAP) of length N/m bits[5]
- an *allocation pointer* (ALLOCPTR) pointing to the first unallocated block within the segment

[5]If parallel marking is desired then there are two options for representing BITMAP:
a. use an entire byte to represent each bit, allowing safe concurrent update without synchronization, or
b. continue to use a bitmap representation but use an atomic OR operation for marking

- an *allocation snapshot pointer* (AllocPtrSnap), which captures the value of AllocPtr at the time the last snapshot was taken.

The suballocator's state is a tuple of

- A Filled list, containing segments which have no unallocated blocks
- An Active list of segments which contain unallocated blocks
- A Current segment, into which the allocator is currently allocating

In addition to the state of the sub-allocators, the allocator will also maintain a list Free of unallocated segments.

2.2 Allocation

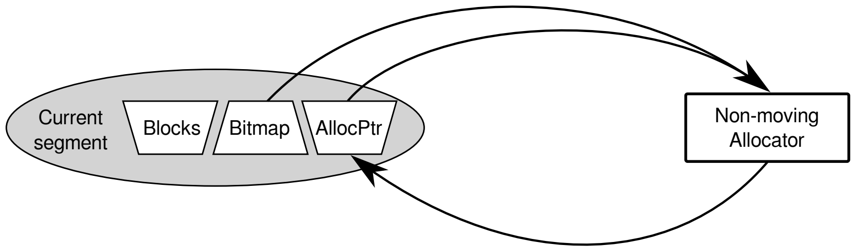

Figure 2: The non-moving allocator reads and updates AllocPtr and reads from the Bitmap of the segment into which it allocates.

Allocation for a request of size n proceeds as follows,

1. Select sub-allocator $H_{\lceil \log_2 n \rceil}$ to service the request
2. Find a segment to service the request
    a. Examine H_m's current segment; if it is full (e.g., its allocation pointer is beyond the last block of the segment) then place the segment in the Filled list and attempt to obtain new segment from Active and if successful proceed to (3), otherwise proceed to (b)
    b. If Active is empty then attempt to obtain a fresh segment from Free and if successful proceed to (3), otherwise proceed to (c)
    c. If Free is empty then either allocate address space for additional segments or initiate garbage collection and retry (2).
3. Set AllocPtr to the address of the next available block (i.e., the first block in Blks after AllocPtr whose corresponding bit in Bitmap is not set). Return the old value of AllocPtr as the allocated memory.

This is ultimately a trade-off between cache effectiveness and synchronization overhead. The performance implications of this decision will need to be characterized if parallel marking is implemented. [Garner2007] offers a slightly dated characterisation of this design decision, showing that atomic operations incur a roughly 10-15% cost in a trace loop on modern architectures.

An interesting implication of (b) is that it means that we may be able to do away with the distinct Bitmap structure entirely, instead tracking the mark state using the low bit of the info table. This would require some

Note that allocation does not set the allocated block's bit in BITMAP; this will only happen during the mark phase. This separation of concerns ensures that only the collector writes into BITMAP, enabling allocation to proceed concurrently with marking.

2.3 Snapshotting

To allow concurrent marking and mutator execution, we use the snapshotting ability of the non-moving heap to capture a coherent picture of the heap. We can then safely collect by implementing our mark phase to compute a conservative approximation of the reachability relation as it was at the time the snapshot was collected. Specifically, this approximation must respect the following invariant, which we will call the *snapshot liveness invariant*:

> All objects that were reachable at the time the snapshot was collected must have their mark bits set at the end of the mark phase.

Ensuring this invariant[6] requires some degree of cooperation from the mutator, which may change pointer references during the mark phase. Specifically, mutators must implement a *write barrier*, ensuring that all pointers that a mutator overwrites are added to a list known as the *update remembered set*, UPDREMSET.[7] The mark phase can then mark the transitive closure of objects added to this set, ensuring that the invariant is respected.

Since we don't want to require stack mutations to be subject to the barrier we must mark stacks eagerly; that is, all pointers living on thread stacks must be added to the mark queue at the time the snapshot is taken. Collecting a snapshot and enabling the write barrier requires synchronization between the collector and all mutator threads. Specifically, the following protocol is used:

1. Stop all mutator threads
2. For all segments in all sub-allocators' FILLED and CURRENT segments[8] assign ALLOCPTRSNAP := ALLORPTR
3. Record root sets (e.g. threads on the scheduler queues and their stacks, weak pointers, and static pointer table entries, stable pointers).
4. Enable the global mutator write barrier flag
5. Resume mutator threads

2.4 Collection

The initial implementation will be a non-parallel, concurrent collector, although extending to allow parallel collection (e.g., multiple collector threads) will not require significant design changes. To enable concurrency, we take advantage of the *snapshotting* capability of the Ueno heap described above. This allows for sound collection in the face of concurrent mutation. However, our approach foregoes much of the complexity of the original Ueno paper which allowed concurrency during the root collection phase to reduce the pause due to the initial stop-the-world period.

[6]For instance, consider that the mutator overwrites the last reference to an object A prior to A being marked. According to the snapshot liveness invariant A must be marked but would not be in this case.

[7]The name "update remembered set" is intended to distinguish it from the generational remembered set used by GHC's current collector to track references from old to young generations.

[8]Note that it is unnecessary to update ALLOCPTRSNAP of segments in the ACTIVE list: segments in ACTIVE by definition have not been allocated into since the last collection and consequently ALLOCPTR will not have changed since we last updated ALLOCPTRSNAP at the end of that collection.

Garbage collection occurs in three phases: the *preparation phase*, followed by the *mark phase* and then the *sweep phase*. The preparation phase is a short stop-the-world period at the beginning of the collection cycle. The mark and sweep phases can both occur concurrently with mutator execution, but not with one another.

2.4.1 Preparation

An old-generation collection cycle begins with a single capability (which we will call the *initiator*) requesting a collection. In response, all other capabilities will suspend mutation. After suspending the mutators we will perform a garbage collection of all young generations. If while tracing the young heap we encounter a reference to an object living in the nonmoving heap we push the object to a list for later marking[9].[10] Additionally, any objects promoted to the nonmoving heap must also be pushed to this list.

Once the initial collection has finished, the initiator will build a list C of segments to be collected from the union of all sub-allocators' FILLED lists. It will then clear the BITMAP of all segments in C.

The collector will then take a snapshot as discussed in Section 2.3. Finally, the initiator will signal to other capabilities that they may resume execution and will itself proceed to the mark phase.

To summarize, in the preparation phase the collector has:

- performed an initial collection, promoting all live data into the non-moving heap.
- determined the set of segments C which will be collected in the coming collection cycle
- initialized the marking bitmaps of the segments to be collected
- taken a consistent snapshot of the heap segments
- collected a consistent root set from living threads
- enabled a write barrier to ensure consistency with the snapshot

2.4.2 Marking

During the mark phase the collector will trace objects from the following roots,

- roots from thread stacks collected in the preparation phase[11]
- the remembered set, UPDREMSET, mentioned above
- stable pointers

[9]One possible implementation strategy for this list is to simply re-use UPDREMSET. This works out nicely as it avoids introducing a new datastructure and easily accomodates parallel collection, since each GC thread holds a capability and each capability has its own update remembered set.

[10]An earlier version of this design instead mandated that all live data be promoted to the nonmoving heap during the preparatory GC. This sacraficed the ability to age objects before a major collection in exchange for avoiding implementation complexity.

While less complex, this design had the potential to copy significant amounts of data unnecessarily to the nonmoving heap. While it was originally thought that this unnecessary copying would occur infrequently enough to be of little consequence, some programs behaved quite pathologically (e.g. the `cryptarithm1` program from nofib, which was found to copy nearly an order of magnitude more under the nonmoving collector than the copying collector).

[11]A compelling, lower-pause-time, alternative to marking stacks during the stop-the-world is to instead do so before the thread is scheduled for the first time after a snapshot. This would shift pause time out of the GC-initiation pause into mutation-time, potentially dramatically reducing overall pause times in programs with many threads. The usual mark flag can be used to record whether the stack has been marked this collection.

- static pointer table (for the `-XStaticPointers` mechanism) entries
- (in the non-threaded runtime only) signal handlers

In addition to object's usual pointers fields, for proper tracking of CAF liveness, the collector must also trace pointers found in the static reference tables (SRTs) of known-live closures.

One important trait of the current moving collector is its ability to "short-out" indirections due to thunk update. Furnishing this ability in a mark/sweep setting requires that we save with each pointer in the trace queue a reference to the object in which the pointer was found. Since interior pointers are problematic for scavenging,[12] we represent this reference as a pair of a pointer to the containing object and a field index.

Marking proceeds in depth-first fashion as follows,

1. pop an item o from the trace stack or one of the above root sources[13]
2. identify where o resides:
    - if o resides in the nursery then ignore it.[14]
    - if o resides in the non-moving heap then identify which segment it lives in and proceed to (3)
3. if either[15]

[12]The need for scavenging of the trace queue is described in Section 2.4.5. The difficulty with interior pointers hinges on the fact that, in order to scavenge, one must be able to find the object header to determine whether the object has been replaced with an indirection. While this is possible for interior pointers into objects in the non-moving heap (since objects are aligned to the block size), this is not possible for pointers into the nursery.

[13]A nice property of this mark algorithm is that its cache utilization can be easily improved by use of *edge enqueuing* [Garner2007] and a *prefetch FIFO* [Cher2004] [Boehm1993].

Under an *edge enqueuing* scheme we add all pointers of a marked object to the mark queue, regardless of whether or not the referents have been marked. This is in contrast with a traditional *node enqueuing* mark algorithm, which adds only those pointers whose referents are not themselves marked. This scheme trades a significantly higher number of enqueue operations in exchange for dramatically better temporal locality during the mark phase. [Garner2007] found that enqueue operations are a negligible contribution to the overall cost of marking, which is rather dominated by the cost of memory accesses.

To further improve locality we use a prefetch FIFO: Here we maintain a fixed-size FIFO data structure which tracks the entries that we will mark in the future. To fetch the next item to mark we look at the tail of the FIFO. As we do so we also push a new item from the trace stack or roots to the head of the FIFO and additionally issue a prefetch instruction on this new item. With proper FIFO size we can dramatically improve the data cache hit rate and consequently hide memory access latency. [Garner2007] shows that a FIFO length of eight is sufficient to hide memory latency on modern hardware.

Together, these two measures will serve to shield the mark phase from much of the suboptimal spatial locality produced by the free-list allocator.

[14]Note that because we have guaranteed that all objects reachable from younger generations were marked during preparation, we can safely ignore objects living in the young generations during mark. Such objects must have been allocated since the beginning of the mark phase and therefore aren't the subject of the current collection cycle.

[15]This criterion for marking identifies the set of objects which was alive at the time that the snapshot was taken (which we are obligated to mark by the snapshot invariant). However, seeing this is a bit subtle. To make this clear consider the following three cases:

- The block's address is less than AllocPtrSnap: In this case the block must have already been allocated at the time the snapshot was taken therefore we must mark it under the snapshot invariant.
- The block's address is greater than or equal to AllocPtrSnap and
    - The block is marked: In this case the block's segment was either in the active or current list when the snapshot was taken but the block was live last time the segment was swept and therefore contains a value object. Consequently, we must mark it.

- the address of *o* is >= AllocPtrSnap and the object is marked, or
- the address of *o* is < AllocPtrSnap

then mark *o* and push its pointers to objects in the non-moving heap, each paired with a reference to *o* and its field index, to the trace stack. In the case of CAFfy things we also must push the object's SRT.

Marking finishes when both the trace stack and all root sources are exhausted. At this point the collector can do the following:

- filter the nonmoving generation's generational remembered set, removing any unmarked objects (since they will be swept shortly)
- disable the snapshot write barrier and signal to the mutators to continue
- proceed to the sweep phase

2.4.3 Marking and cyclic heaps

Notice how the mark criterion requires that we mark closures that have already been marked. This is problematic in the presence of cyclic structures on the heap as the mark will no longer terminate (strangely, this is an issue which the Ueno paper does not mention).

The problem is that we cannot distinguish between the following situations:

1. An object living in an active or current segment which is marked because it was live the last time we swept it
2. An object which is marked because we have already marked and traced it in the current major GC

We saw above that we must mark in situation (1) since failing to do so may lead to incorrectly swept objects. However, we must not mark in situation (2) since doing so may lead to non-termination. Consequently, we must encode more information in the segment metadata to distinguish these cases.

To account for this we extend the segment bitmap to include another bit per block. This will allow us to represent the following three states:

a. block is unmarked
b. block was marked in a previous collection but hasn't been marked/traced in the current GC
c. block has been marked/traced in the current GC

To avoid needing to update the bitmaps on every major GC to flip (b) and (c), we use the same trick used by the moving GC to mark static objects. We start by defining two states which we call mark epochs A and B. We then give the assign the following meanings to the mark bit states:

`0x0` unmarked

– The object is not marked: In this case the block was not allocated at the time the snapshot was taken (even if it may be filled now, due to concurrent mutation) and consequently we are under no obligation to mark or trace it. Moreover, it is impossible to determine from AllocPtrSnap alone whether the block has been allocated and consequently it would be unsafe to do so (since it may not even contain a valid object).

`0x1` marked in epoch A

`0x3` marked in epoch B

We then track the epoch of the current ongoing mark as global state, CurMarkEpoch, which gets flipped during mark preparaion. If and only if the epoch indicated by a block's mark bits matches CurMarkEpoch then we can conclude that it was marked in the current collection.

2.4.4 Sweeping

In the sweep phase the collector examines the bitmap of each segment, which now conservatively[16] reflects block liveness, and

- if all blocks of the segment are dead, then the segment is moved to Free
- else if all blocks of the segment are live, then the segment is moved to Filled
- otherwise the segment is moved to Active

Before returning a segment to Active, the collector initializes the segment's AllocPtr and AllocPtrSnap to point to its first available block.

In addition to sweeping blocks, the sweep phase must traverse the generational remembered set and remove closures which were found to be dead (lest a younger generation collector try to traverse a closure that was already swept). This requires that we have a predicate for determining whether a closure is alive (which is surprisingly subtle, as discussed in Section 2.5.2).

2.4.5 Interaction with the nursery collector and mutator

Under this proposal, the mutator only allocates into its nursery, meaning that no changes will need to be made for code generation of mutator allocations.

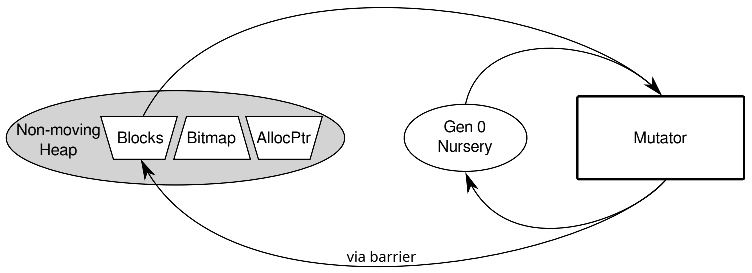

Figure 3: The mutator may allocate into the nursery and may read and mutate the non-moving heap. However, a write barrier must be respected during mutation.

Only the nursery garbage collector allocates using the non-moving allocator. The nursery collector will use the allocation algorithm described in Section 2.2 to allocate space for values evacuated from the nursery.

[16]Specifically, the liveness bitmaps will designate as dead a subset of the blocks that were dead at the time that collection was initiated.

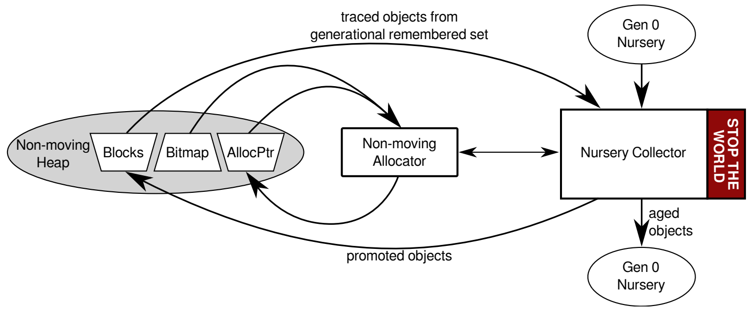

Figure 4: The nursery collector may promote some values to the old generation using the non-moving allocator while others may remain in the nursery due to aging. The nursery collector stops all mutator threads as well as the non-moving mark thread.

Note that nursery collections may run concurrently with the sweep phase of the non-moving collector. The safety of this concurrent allocation is due to mutation invariants respected by both the allocator and non-moving collector which can be phrased in terms of ownership (where a value may only be observed or mutated by its owner):

- the bitmap bits corresponding to all blocks before AllocPtrSnap belong to the collector
- the bitmap bits of all blocks after AllocPtrSnap belong to the allocator
- the mutator has ownership of allocated Blks[17]
- AllocPtr of segments in Filled belong to the collector
- AllocPtr of segments in Active belong to the allocator
- AllocPtr of a Current segment belongs to the allocator
- AllocPtrSnap belongs to the collector

The mutator and the non-moving collector interact in a few ways:

- the mutator can modify mutable objects (e.g. arrays) residing in the non-moving heap; this requires that such objects be added to the generational remembered set (as is already done today) and UpdRemSet
- the mutator can update thunks residing in the non-moving heap. These will be added to UpdRemSet when the snapshot write barrier is enabled
- the collector can enable and disable the mutator's snapshot write barrier

2.4.6 Scavenging after promotion

The nursery collector must scavenge any objects that it evacuates. Typically this happens by adding the block descriptor of the target block to the `todo_bd` list of the copying GC's `gen_workspace` and setting the block's Cheney-style Scan and Limit pointers. While a similar mechanism will work in the case of evacuation to the non-moving heap, we need to take care to ensure that the scan pointer (`bdescr.u.scan`) points to the data portion of the

[17]The claim that the mutator has ownership of Blks doesn't strictly hold in the presence of the selector optimisation and indirection short-cutting, both of which the proposed design implements. However, in this case the mutations made by the collectors are semantics-preserving and made in such a manner that the mutator cannot observe any inconsistent state. More about this below.

non-moving segment, not the segment header/bitmap. This will be done when NextFree is set after sweep.

While we could in principle allow the nursery GC to scavenge all objects past the Scan pointer until the end of the segment, this would mean that many objects would be scanned unnecessarily (e.g. any objects pre-existing in the segment).

To avoid unnecessarily scavenging objects which contain no references to moved objects, we propose that `scavenge_block` learn to look at the non-moving bitmap. Specifically, it is only necessary to scavenge objects between the Scan and Limit whose bitmap bits are not set as these are precisely the objects that were evacuated in the current copying collection:

- If an object's bitmap bit is set, then we know that it was alive at the time of the last major garbage collection.
- If an object bit is unset, then we know that it was evacuated into the non-moving heap during the present nursery collection and should be scavenged.

As we scavenge an object, we set its bitmap bit since we have established that it is live. Note that it is quite important that we do this, lest we potentially later scavenge dead objects, resulting in memory unsafety. To see why this is necessary, consider the following situation (see ticket #25):

1. During a minor GC we evacuate two objects to the nonmoving heap:
   a. We have object O_1 in a Filled segment
   b. We have object O_2 in a Current segment which holds a pointer to O_1
2. We scavenge both objects
3. Both objects become unreachable
4. Soon thereafter we run mark & sweep; O_1 will be swept. However, since O_2 is in a current segment O_2 is not swept (as we only sweep Filled segments).
5. Later we evacuate more objects into O_2's segment; since AllocPtr has the same value it had in step (1), O_2 will be in the range of scavenged objects. Furthermore, its mark bit will be unset, meaning that we will scavenge it. In so doing we will attempt to evacuate O_1, which died long ago.
6. Chaos ensues.

By marking O_2 when we scavenge it in step (2) we ensure that it doesn't get scavenged in step (4), after we have already swept O_1.

To track which segments need scavenging a work-list needs to be maintained; at the beginning of a minor GC all Current segments will be added to this list. At the same time, the segment's Scan pointer will be initialized to the value of AllocPtr.[18] Further, whenever a segment is filled, the new Current segment will similarly need to be added to the work-list and have its scan pointer initialized.

Finally, in addition to scavenging segments containing newly-promoted objects, we also must scavenge the oldest generation's generational remembered set (`mut_list`). To see why, imagine that we have a mutable object P living in the non-moving heap. Further, imagine that P has a pointer field. Naturally, P was scavenged when it was originally promoted to the non-moving heap. Now imagine that a mutator modifies P's pointer field to point

[18]If we can guarantee that allocation will *only* occur as the result evacuation then we can omit the update of Scan. This is the case since we know that Scan was updated when the block became active and no allocation has occurred outside of evacuation.

to a new object, O, living in a younger generation. Now imagine that we initiate a major collection; naturally, before we can start such a collection we will perform a GC, promoting all of the younger generations' contents to the non-moving heap. After doing so we must scavenge P, lest it retain a reference to O's (now dead) location in the moving heap. This is in contrast to the moving collector, which will needs no special treatment for `mut_list`; if P is found to be live then it will be evacuated and later scavenged in the course of normal copying collection.

Moreover, in scavenging the generational remembered set we ensure that objects marked as dirty (e.g. `MUT_VAR`s) are returned to their clean state.

2.5 Implementation and interactions

The above non-moving allocator and collector will be implemented in GHC's existing runtime system. While it will not be necessary to modify GHC's closure structure, a significant amount of refactoring will need to be carried out to accommodate multiple collectors.

Below we will discuss a variety of interactions with GHC's existing runtime and features that must be accounted for in the design and implementation.

The non-moving collector will be configurable with one or more copying generations and a single terminal non-moving generation. However, for simplicity the discussion below will assume a single moving nursery followed by a non-moving old generation.

2.5.1 Interaction with GHC's block allocator

The non-moving heap segments will be allocated as blocks from GHC's existing block allocator. The block allocator currently uses a default block size of 4 kilobytes, eight times smaller than the 32 kilobyte segment size used by Ueno et al. While it is possible to treat the block size and the segment size as independent quantities, the implementation becomes significantly simpler if they are tied. Moreover, with large page support[19] becoming increasingly common, it is possible that GHC's current block size is suboptimal given today's hardware and software. Consequently, we will investigate the effect of increasing the block size and potentially raise the default value.

Non-moving blocks will be distinguished from the blocks belonging to the copying collector through a `BF_NONMOVING` flag added to the existing `flags` field of the `bdescr` type. However, even without knowing of this flag the moving nursery collector will know to not trace pointers within the non-moving heap (other than those in the generational remembered set) by the block's generation field.

2.5.2 Liveness predicate

There are two places during sweep where the non-moving collector needs to know whether a given block contains a live closure:

- weak pointer marking (Section 2.5.17)
- generational remembered set sweeping (Section 2.4.4)

[19] See, e.g., https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/vm/transhuge.txt

For this we use a predicate IsAlive(blk). Defining this predicate properly is a bit subtle. Note that this predicate may only be used after a full mark has been finished but before sweep so we know that the mark bitmap and snapshot pointer reflect the snapshot of the on-going collection.

The IsAlive predicate must consider four cases:

- the block is below AllocPtrSnap: In this case the closure was reachable when the snapshot was taken and consequently we must conclude that the block is alive.
- the block is above AllocPtrSnap:
  - BitMap(blk) $= 0$: In this case the block was not allocated when the snapshot was taken. however, it may have been allocated since then and therefore we must conclude that the block is alive.
  - BitMap(blk) $=$ CurMarkEpoch: In this case the object was alive in the last snapshot. Consequently we must conclude that the object is still alive.
  - BitMap(blk) $\neq$ CurMarkEpoch: In this case the object was not reachable in the last snapshot. We can therefore conclude that the object is dead since the snapshot invariant guarantees that all objects reachable in the snapshot would be marked.

2.5.3 Pointer `Array#`s

GHC's `Array#` and `SmallArray#` types implement contiguously-allocated arrays. Since the non-moving collector is concurrent, collecting these poses no particular challenge. To avoid complex interactions with the nursery collector, the non-moving card table of `Array#` won't be usable by the non-moving collector. However, the non-moving collector has little need for the card table as updates are already accounted for by way of UpdRemSet. Marking an `Array#` will involve pushing all pointers contained within to the mark queue and setting a new `BF_NONMOVING_MARKED` flag in its block flags. The marker will know it can ignore `BF_NONMOVING_MARKED` blocks.

To avoid duplicating the entire array on the mark queue, we might, as an optimization, introduce a new form of mark queue entry (which we'll call `MARK_ARRAY`) containing a pointer to the array and the index of the first element that has not yet been traced. When an `Array#` is first encountered it will be pushed as a `MARK_ARRAY` with `idx=0`. When the marker encounters such an entry it will break off a fixed-size chunk of pointers from the array, and push them to the mark queue along with a new `MARK_ARRAY` entry covering the rest of the array. This allows us to bound the size of the mark queue while preserving locality during tracing.

2.5.4 Pinned `ByteArray#`s

GHC exposes a mechanism for users to mark `ByteArray#`s as immovable ("pinned"). This mechanism is the basis of the frequently-used `bytestring` library and is critical for interoperability with foreign code. This requires no special support in the non-moving collector as allocations are pinned by default.

However, since the moving collector will still be in use for nursery allocations, we must leave the runtime's existing allocation-pinning logic in place.

2.5.5 Large objects

Like many collectors, GHC's moving collector currently treats heap objects larger than a fixed threshold (`LARGE_OBJECT_THRESHOLD`, currently set to 80% of the block size or around 3 kilobytes) as immovable. Such objects are allocated into their own blocks (allocated using the block allocator) which are distinguished with the `BF_LARGE` block flag.

This special treatment will become particularly important in the non-moving collector, which has a finite number of sub-allocators with which to service allocation requests. With a dedicated allocation path for large objects, the non-moving allocator can consist of as few as nine sub-allocators, $\{H_3, ..., H_{12}\}$.

Beyond basic tracing logic in the mark phase, no particular support will be necessary in the non-moving collector to support this dedicated large-object path other than the addition of a new block flag, `BF_NONMOVING_MARKED`.

In practice, there are a few thorny implementation details that must be handled. Large objects are currently tracked in the `generation` datastructure in GHC's storage manager which is mutated by both major and minor GC. In particular, each generation has three lists of objects: `large_objects`, `todo_large_objects`, and `scavenged_large_objects`. Over the course of a moving collection live large objects are moved from `large_objects` to `todo_large_objects` as they are evacuated. The scavenger then pulls objects from `todo_large_objects`, scavenges them, and places them on `scavenged_large_objects`. At the end of a collection the collector traverses the collected generations and frees their `large_objects`. For all non-collected generations the blocks in `scavenged_large_objects` are moved to `large_objects`.

The fact that `oldest_gen` (the `generation` of the oldest GC generation) is global state poses a bit of a complication in the presence of concurrent marking. To avoid having to share the oldest generation's large object list with the younger generation collection we define two new sets of large objects, `nonmoving_large_objects` and `nonmoving_marked_large_objects`. After the moving phase of a major collection has finished we move the contents of `oldest_gen->large_objects` to `nonmoving_large_objects` as well as set the `BF_NONMOVING_SWEEPING` flag[20] of each block we move; this is the set of large objects that we will collected during the concurrent mark-and-sweep. The mark phase will move large objects that it encounters with `BF_NONMOVING_SWEEPING` set from `nonmoving_large_objects` to `nonmoving_marked_large_objects`. Objects not found on `nonmoving_large_objects` can be assumed to either be in a younger generation or already marked.

During sweep we free all objects in `nonmoving_large_objects` and move the contents of `nonmoving_marked_large_objects` back to `nonmoving_large_objects`.

[20]The `BF_NONMOVING_SWEEPING` flag allows us to identify which large objects are in the snapshot during marking. This is important since minor generation collections will add promoted objects to `oldest_gen->large_objects` during the course of a major collection. Such objects aren't part of the snapshot for the on-going major collection and can therefore be ignored by mark. Moreover, failing to ignore them would complicate implementation as we wouldn't know which block list, `oldest_gen->large_objects` or `nonmoving_large_objects`, to remove a large object block from during marking.

2.5.6 Constant applicative forms

As discussed in Section 2.4.2, GHC has special arrangements for tracing references to constant applicative forms (CAFs) and their referees from static code. Specifically, these dependencies are tracked via a static reference table (SRT) referred to by the static code's info table. SRT entries will need to be traced by the mark phase to ensure that CAFs liveness is correctly assessed.

Beyond tracing SRTs, the non-moving collector also needs to maintain a mark flag for each CAF. There are a few options here:

1. maintain a list of marked static objects in an auxiliary data structure (e.g. a hash set).
2. claim an unused bit in the static object closure for use by the non-moving collector as a mark bit.

Initially we propose to implement option (1), potentially moving to (2) if mark performance is found to suffer as a result.

2.5.6.1 Intrusive static marking

The use of an auxiliary data structure for marking static objects is both unduly expensive and quite inconsistent with how marking of other objects is performed, both in the moving and non-moving collectors.

Attaching auxiliary information to static objects is tricky since they live outside of the dynamic heap. For instance, the moving collector typically uses the containing block descriptor to determine whether a closure in the dynamic heap lives in from- or to-space. This is not possible in the case of static objects. Instead the moving collector encodes whether a static object has been evacuated in an additional field: the static link field. This field has two roles:

- it serves as an intrusive link field so that the list of CAFs can be collected and later walked in the `-debug` RTS.
- the two low-order bits encode the static flag, which records whether the closure has been evacuated in the current GC.

The static flag field has four states:

- `0`, which indicates a closure which has never been seen by the GC
- `STATIC_FLAG_A` and `STATIC_FLAG_B`, which indicate a closure that has been marked either in either the current or previous GC (depending upon the value of the global `static_flag` state, which is flipped before every major GC)
- `0b11`, which indicates that the GC should ignore this closure (e.g. because it is not CAFfy)

The static link field is set by the moving collector during evacuation.

Under the non-moving collector each static object will be in one of five states:

- the closure has never been seen by the GC
- the closure was evacuated to the non-moving in the current major GC cycle but has not yet been marked
- the closure was evacuated to the non-moving in a previous major GC cycle but has not yet been marked

- the closure has been marked
- the closure can be ignored by the GC (e.g. because it is not CAFfy)

We can view these states as a product of the existing four-state static flag and an independent binary non-moving mark flag. These states can be encoded in the three low-order bits of the static link pointer. Unfortunately, this does mean that we lose support for 32-bit platforms, which have only two unused low-order bits.

Which setting of the non-moving mark flag we consider to be "marked" will be tracked in by a global flag `nonmoving_static_flag`. This flag will be flipped during the preparation phase of the non-moving collection. To ensure that CAFs evaluated during a concurrent non-moving mark are properly marked during the next major GC `newCAF` must take care to set the non-moving mark flag of the CAF with the current value of `nonmoving_static_flag` when it initializes the new CAF.

2.5.7 Selector optimization

GHC's existing garbage collector implements an optimization known commonly as the *selector optimization* [Wadler1987]. This optimization avoids a set of common space leaks caused by thunks of applications of field selectors. For instance, consider a heap containing the following:

```
w = (_, v)
x = snd w
y = (x, _)
z = fst y
a = f z    -- N.B. a thunk of `f` with free variable `z`
```

When collecting this heap, the garbage collector would replace `a`'s pointer to `z` with the value of `z`'s reduction, `v`. This potentially allows `w`, `x` `y`, and `z` to be collected, which avoids some classes of space leaks. We call a sequence of selector applications that reduce to a WHNF result a *selector chain* (which we will abbreviate as "chain"). In the case above the chain induced by the reduction of `z` is `[z = fst (x,_), x = snd (_, v)]`.

Since this feature is a bit subtle, we will first review the existing implementation in the moving collector, followed by a discussion of the proposed implementation in the non-moving collector.

2.5.7.1 Implementation in moving collector

This optimisation is implemented during evacuation. Before trying to evacuate a selector closure it checks that a) it is not a static closure, and b) it hasn't yet been evacuated. If either of these conditions are not true then it gives up, falling back to standard evacuation.

To evacuate a selector closure the collector first locks the closure by replacing it with a whitehole. It then evacuates the closure and overwrite the first field of the closure that remains in *from space* for use as a link pointer to keep track of the selector chain currently being traversed. For this reason, the current implementation relies crucially on the stop-the-world nature of the moving collector.

After whiteholing the selector the collector then looks at the selectee; there a few cases here:

- *the selectee is a forwarding pointer:* In this case the selectee has already been evacuated. Consequently, we choose not to perform the selector optimisation since the space for the selectee has already been allocated. We unwind the changes made, replacing all elements of the chain we've traversed with indirections.
- *the selectee is an indirection:* In this case we simply follow the indirection and try again.
- *the selectee is another thunk selector:* In this case it adds the selector onto the chain, setting its link field to point to the previous closure in the chain.
- *the selectee is a constructor:* In this case we have possibly reached our result; we examine the value of the selected field and check whether it is also a selector thunk or indirection:
  - If it is an indirection then we follow it
  - If it is a selector then we recurse.
  - If it is neither an indirection nor a selector then we have found the result of the chain
- *the selectee is a thunk:* In this case the selector optimisation does not apply; we unwind the changes made, replacing all elements of the chain we've traversed with indirections.

The implementation enforces a recursion depth limit (namely `MAX_THUNK_SELECTOR_DEPTH`, currently 16) to bound stack usage. However, this limit only counts nested selector thunks, e.g. `fst (fst (... (fst x)))`. The collector will traverse interleaved selector-constructor pairs without limit, e.g.

```
a = (fst b, _)
b = (fst c, _)
c = (fst d, _)
d = (x, _)
```

When the collector arrives at the result of a chain it walks back over the chain it traversed (via the link pointer), replacing each selector with an indirection pointing to the final result and updating the referee of the selector to reflect the location of the result in *to space*.

If the collector instead arrives at a thunk or whitehole (indicative of another GC thread already evacuating the closure) then the selector optimisation aborts. It walks back over the chain, replacing all selectors with indirections.

2.5.7.2 Implementation in non-moving collector

Let's examine how the mark phase would handle the example considered earlier in this section:

```
w = (_, v)
x = snd w
y = (x, _)
z = fst y
a = f z    -- N.B. a thunk of ‘f‘ with a single free variable, ‘z‘
```

Where `a` is a root. Marking `a` will proceed as usual as it is a standard thunk. Note, however, that the selector optimization requires that we be able to update `a`'s field referring to `b` when we later discover that `z` is a selector thunk. To ensure that this can be done, mark queue entries must include a reference to the closure location and field where the reference to the object to be marked was found (known as the “origin” of the entry). Consequently, a marking `a` will result in the entry `{closure=z, origin=a[0]}` being pushed to the mark queue.

When the mark queue entry for `z` is popped, the mark phase will notice that its head, `fst`, is a selector and attempt to apply the selector optimisation. Before proceeding, the mark phase will white-hole the selector[21] to indicate that it is under evaluation and to avoid loops. As in the moving collector, we will chain selector closures together by using the selector closure’s link pointer.[22] To ensure that younger generation collections don’t interfere with this process it is important that the moving collector’s preconditions to the selector optimisation prevent it from following references into the non-moving heap.

Once the non-moving collector has white-holed a selector, it will then walk down the selectee chain in much the same way that the moving collector, with the same depth limit on nested selectors.

If a WHNF result is reached then the collector will walk back over the chain, replacing each selector with an indirection. Finally, the collector will also replace the value of the origin field recorded in the queue entry with the final result. Note that the origin field update must be done with a compare-and-swap to ensure that the collector does not overwrite a modification made to the object by the mutator since the mark queue entry was pushed (e.g. in the case of a mutable `Array#`).

Note that it is still necessary to mark the entire chain after having applied the selector optimisation due to the snapshot invariant. This means that we will not be able to reclaim the chain until the next collection cycle.

2.5.8 Indirection short-cutting

The moving collector is able to short-out references through indirections encountered during evacuation. This is important for program performance as thunk updates introduce indirections during the course of normal evaluation.

This feature can be viewed as a limited form of the selector optimisation described above. Implementation in the non-moving collector follows the selector optimisation closely: when an indirection is marked the mark phase checks that the indirectee is not also an indirection. If it is then it follows it and loops. When something other than an indirection is reached, it overwrites the origin field with the result via compare-and-swap.

[21] Due to its concurrent nature, the non-moving collector needs to be very careful in how it changes the heap to avoid allowing the mutator to see an inconsistent state. In particular, we must never change the type of a closure which a mutator may find a tagged reference to. Thankfully, references to selector are never tagged as constructors; afterall, selectors are thunks. This means that white-holing a selector is safe, assuming we take care to CAS the info table to avoid racing with evaluation by a mutator.

[22] Thunk closures have what is called an “SMP header” which contains the word that is used for the chain link pointer. This allows us to chain together selectors without overwriting their selectee fields, which is quite important since we may need to revert the changes made by the GC if the selector optimisation ultimately comes to a thunk and must abort.

Note that the presence of the SMP header isn’t quite as important in the moving collector since it first evacuates the selector before modifying it. This means that it can modify the closure in *from space* however it likes since it will ultimately be overwritten by an indirection anyways.

As in the case of the selector optimisation, the indirection must be marked as it was alive when the snapshot was taken.

2.5.9 Write barrier

As discussed in Section 2, a write barrier is necessary to ensure that collection respects the snapshot liveness invariant. This barrier affects writes to pointer fields. For instance, if `x` is a pointer to a pointer field, then the assignment `*x = y` requires that the initial value of the pointer `*x` is added to UPDREMSET.

The write barrier will be controlled by a global flag defined by `libHSrts` and will require the following changes to the runtime system and generated code:

Thread suspension When lazy blackholing is enabled the thread suspension logic is responsible for traversing the to-be-suspended thread's stack and blackhole all thunks under evaluation (identified by update frames on the stack). This codepath will need to be taught to add the free variables of such under-evaluation thunks to the update remembered set.

source reference: `rts/ThreadPaused.c:threadPaused`

Eager blackholing When eager blackholing is enabled (with the `-feager-blackholing` compiler flag) the code generator produces code to replace entered thunks with `EAGER_BLACKHOLE` objects.[23] This code needs to be modified to add the free variables of such thunks to the update remembered set.

source reference: `compiler/codeGen/StgCmmBind.hs:thunkCode`

`UPD_FRAME` entry code In this case we will be replacing a thunk closure or blackhole with an indirection. When the write barrier is enabled the entry code will add the pointer fields of the thunk to UPDREMSET. Furthermore, the code will add the thunk's SRT to ensure that the static references of its code are marked.

source reference: `rts/Updates.h:updateWithIndirection`

`MutVar#` These objects belong to a class of closure types which we call "dirty/clean" objects. That is, each `MutVar#` has a dirtiness flag, encoded by its info table. Specifically, a `MutVar#` may have one of two info tables:

- `MUT_VAR_CLEAN_info`: Denotes a `MutVar#` which is known to contain no a reference to an object living in a younger generation. Such objects are not in the generational remembered set.
- `MUT_VAR_DIRTY_info`: Denotes a `MutVar#` which has been mutated and therefore may refer to an object living in a younger generation. Such objects are added to the generational remembered set when they are dirtied (by the `dirty_MUT_VAR` function).

This dirtiness flag allows the runtime system to avoid adding a given object to the generational remembered set more than once.

[23] `EAGER_BLACKHOLE` objects differ from normal `BLACKHOLE` objects in that they do not guarantee single-entry. The thread suspension code later uses stronger synchronization to break ties between multiple threads who entered the same thunk, replacing the `EAGER_BLACKHOLE` with a normal `BLACKHOLE`.

The non-moving collector's update remembered set has slightly different needs: it must ensure that the pointers reachable by the `MutVar#` at the time of the snapshot are marked. To do this we must record in UpdRemSet the *old* pointer value before the first mutation of each `MutVar#` after the snapshot is taken.

To do this we rely on the fact that all live objects are promoted to the old generation before a major collection. This means that all `MutVar#`s can be marked as clean during the preparatory phase of a major collection (specifically this will happen when scavenging the `mut_list` during the preparatory minor collection).

The fact that all `MutVar#`s are marked as clean on initiation of a major collection allows us to use the same dirtying logic used by the generational remembered set in the update remembered set write barrier. Specifically, `dirty_MUT_VAR` will record the old value of a `MutVar#` to the update remembered set if and only if the `MutVar#` is clean.

source references:

- `rts/sm/Storage.c:dirty_MUT_VAR`
- `rts/sm/StgCmmPrim.c:emitPrimOp` (the `WriteMutVarOp` case)

array mutation GHC's current moving collector always considers mutable arrays to be in its generational remembered set.[24] This means that they will be scavenged by collections of all generations younger than that in which the array resides.

Therefore, `MutableArray#` and `SmallMutableArray#` mutations occurring during the concurrent mark must implement a write barrier adding overwritten pointers to UpdRemSet. Such mutations arise from a number of GHC primops including `writeArray#`, `copyArray#`, and `casArray#`. Most of these are emitted by the code generator.

source reference: `compiler/codeGen/StgCmmPrim.hs:doWritePtrArrayOp`

`TVar#` and `MVar#` Like arrays, these objects already have write barriers which add mutated objects to the remembered set, tracking dirtiness via the info pointer. Unlike arrays, these object types hold only a small number of pointers. This allows us to implement the write barrier significantly more efficiently.

The non-moving collector exploits a modified version of the clean/dirty behavior described above to ensure that only the first mutation of an object results in a push to the update remembered set.

source reference: `rts/sm/Storage.c:dirty_TVAR`

`WeakPtr#s` The tombstoning of a weak pointer is a form of mutation. However, as weak pointers are not traced, it does not change the reachability graph and therefore it needs no special treatment to maintain the snapshot liveness invariant.

stacks Thread stacks are ubiquitous and heavily mutated. To make this common case more efficient, we follow the existing collector giving them special treatment. Stacks in GHC are chunked, each chunk being represented by a `STACK` heap object. Any stack chunk currently at the head of a running thread's stack is said to be "dirty", a status which

[24] Arrays also have dirty/clean flags. However, unlike `MutVar#`s, this flag isn't useful for the purpose of maintaining the snapshot invariant since arrays may contain more than one pointer.

is indicated via a flag in the `STACK` closure. Dirty stacks are unconditionally on the generational remembered set to account for possible references into the nursery.

To ensure that the snapshot invariant is respected we must eagerly mark a stack chunk's entries before it is dirtied. Note that this only requires that we pause mutation to mark stack chunks that are dirtied. Chunks that are not dirtied can be marked concurrently.

However, since the structure of stack entries may change in the course of mutation, we must ensure that marking by the concurrent collector does not proceed concurrently with mutation. That is, when the mutator dirties a stack it must wait for any active concurrent marking to finish before commencing mutation. This is achieved via a locking handshake using the stack's dirty field.

source reference: `rts/sm/Storage.c:dirty_STACK`

thread state objects (TSOs) Thread state objects contain all of the state of execution of a Haskell thread. This includes a stack, scheduler state, STM transaction metadata, and a blocking queue listing threads blocked on thunks the thread is currently evaluating. Like stacks, TSOs are treated specially under the existing collector, a thread's TSO being marked as dirtied whenever the thread is scheduled for execution whenever the thread is scheduled for execution.

source reference: `rts/sm/Storage.c:dirty_TSO`

(proposed) mutable constructor fields Simon Marlow's mutable constructor fields proposal describes a facility for first-class mutable constructor fields in GHC. If this proposal is implemented, the code generated for such field mutations will need to contain the same write barrier as is used in array mutation.

2.5.10 Collector selection

As some of the above changes affect code generated by GHC, we have a choice in implementation strategy:

a. Make the choice between copying and non-moving collectors a compile-time decision by introducing a new *way* (e.g. as currently the profiling, threaded RTS, and eventlog options are handled)

b. Always compile in both collectors and allow the decision to be made at runtime, accepting that the unused codepath will incur a small runtime and size overhead.

While (a) is conceptually simpler and avoids any chance that the current collector will regress, it would essentially bring the number of GHC's support ways from 4 to eight, which greatly complicates packaging and distribution of the compiler, as well as downstream tools' interactions with it.

Consequently, we will instead opt to always compile in the new barriers. This will mean that each `Array#` write will incur an extra branch conditioned on the barrier enable flag. The code generator will produce code to ensure that this branch is predicted *not-taken*, ensuring minimal impact on the performance of the copying collector (which will never enable the barrier). We expect the code-size overhead will be manageable.

Additionally, for the sake of testing we will place the new collector behind a CPP macro, allowing it to be removed entired from the runtime system.

2.5.11 Collection scheduling

There are numerous options for garbage collection scheduling policy. GHC's current implementation uses a simple growth rate heuristic (tuned with the `+RTS -F` flag) to determine major collection scheduling. The same heuristic will be initially used to schedule collections by the non-moving collector. Additional policies can be explored as necessary.

Additionally, in the case of concurrent collection there is the possibility that the mutator will run "faster" than the collector. To ensure that the heap size doesn't grow in an uncontrolled manner, we will cap the amount that the mutator can allocate over the course of a given non-moving collection. If the mutator exceeds this limit, it will be forced to pause by the nursery collector.

2.5.12 Update remembered set

The generational nature of GHC's current moving collector requires that references from old generations to young generations are tracked via a remembered set which we have here called the "generational remembered set". The current implementation attaches such a remembered set to every heap generation.

The non-moving collector will additionally require a remembered set for pointers contained by objects living in the non-moving heap who have been overwritten during the mark phase, UpdRemSet.

This remembered set will enforce the following write barrier: Consider that the mutator wishes to write a reference to object `p` to pointer field `O->payload[i]`:

1. The mutator does the usual generational remembered set update
2. The mutator checks whether the non-moving write barrier has been enabled by the mark phase; if not, then proceed to (4)
3. If the write barrier is enabled:
    a. The mutator checks the block descriptor of `O`'s containing block; if it is not in the non-moving generation then proceed to (4)
    b. Otherwise, add a value of the `O->payload[i]` to UpdRemSet, along with its origin closure and field.
4. Set `O->payload[i] = p`

This is the only data structure in the non-moving collector which undergoes concurrent access by both the mutators and collector and it consequently demands careful design.

For this we propose a chunked, dense array representation. Under this design, each capability will maintain an accumulator where its local remembered set is accumulated. This accumulator will take the form of a "chunk", represented by a dense array of pointers. When a capability has filled its local accumulator, it links the chunk into a global linked list, RSetChunks, for consumption by the non-moving collector.

To safely proceed to the sweep phase, the collector must first ensure that all capabilities have flushed any pending accumulator chunks to RSetChunks. To ensure this, the following handshake is carried out between the collection initiator and the remaining capabilities at the end of mark phase:

1. the initiator initializes a global semaphore with value 0

2. the initiator signals all capabilities to enter the garbage collector using the existing `Capability.interrupt` flag
3. on hitting a yield point, each mutator will enter the garbage collector, add its pending remembered set chunk to RSetChunks, and block
4. the initiator will mark chunks as they are added to RSetChunks
5. when all capabilities have signaled the semaphore and RSetChunks is empty, the initiator unblocks the capabilities and proceeds to the sweep phase

This design optimizes for cache locality and amortises expensive inter-thread synchronization over many set additions to minimize the overhead of the write barrier on the mutator.

One potential optimization is to allow capabilities to mark their own remembered set chunks when they are paused instead of deferring this to the initiator.

2.5.13 Eager promotion

The current moving collector employs an eager promotion policy for inter-generational references. This promotes closures residing in generation 0 but referred to by an older generation directly to the older generation. This is an important optimization as this situation arises frequently due to updates of old thunks. This should continue to work under the non-moving collector.

2.5.14 Cost-center profiler

GHC's cost-center heap profiler relies on the garbage collector to gather heap census samples. This census functionality must be implemented in the mark phase of the non-moving garbage collector.

Furthermore, `processHeapClosureForDead`, which collects data for the biographical profiler, must be adapted to include the marking bitmaps of the non-moving heap and called during the sweep phase.

2.5.15 Compact regions

The compact normal-forms mechanism introduced in GHC 8.0 provides a means for users to collect closed heap subgraphs into a single heap region. Not only does this improve garbage-collection efficiency by removing the need to trace pointers within the region, but it can also serve as an efficient means of serialization. Compact regions are represented in the heap by blocks bearing the `BF_COMPACT` flag. The non-moving mark phase will need to set the `BF_MARKED` flag on such blocks during marking.

2.5.16 Stable pointers

GHC's `StablePtr` facility allows the user to create a stable handle which can be passed to foreign code as an opaque reference to a Haskell value. This is implemented in the runtime system using a dictionary that maps stable pointer handles to their referents. The moving garbage collector considers entries in this dictionary to be roots.

While the stable identity provided by this dictionary indirection isn't strictly needed in the non-moving heap (as heap object addresses are stable by default), it is still necessary as we retain the moving nursery. Consequently, the handle dictionary indirection will remain unchanged.

The concurrency introduced in the non-moving collector does have another minor effect: GHC's stable pointer creation path will need to be modified to add `StablePtr`s created during the mark phase to the UpdRemSet to maintain the snapshot liveness invariant.

2.5.17 Weak and foreign pointers

GHC's `ForeignPtr` mechanism allows the user to register finalizers to be run after a heap object dies. The copying collector currently implements this in terms of the runtime's `WeakPtr#` facility [Elliott1999]. Despite what the name suggests, GHC's `WeakPtr#` provides significantly more than the traditional non-traced pointer:

- a weak pointer may have both a key and a value, providing a solution to the key-in-value problem discussed in [Elliott1999]
- a means of attaching a finalizer to be run when the value becomes unreachable; two types of finalizers are supported:
    - A Haskell finalizer (of type `IO ()`) may be defined when the `WeakPtr#` is created with `mkWeak#`
    - C finalizers may be added to an existing `WeakPtr#` with `addCFinalizerToWeak#`

To implement `WeakPtr#`, the runtime system maintains a list of weak pointers, WeakPtrList. Adjusting the non-moving mark/sweep collector for weak pointers involves adjusting the mark phase slightly (paraphrasing [Elliott1999]):

1. mark the heap reachable from the roots, as usual
2. for each weak pointer in WeakPtrList with a marked key, mark the heap reachable from its value field or its finalizer and move the weak pointer to a new list, WeakPtrList$'$.
3. repeat step (2) until no WeakPtrList entry has a marked key
4. for each weak pointer in WeakPtrList:
    - replace the `WeakPtr#` with a tombstone if it is marked
    - if the weak pointer has a finalizer: move the pointer to a new list, FinalPendingList, and mark the heap reachable from the finalizer.
5. set WeakPtrList := WeakPtrList$'$.
6. mark any unmarked weak pointers on WeakPtrList so they aren't discarded
7. spawn a finalization thread to execute the finalizers in FinalPendingList.
8. proceed to the sweep phase

Note that there are a few subtleties due to the generational collector: the copying collector does not finalize weak pointers living in older generations than the generation currently being collected. However, note that an object's key may still live in an older generation. In this case the moving collector's closure liveness predicate, `isAlive` (defined in `rts/sm/GCAux.c`), returns true as a conservative estimate of the key's liveness. `isAlive` will need to be similarly modified for the nonmoving collector, returning true for objects that live in the nonmoving heap. The weak marking logic will then need to handle major GCs specially. See Section 2.5.2 for details.

Note that the `link` field of `StgWeak` can be safely used by the non-moving collector for weak objects residing in the non-moving heap. This is because the copying collector will only attempt to link weak objects living in the generation it is collecting into its weak object list.

Also note that the possibility of aged objects poses a bit of complexity. Specifically, if one has a weak pointer living in the nonmoving heap with a key in the moving heap. In this case, unlike the moving collector, we must be certain to evacuate the weak pointer's key when it is scavenged.[25] This ensures that the key is still valid when we later go to check its liveness when we tidy the weak.

2.5.18 Stable names

The `StableName` facility [Elliott1999] allows the user to associate a stable abstract name, embodied by a Haskell `StableName` value, with a Haskell value. This facility shares a great deal of implementation with the stable pointers facility described in Section 2.5.17. However, stable name management additionally requires a dictionary of names keyed on referent address to ensure that the `makeStableName` operation is idempotent. The current moving collector must update this map after evacuation as object addresses have changed (see `updateStableTables`). However, since values in the non-moving heap have stable addresses, this will not be necessary for non-moving collection.

Like `mkStablePtr`, `makeStableName` requires a write barrier to add any objects in the nonmoving heap (and the `StableName#` object itself) to the update remembered set while a mark is underway.

2.5.19 Sparks

GHC's "spark" mechanism enables expressions to be added to a global pool, where they can be opportunistically evaluated by idle capabilities. This requires a bit of cooperation from the garbage collector to ensure that sparks that become unreachable (so-called "fizzled" sparks) are removed from the pool.

Because spark evaluation can proceed concurrently with non-moving collection, this will require a read barrier in the spark evaluation code. Specifically, the spark evaluation code check whether a non-moving collection is

2.5.20 Deadlock detection

GHC's garbage collector is in part responsible for identifying mutator deadlocks due to `STM` and `MVar` blocking. When the scheduler realizes it has no threads that can be run it forces a major garbage collection (see `scheduleDetectDeadlock`). This collection will resurrect

[25]There are specifically two ways in which a weak living in the non-moving heap but with a younger key might be scavenged:

a. a weak object that gets promoted during the preparatory collection of a major GC will be scavenged when the to-space segment is scavenged by `scavenge()`

b. a weak object that was already in the major generation when the collection began will be on the generational remembered set and will be scavenged via `scavenge_mutable_lists()`.

However, case (b) cannot actually occur since the key will have been promoted to the non-moving heap when the `WEAK` itself was promoted, as described in (a).

2.5.21 Software Transactional Memory

GHC's STM implementation

2.5.22 Mark queue

Due to the variety of features introduced above, the non-moving mark queue is a bit more elaborate than the usual queue-of-pointers seen in traditional mark-sweep collectors. Specifically, a queue entry is isomorphic to the following Haskell type:

```haskell
data MarkQueueEntry

    -- | Mark a closure referred to from pointer @originField@
    -- of closure @originClosure@
    = MarkClosure { markClosure   :: Ptr Closure
                  , originClosure :: Ptr Closure
                  , originField   :: FieldIndex
                  }

      -- | For efficiently marking Array#s
    | MarkArray { markArray  :: Ptr Array
                , startIndex :: FieldIndex
                }


type FieldIndex = Word
type Ptr a = Word
```

`MarkClosure` encodes the typical case of a "normal" closure in need of marking. However, since we need to need to be able to "short-out" indirections, we also keep a reference to the closure and field containing the pointer (Section 2.4.2).

`MarkArray` allows us to mark `Array#`s while keeping the mark queue of bounded length, as discussed in Section 2.5.3.

2.6 Differences in collection behavior from existing collector

There are a few ways in which the collection behavior of the non-moving collector will differ from that of the existing collector. These are largely due to the concurrent nature of the new collector:

- *Selector thunk chains shorted-out by the selector optimisation won't be reclaimed until a later non-moving collection:* This is necessary due to the snapshot invariant: the selectors were live at the beginning of the mark and consequently we can't free them since a mutator may have introduced a new reference in an object allocated while the mark was underway.
- *Finalization isn't as prompt:* Finalizers attached to weak pointers won't be run in the same collection as the weak was reclaimed. Note that currently GHC makes no guarantees about the promptness of finalization specifically to allow designs like the

non-moving collector. However, some user code may nevertheless rely on the current behavior.

2.7 Optional: Compaction

While the heap layout discussed above is resistant to fragmentation thanks to the size-stratified sub-allocator structure, programs which may trigger fragmentation can nevertheless be constructed. Such programs would exhibit large resident-set sizes with a comparatively small quantity of live data.

If fragmentation is found to be a problem in practice an incremental compaction phase [BenYitzhak2002] can be implemented. Such a phase would be strategically scheduled on subsets of the heap by the collector to reduce fragmentation while minimizing pause times.

2.8 Optional: Allocating small pinned ByteArrays in nonmoving generation

The availability of a non-moving heap offers an interesting opportunity to improve GHC's existing `ByteArray#` pinning mechanism. Pinned `ByteArray#`s allocations are currently accumulated into `BF_PINNED` blocks. This tends to lead to fragmentation since any single `ByteArray#` allocation remaining alive will result in the entire block also remaining alive.

However, this can be avoided by teaching `allocPinnedByteArray#` to allocate small requests directly into the non-moving generation.

2.9 Work plan

The implementation effort will begin in January 2018. It is expected that refactoring the runtime system to accommodate a second garbage collector will take up to two months. After this, three months will be needed to produce a working prototype of the non-moving collector.

Refinement of the prototype will require another four months. This will involve fixing of corner-cases, implementation of profiler support, and further code refactoring. After this work can proceed on characterizing and tuning the performance of the non-moving collector. Depending upon the performance of the collector, we may take a few weeks to introduce parallel collection at this point. The remaining time will be spent optimizing, documenting, and up-streaming the completed implementation.

2.10 Glossary

aging A technique used in generational collectors where objects are retained in a generation for more than one collection cycle before promotion.

promotion The act of moving an object from a young to an older generation in a generational garbage collector.

mutator The user program being serviced by the garbage collector.

garbage collector A subsystem responsible for determining object liveness and releasing dead objects.

generational garbage collector A garbage collector which exploits the observation that most objects are short-lived to reduce garbage collection costs.

nursery/generation 0 The generation into which new objects are allocated by the mutator.

incremental garbage collector A garbage collector which can make progress on reclamation while only processing a fraction of the heap in a given run.

parallel garbage collector A garbage collector which can utilize multiple cores to make progress on collection.

concurrent garbage collector A garbage collector which can run while the mutator runs.

heap fragmentation The tendency for the large blocks of memory to become partitioned into increasingly smaller blocks of free space over time due to repeated allocation and deallocation.

compacting/moving garbage collector A garbage collector which eliminates heap fragmentation by relocating old, fragmented objects into a contiguous block of heap.

bump-pointer allocator A memory allocator which provides blocks of memory by filling a contiguous block of free memory starting from the beginning of the block.

remembered set A set which tracks objects which must be traced in addition to the usual trace set to maintain an invariant of the collector. This proposal discusses two such sets: the *generational remembered set* and the *update remembered set*.

root set The set of objects, generally including thread stacks and , with which marking begins.

write barrier A requirement in service of the garbage collector placed on a mutator thread modifying a pointer.

evacuation In a moving garbage collector: the process of moving a heap object residing in *from space* into *to space*.

scavenging In a moving garbage collector: the process of evacuating or fixing-up the pointers of a previous-*evacuated* heap object.

tracing The process of discovering the objects reachable from a heap object.

marking The process of recording that a heap object is reachable, typically in a mark bitmap.

sweeping The process of walking the heap looking for unmarked objects whose memory can be returned to a free-list for future use by the allocator.

3 References

[BenYitzhak2002] Ben-Yitzhak, Ori ; Goft, Irit ; Kolodner, Elliot K. ; Kuiper, Kean ; Leikehman, Victor: An algorithm for parallel incremental compaction. In: *ISMM* : ACM, 2002

[Boehm1993] Boehm, H.J.: Reducing garbage collector cache misses. In: *ISMM* : ACM

[Cheadle2000] Cheadle, A.M. ; Field, A.J. ; Marlow, S. ; Peyton Jones, S.L. ; While, R.L.: Non-Stop Haskell. In: *ICFP*, 2000

[Cher2004] Cher, C.Y. ; Hosking, A.L. ; Vijaykumar, T.N.: Software prefetching for mark-sweep garbage collection. In: *ASPLOS*

[Detlefs2004] Detlefs, David ; Flood, Christine ; Heller, Steven ; Printezis, Tony: Garbage-First Garbage Collection. In: *ISMM* : ACM

[Elliott1999] Peyton-Jones, Simon ; Marlow, Simon ; Elliott, Conal: Stretching the storage manager: weak pointers and stable names in Haskell. In: *International Workshop on the Implementation of Functional Languages* : Springer-Verlag, 1999

[Garner2007] Garner, Robin ; Blackburn, Stephen M. ; Frampton, Daniel: Effective Prefetch for Mark-Sweep Garbage Collection. In: *ISMM* : ACM, 2007

[Johnstone1998] Johnstone, Mark S. ; Wilson, Paul R.: The memory fragmentation problem: solved? In: *ISMM* : ACM, 1998

[Kermany2006] Kermany, Haim ; Petrank, Erez: The Compressor: Concurrent, incremental, and parallel compaction. In: *PLDI* : ACM

[Marlow2007] Marlow, Simon ; Rodriguez Yakushev, Alexey ; Peyton Jones, Simon: Faster laziness using dynamic pointer tagging. In: *ICFP*

[Marlow2008] Marlow, Simon ; Harris, Tim ; James, Roshan P. ; Peyton Jones, Simon: Parallel generational-copying garbage collection with a block-structured heap. In: *ISMM*

[Tene2011] Tene, G. ; Iyengar, B. ; Wolf, M.: C4: the Continuously Concurrent Compacting Collector. In: *ISMM* : ACM

[Ueno2016] Ueno, Katsuhiro ; Ohori, Atsushi: A fully concurrent garbage collector for functional programs on multicore processors. In: *ICFP* : ACM, 2016

[Ungar1984] Ungar, D.: Generation scavenging: A non-disruptive high performance storage management reclamation algorithm. In: *SIGPLAN Software Engineering Symposium on Practical Software Development Environments* : ACM

[Wadler1987] Wadler, Philip: Fixing some space leaks with a garbage collector. In: *Software Practice and Experience* Bd. 17, Nr. 9