Abstract
Over the years, traditional tracing garbage collectors have accumulated assumptions that may not hold in new language designs. For instance, we usually assume that run-time objects do not hold addressable sub-parts and have a size of at least one pointer. These fail in systems striving to eliminate pointers and represent data in a dense, serialized form, such as the Gibbon compiler. We propose a new memory management strategy for language runtimes with mostly serialized heaps. It uses a hybrid, generational collector, where regions are bump-allocated into the young generation and objects are bump-allocated within those regions. Minor collections copy data into larger regions in the old generation, compacting it further. The old generation uses region-level reference counting. The resulting system maintains high performance for data traversal programs, while significantly improving performance on other kinds of allocation patterns.

CCS Concepts: • Software and its engineering → Compilers; Garbage collection.

Keywords: Compilers, Garbage Collection, Data Representation

https://doi.org/10.1145/3652024.3665512

1 Introduction
Since manual memory management is extremely error-prone, high-level languages have, for several decades, employed automatic memory management (AMM), such as tracing garbage collectors and reference counting. During this time, the assumptions about the target language runtime varied just like the design of languages themselves, and the particulars of an AMM system is often influenced by the language it is attached to. Attempts to design generic memory systems have appeared over time but they still reflect assumptions from their originating languages and implementations, and new language designs can prompt new, unforeseen challenges for AMM.

Most established AMM designs target language runtimes where program values are represented with pointers to small objects allocated sparsely on the heap. These runtime objects are atomic in two senses: first, they never change their shape, and second, there are no sub-parts of the objects that are individually addressable—the targets of pointers elsewhere in the heap. Another common assumption is that an object has the size of at least one pointer.

Recently, a novel approach to representing program values as dense structures (i.e. pointer-free, serialized byte arrays) has been studied. This has huge performance advantages [11, 15, 18] due to minimized scattered memory accesses and maximized data locality. Several works have explored computing directly with serialized data representations of varying density: Cap’N Proto [27], FlatBuffers [12], Compact Normal Form (CNF) [30], Dargent [7]. Such efficiency is also achieved by the Gibbon compiler [29] for a polymorphic, higher-order, strict subset of Haskell. Gibbon compiles programs to C code and employs whole-program compilation and monomorphization.

In Gibbon’s implementation, the heap hosts dense structures residing in regions, which represent growable units of allocation. Each region contains a single value (corresponding to a logical value from the source language), which is made up of numerous objects (allocations corresponding to data constructors); e.g., a list value can consist of several cons-cell-objects. Objects in the same region are packed side-by-side and pointers between them are the exception rather than ubiquitous1.

1Consider, for example, the following definition of the List datatype:
data List = Cons Int List | Nil. The Cons data constructor has the Int value inlined immediately after a one-byte constructor tag; the tail is treated the same and inlined immediately after the 8-byte integer. So, a list with 100 elements can fit in 901 contiguous bytes: 9 × 100 plus one byte for the Nil constructor.
With densely represented values, pointers have to be employed sometimes to recover some of the flexibility or space efficiency (sharing) of the traditional pointer-based approach. E.g., in Gibbon, so-called indirection pointers preserve asymptotic space and time complexity, e.g., when compiling a program that shares common data between two values. The compiler automatically extends the datatype to include a tagged indirection pointer, e.g. \textit{Ind(}Ptr\List{)}, as an implicit extra case in the sum type. Crucially, such pointers allow objects to refer to objects in other regions and make Gibbon heaps only mostly serialized.

Prior work on AMM for mostly serialized heaps used region-based memory management (MLkit [23], UrWeb [9]). In these works, object lifetime is determined automatically by the compiler and it depends on the lexically-scoped region the object is assigned to. Object lifetimes must be conservative and may be overly long, in the worst case, equivalent to the entire program execution. In pathological cases, this can lead to unbounded space leaks [23]. Gibbon’s approach is similar, but indirect pointers can cause objects to stay alive beyond their region’s lexical scope, thereby prompting the need for an additional region-level reference-counting scheme [14]. In contrast, traditional tracing garbage collectors, while not garbage free, can bound the heap size using the semi-space strategy [8]. For that reason, MLKit later added backup garbage collection within regions [10].

Tracing collectors are attractive but cannot be used with mostly serialized heaps directly. Supporting denser data representations violates some of the usual assumptions baked into many memory management systems, namely: (1) the heap consists of objects of statically-known sizes (plus arrays) connected by pointers, (2) pointers always refer to the starting address of the target object, and (3) every data constructor (e.g. a node in a tree or a cons-cell in a list) is its own independent object in memory at runtime.

This paper tackles the above challenges and presents an AMM system for mostly serialized heaps in the context of the Gibbon project. Our contributions are as follows:

- The first practical and complete solution to automatic memory management in mostly serialized heaps. We adopt a generational collector design, with copying collection for the young generation while keeping Gibbon’s reference-counted regions for the old generation. This hybrid approach (similar to Ulterior [3]) enables fast bump-allocation of new objects and regions, while retaining efficient handling of large and growing regions.
- Novel design choices prompted by our unusual setting. We introduce new regions during collection time, to keep all roots valid after collection (Section 3.1). Because values continue growing after promotion to the old generation, we allow allocation into old-generation regions (Section 3.3), which, in turn, incurs a need for a remembered set. We allow sharing, in spite of the fact that many objects are smaller than a word-sized pointer, and thus develop a new approach to forwarding pointers (Section 3.4).
- An evaluation of the resulting system—GC-Gibbon—on both tree-traversals, which are favorable for the serialized data approach, and benchmarks that stress small object allocation, where Gibbon used to be at its weakest (Section 5.1). We show that GC-Gibbon retains Legacy-Gibbon’s strong performance on tree-traversals and attains reasonable performance on out-of-order, small allocations, approaching mature systems. For small allocations, our system is 3.79, 0.46, and 1.09 geomean faster than Legacy-Gibbon, GHC, and Java, respectively. For tree-traversals, our geomean speedup is 1.02, 2.19, and 1.5. Gibbon’s memory consumption reduces to 47% (geomean) on the small allocation benchmarks. We also evaluate our design choices in Section 5.2.

2 Anatomy of Mostly Serialized Heaps

In this section we overview mostly serialized heaps as implemented in Gibbon and give examples when the approach shines or falls short.

To serialize the data representation Gibbon uses LoCal [14, 28] (Location Calculus), a first-order intermediate representation (IR) with an explicit byte-addressed, mostly serialized data layout. Gibbon converts input programs into LoCal programs using location inference (a variant of region inference [23, 25]), and thereby makes explicit the byte-level data layout of all values.

Regions, objects, and chunks. In LoCal, all values reside within regions. Every region itself resembles a heap with a single allocation cursor, which is used to perform all writes in it. The allocation cursor always points to the next available cell on the heap, and new objects are bump allocated. Locations are addresses within a region that are used to read and write raw data. LoCal always allocates objects in a region in order—every location in a region must be written to before a new writeable location can be created after it. This
serial ordering imposed on locations is what serializes the resulting value.

There are two subcomponents to each object: there is the fixed portion of an object, consisting of a tag, and any constant-sized fields such as integer scalars. Then there is the extended portion of the object which fills a variable number of bytes due to child objects being “inlined” within its representation. Even the extended object may be smaller than the complete logical value, which would include all memory reachable by the transitive closure of any indirections.

Gibbon allocates a constant-sized chunk of contiguous memory for each fresh region, as Figure 1 shows. When this chunk is exhausted, a new one which is double in size is allocated and linked with the previous one using a pointer. This doubling policy is used up to an upper bound, after which constant-sized new chunks are allocated.

In order to detect if a chunk is exhausted, every write operation needs to know where the current chunk ends, so that it can perform bounds checking. For this reason, every location is dilated to be a pair of (alloc, end) cursors at run time. To mark that the serialized stream of data continues in another chunk Gibbon implicitly adds yet another reserved constructor—and thus one-byte tag value—to each datatype: (EoC Ptr), which signals an end of chunk and stores a pointer to the head of the next chunk. We refer to these pointers as end-of-chunk pointers. When a reader hits an EoC tag, they must use the stored pointer to resume reading. We refer to read or write pointers into chunks as cursors because of their largely contiguous motion.

Sharing via tagged indirection pointers. Tagged indirection pointers enable a program to share a value among multiple locations. For example, one might write code to construct a binary tree node as

\[
\text{let } x = \text{buildTree } n \text{ in Node } n \times x
\]

and expect a single, shared value to be allocated for the left and right subtrees. But without a way to store the addresses of previously allocated values in the data representation, the right subtree would have to be allocated by copying the entire left subtree at the appropriate location. To avoid copying, Gibbon implicitly compiles every datatype \(d\) to have an additional reserved constructor (Ind (Ptr d)), which stores an absolute pointer to a value of type \(d\). Given such additional constructors, Gibbon can compile the above code to the following:

\[
\text{let } x = \text{buildTree } n \text{ in Node } n \times (\text{Ind (addrOf } x))
\]

In this version, Gibbon can ensure that the call to buildTree allocates directly in place, in the left subtree after the Node tag. Thus a pointer is needed only for the right subtree:

\[
\text{data List = Cons Int List | Nil}
\]

\[
\text{mkList } 0 = \text{Nil}
\]

\[
\text{mkList } n = \text{Cons } n \ (\text{mkList } (n-1))
\]

Figure 2. Gibbon source code for \text{mkList} and in-memory representation of its result. \(C\) is short for \text{Cons}.

\[
\text{reverse } \text{Nil } \text{acc} = \text{acc}
\]

\[
\text{reverse } (x:xs) \text{ acc} = \text{reverse } xs \ (x:\text{acc})
\]

Figure 3. Gibbon source code for accumulator-style list reversal and in-memory representation of its result. \(C\) is short for \text{Cons} and \(N\) is short for \text{Nil}.

Indirection pointers are critical to Gibbon’s ability to compile functions without changing their asymptotic complexity. They provide “opt in” pointers, with the default case being dense serialization, and the exceptional case being indirection. The runtime overhead (branching on a one byte Ind constructor) is placed on the exceptional case.

The Splendors and Miseries of Gibbon. Dense representations of data are sensitive to allocation patterns of algorithms in use. Here we show one example where Gibbon is at its best—in-order allocations—and another one where Gibbon’s approach suffers the most—out-of-order allocations.

Figure 2 shows the Haskell source for a simple \text{mkList} function, as well as the resulting structure in memory. Thanks to the destination-passing style [21] LoCal employs, Gibbon is able to produce densely represented list of integers in the first chunk of the output region. If the list does not fit in one chunk, Gibbon will allocate more and connect the two together. This representation is still far more efficient than the traditional linked structure where every node involves an isolated allocation.

Gibbon programs use bump allocation in regions to achieve efficient, linear allocation patterns, such as that of \text{mkList}, where the result value, the constructed list, is laid out in a single region. But if a function’s allocation pattern differs, such as when reversing a linked-list or when building a binary tree by allocating its right subtree before the left, the resulting value is placed across multiple regions. Consider the Haskell

\[
\text{reverse Nil acc} = \text{acc}
\]

\[
\text{reverse (x:xs) acc} = \text{reverse xs (x:acc)}
\]

\[
\text{let } x = \text{buildTree } n \text{ in Node } n \times x
\]

\[
\text{data List = Cons Int List | Nil}
\]

\[
\text{mkList } 0 = \text{Nil}
\]

\[
\text{mkList } n = \text{Cons } n \ (\text{mkList } (n-1))
\]

\[
\text{let } x = \text{buildTree } n \text{ in Node } n \times (\text{Ind (addrOf } x))
\]

\[
\text{data List = Cons Int List | Nil}
\]

\[
\text{mkList } 0 = \text{Nil}
\]

\[
\text{mkList } n = \text{Cons } n \ (\text{mkList } (n-1))
\]

\[
\text{let } x = \text{buildTree } n \text{ in Node } n \times (\text{Ind (addrOf } x))
\]

\[
\text{data List = Cons Int List | Nil}
\]

\[
\text{mkList } 0 = \text{Nil}
\]

\[
\text{mkList } n = \text{Cons } n \ (\text{mkList } (n-1))
\]

\[
\text{let } x = \text{buildTree } n \text{ in Node } n \times (\text{Ind (addrOf } x))
\]
code for the standard accumulator-style list-reverse function given in Figure 3. For every successive recursive reverse call, Gibbon has to allocate a new region for the new accumulator value, since it does not have any other region to use (the destination region for the result cannot be used yet because the intermediate values of the accumulator do not have an obvious connection to the overall result). As a result, every output cons cell goes in a separate region, linked together using indirection pointers—a traditional linked list.

Programs like reverse which allocate small regions at a very high rate show the overheads of region allocation and collection in Gibbon. In fact, the list-reverse program compiled by Gibbon is $4x$ slower than its pointer-based version. This isn’t surprising because the Gibbon program performs the same number of region-malloc’s as the pointer-based version does for objects, but also does additional work to track region metadata information such as the reference counts and outsets (pointers to regions to which objects in this region point, shown in Figure 1).

Another problem that is highlighted in Figure 3 is that of fragmentation. First, reverting to a pointer-based representation means all subsequent traversals of this list would be slower because of poor data locality and pointer chasing. A little slower than in a traditional pointer-based heap, because of processing the extra tag-check on each indirection pointer. Second, the space usage is not efficient, with only one Cons cell per region, most space is left unoccupied.

3 Design of GC-Gibbon

GC-Gibbon adopts a copying generational collector with compacting ability for its young generation and keeps Gibbon’s reference-counted regions for the older generation. The old regions, however, use deferred reference counts now.

All new chunks are allocated in the young generation by bumping the allocation pointer. When the young generation becomes full (i.e., exceeds a certain threshold), it has to be collected: live data is copied to the old generation and the young generation is reset. We retain Gibbon’s reference-counted regions for the old generation. After copying is complete, any old-generation regions that are dead as per the deferred reference counting mechanism are freed. In the future we plan to extend this with an additional tracing collector to collect regions in the old generation.

In the next sections, we explain how we treat a minor collection: tracing the young generation starting from the roots and copying to a different region representation in the old generation where regions are granted their own growing, memory allocations, and equipped with extra metadata.

3.1 Managing GC Roots

Shadow Stack. We use a shadow-stack [13] to maintain a root set of live objects for collection. Following the convention, addresses of objects (locations in LoCal) that are live after an allocation point are spilled to the shadow-stack, and are restored later, potentially having been updated by an intervening collection. Spilling is also necessary across function call sites, as the function might perform allocations.

Along with the spilled location, we also spill and restore the end-of-chunk address for the chunk that the spilled location belongs to. There are two reasons: (1) if the location is in a young-generation chunk, it will be promoted to the old generation, and its end address will change, and (2) if it’s in an old-generation chunk, the collector would require its end address to access the region-level metadata (reference count, outset etc.) whose address is stored at the footer. Also, the type of the value residing at the spilled location is stored on the shadow-stack. It serves as a numeric index into an infotable (described in Section 4) that stores layout information required to guide the evacuation routine.

Evacuation order of GC roots. Like other tracing collectors, the order in which roots are processed influences the layout and locality of the resulting old-generation heap [1]. However, our collector is even more sensitive to this ordering: certain traversal orders produce a compact heap with no indirection pointers, while some others produce significantly pointer-heavy heaps.

To illustrate sensitivity of our collector to the root ordering, we consider the young generation and the root set in Figures 4a and 4b. The young-generation object is identical in both cases, but the order of GC roots corresponding to the child objects is different. In Figure 4a, the oldest root corresponds the parent object, and younger roots correspond to child objects deeper in the graph. In Figure 4b, the roots are ordered in the opposite order. This ordering of roots in the
root set is a consequence of different workloads: in-order allocators like mkList tend to create root sets like Figure 4a, while reverse-order allocators like reverse tend to create root sets like Figure 4b. If a collection is triggered at this point, there is no fixed order of root set traversal (e.g., newest-to-oldest versus oldest-to-newest) that would handle both examples efficiently\(^3\). For these list examples, an efficient traversal starts evacuation with the head of the list, promoting the entire list into a single old-generation region (Figure 4a). An inefficient traversal would evacuate the tail of the list first, and then subsequently evacuate earlier parts of the list which would be linked by (unnecessary) indirections to the already-evacuated portions (Figure 4b).

The key insight here is to pick an ordering that consistently evacuates upstream data earlier, irrespective of the order of allocation. Specifically, we want a traversal order that (1) evacuates roots for the newer regions before the older regions, and (2) within each region, evacuates roots that are towards the beginning of the region before roots that are towards the end.

We get a favorable ordering as follows. First, we bump the allocation pointer of the young generation backwards. That is, the allocation pointer starts at the end of the young generation (the high address), and moves towards the start (the low address). Next, we sort the root set such that roots corresponding to objects at lower addresses appear before those at higher higher addresses. This policy gives us both the desired properties because objects towards the beginning of the region already occupy addresses lower than those of objects towards the end, and the reversed bump allocation puts newer regions at addresses lower than those of older regions.

For a desirable post-collection heap, indirections should only be proportional to actual sharing in the data. To quantify this notion more precisely, we define an optimal heap with minimum indirection count:

**Definition 3.1** (Minimal indirections post-collection). The minimum number of indirections post-collection includes: up to one per root, plus \(N−1\) indirections for every object which has \(N > 1\) references to it.

If a live object has only a single reference to it, then it must be placed in the same region as the object from which it is reachable. The per-root indirections are there because the roots themselves represent pointers into the heap data. Further, we place these pointers in fresh regions only if the object has not been copied already, otherwise, the root is directly updated to point to the object’s new address.

Our post-collection heaps always achieve minimal, sharing-only indirections, as in Definition 3.1. This compactness comes at the cost of sorting the roots. In practice, root sets are small enough that the time spent on sorting is worth the resulting improvement. Also, the root set is bounded by the program stack size. If we wanted to further bound sorting time to a constant, we could use a partial sorting algorithm, or, completely skip sorting, trading it off against having more indirections in the post-collection heap.

### 3.2 Bird’s-Eye View of Evacuation

We now sketch the algorithm to evacuate each root in the root set. Here we focus on evacuating completely written objects, deferring the discussion of partially-written objects to Section 3.3 and sharing to Section 3.4.

The collector processes all roots present on the shadow-stack one by one. For each root, it first checks whether a young-generation object has already been evacuated, in which case it updates the root to have the the corresponding addresses of the relocated data in the old generation. Otherwise, it allocates a fresh region in the old generation to relocate this object. This fresh region can grow during collection, using the normal policy of doubling the size of each additional chunk in the linked series of chunks. Then, the collector learns the kind of object it is evacuating by inspecting the first tag in the object, based on which it carries out the evacuation. Once evacuation is complete, the start and end addresses of the root are updated. The pseudo-code of the evacuation algorithm is given in Appendix D.

The first tag of an object defines this object’s kind and informs evacuation. Below we explain the strategy for every object kind.

**Tagged indirection pointer:** If the target of the pointer is in the young generation, it is inlined by copying its data. Otherwise, the indirection pointer is copied as it is. In the latter case, the indirection pointer written by the collector is an old-to-old pointer, and the reference count of the region containing the target object must be updated as our (deferred) reference counts include only old-generation pointers.

Updating reference counts requires accessing the target region’s metadata, which must be findable given only the contents of the tagged indirection pointer being copied. To accomplish this we use the 64-bit indirection pointers to pack in both (1) the address of the target object, and (2) the offset from there to the target chunk’s footer—which is where the region metadata resides. We give details of this pointer encoding in Section 4.

Recall that the mutator uses indirection pointers to share objects among different regions. There is additional work required here to carry forward this sharing into the old generation. We postpone this discussion to Section 3.4.

**End-of-chunk pointer:** Due to the pretenuring policy (Section 3.3) the target of an end-of-chunk pointer is always in the old generation, and thus an object is considered completely evacuated upon reaching this tag. But, we still need to combine metadata information for two regions: (1) the fresh region that was created to copy the object that ends...
in this end-of-chunk pointer, and (2) the old-generation region that was created earlier due to pretenuring to store the remainder of this object. For this reason, all end-of-chunk pointers also need to encode an offset from their target to the target chunk’s footer.

**Burned or forwarded object:** Object that has already been evacuated, discussed in Section 3.4.

**Cauterized object:** Partially-written object, discussed in Section 3.3.

**Regular data constructor:** If the tag is not among the reserved tags listed above, it corresponds to an allocation of a regular data constructor and is copied by referring to the info-table (Section 4). Its fixed portion, consisting of a tag and constant-sized fields, is copied directly using `memcpy`. The extended portion, consisting of child objects occupying variable number of bytes, is processed by recursively inlining the children into the destination region.

### 3.3 Growing Partially-Written Objects

Certain objects encountered by GC-Gibbon’s copying collector might be only partially written. For example, the mutator could be in the middle of allocating a tree structure when it triggers a collection, which leaves the young generation with a region containing a tree node having a left field but no right field (yet). When such a tree node is promoted to the old generation, the collector must stop copying after the left field, otherwise it will keep reading uninitialized data. Thus the collector must be able to detect the ends of such partially-written objects. Furthermore, once the minor collection is complete the collector must decide where to grow this object, that is, where to restart construction of the remainder of the object (the right field)—in the young or in the old generation.

We use a region’s allocation cursor (Section 2) to detect partially-written objects. Every region has a single allocation cursor, where the next object would be written. As a corollary, if a region does not have an allocation cursor, it cannot have any partially-written objects in it. Before beginning copying, the collector writes a special reserved tag at all live allocation cursors, effectively *cauterizing* the regions to mark the end of initialized data. The copying routine described in Section 3.1 stops copying upon reaching this tag, so as to not read any uninitialized data. To support this, the mutator spills all live writeable locations to a separate shadow-stack before starting a collection, and restores their updated addresses after the collection is complete. Regions that contain a fully constructed value do not have an allocation cursor, as they do not have any writeable locations in them. Correspondingly, such regions do not undergo cauterization and the live objects within them are promoted in the standard way.

The design choice of where to restart construction of the remainder of partially-written values is a tradeoff between (1) requiring a write barrier for new indirection pointers written into the old generation, and (2) sacrificing the benefits of pretenuring large and growing regions into the old generation.

**Design choice 1: Restarting construction in the young generation:** We would allocate one young-generation chunk for each partially-written object that was promoted to the old generation, and, to continue its construction, update its allocation cursor to point to the beginning of the new chunk. Next, we would use an end-of-chunk pointer to link the end of the promoted object to the beginning of this fresh young-generation chunk. These end-of-chunk pointers (pointing from the old generation to the young generation) would serve as a *remembered set* of roots for the subsequent minor collection. However, crucially, all writes would now always happen in the young generation. As a result, and since Local is a pure language, this remembered set would remain constant until the subsequent collection. Moreover, all new indirect pointers would be young-to-old pointers, and could be created without a write barrier (for maintaining a separate remembered set), making them fairly cheap to create. With this policy, we would use a remembered set of end-of-chunk pointers that is updated once per collection, instead of a remembered set of indirection pointers that is maintained by the mutator using a write barrier.

While this policy reduces the cost of indirection pointers, it precludes the collector from performing pretenuring [5, 19, 26], which requires that the mutator be able to allocate certain long-lived objects directly in the old generation. This would have a significant impact on the performance of programs that allocate large values, often using a small number of regions. Such programs would exhaust the young generation frequently and trigger a collection. Moreover, programs that have a large number of regions under construction simultaneously could cause an exceptional situation where after the minor collection, a large portion of the young generation is populated by these new chunks for older promoted values, also increasing the number of collections (unless the young generation is allowed to grow).

**Design choice 2: Restarting construction in the old generation:** In GC-Gibbon we use the dual of the previous choice, namely to continue the construction of partially-written objects in the old generation. After an object is promoted, its allocation cursor is updated to point to the frontier of its old-generation chunk, and no new chunks are created in the young generation. As discussed above, the choice to allow allocations directly in the old generation has two consequences: the collector can perform pretenuring, but pointer creation needs to be protected by a write barrier since the remembered set of indirections pointing from the
Garbage Collection for Mostly Serialized Heaps

An object can have more than one inbound pointer, and thus a tracing collector must maintain sharing as it relocates data. For example, the rootset can contain stack variables with multiple pointers into different parts of the same object, as is the case when the mutator recurs through multiple levels of a tree. If we fail to detect sharing while copying live data, all the local variables on the stack may end up with their own copy of the data, which is wasteful.

There are several challenges to maintaining sharing given a dense, mostly serialized heap. First, there is insufficient space for forwarding pointers inside many objects’ layouts. Second, when the collector is mid-way through copying a object, and finds a sub-portion of it has already been copied, it needs a skip-over address to skip over the already-copied portion and resume copying after it. The simplest solution to both these challenges is to maintain two side-metadata tables during collection: (1) to store all forwarding information, i.e. a table that maps (start,end) intervals in the young generation to their corresponding addresses in the old generation, and (2) another table to store skip-over addresses. Unfortunately, using these tables naively is expensive and makes the collector several times slower. Instead, we explore an approach that stores this metadata in the copied portion of objects where possible, and only uses side-metadata tables as a fallback (the slow path), as we explain next.

---

5The remembered set can never shrink, because pointers in the old generation cannot be deleted by the mutator as LoCal is a pure language.
an indirection pointer can be computed in a straightforward manner: we use the forwarding pointer that will be written in place of the old indirection.

Consider a scenario where the from-space heap is as illustrated in Figure 5b and the collector follows “inbound indirection pointer 1” and reaches the burned tag of object 1 at byte-address 9. It will now scan to the right and immediately discover a forwarding pointer at byte-address 10. This forwarding pointer points to the to-space byte-address 1010. The collector will compute the forwarded address of the object at byte-address 9 as: 1010 - (10 - 9) = 1009.

Our goal in designing the sharing-preservation aspect of our collection algorithm is to bound the amount of scanning time necessary to resolve a forwarded address for any tag residing in the from-space before collection. Nevertheless, for completeness we need to also introduce a global table as the place of last resort to store forwarding information. The table maps (start,end) intervals in the from-space to their corresponding start locations in the to-space. The collector may enter in the middle of a burned interval, so the table needs to be an interval map allowing forwarding-address lookups keyed on locations anywhere within the forwarded interval.

We fall back to the table when we fail to find forwarding information by scanning. We can fail by hitting the end of the span, which we recognize based on encountering an already-burned tag of the next span. The collector also exposes a max\_scan parameter for the maximum number of burned bytes that should be traversed in search of a forwarding pointer. By default this is set to the young-generation chunk size. After scanning this many bytes, we fall back to the global table to lookup forwarding information.

Symmetrically, with writing as with reading, after reaching the end of the span or burning max\_scan bytes, without successfully forwarding, we populate an entry in the table. Because the bound must hold from any starting location, after successfully forwarding an object, we begin counting again, up to max\_scan. If we hit an indirection, it is itself a forwardable object. In this case, we also mark the object downstream from it as start of a new span (in addition to resetting the max\_scan counter).

**Skipping-over:** A worst-case scenario is, for instance, a tree value with shape only, containing no scalars, but whose subtree was already copied and burned. This value consists exclusively of small, non-forwardable objects:

\[
\begin{array}{c}
N \\
N \\
B \\
\ldots \\
B \\
L
\end{array}
\]

Scanning to the end of this interval will fail to turn up a forwarding pointer. Fortunately, the size of the problematic interval is bounded by the chunk size of the starting region in the young generation. The reason is that, otherwise, the interval would contain a chunk redirection pointer to another chunk, which itself is forwardable. Unfortunately, the landlocked, evacuated value needs to not only have its new forwarding address resolved (for writing an indirection in the to-space), but the collector also needs to know where the burned value ends in the from space, so that it can subsequently continue collection.

To this end, our algorithm introduces a second table for storing skip-over addresses. The table provides a fallback and a slow path for evacuation, just like the table for forwarding pointers. However, our collector creates a table entry only when it enters a region for the first time at a non-zero location\(^6\). The reason is that values which begin at location zero in a region never need to be skipped over in this way. Either they are top-level values, or if they are referred to, it is via an indirection, which itself is trivial to skip over. Furthermore, such location-zero values are always forwardable, because we leave enough room in the footer of each region chunk to store a forwarding pointer. Conversely, when the collector stumbles on a transition to burned data in the middle of a chunk, it switches to the slow-path, performing a table lookup.

To support skipping ahead, the table stores only the end of the entire burned interval, i.e. the end of the value rooted in the first burned byte. This information is sufficient because the collector can only jump into the middle of the burned interval by following an indirection there. In that case, forwarding information is needed but skip-over information is not, because it is trivial to skip-over the value by skipping over the indirection itself. Adjacent burned intervals are not ambiguous with a single interval, because they correspond to two logical values, and as such are copied by separate evacuations resulting in separate entries in the table. In Figure 5b, our collector would create an entry in the skip-over table for the object at byte-address 109, since this object lives at a non-zero location and is burned by following an indirection pointer. The collector will look up this entry in the table if it evacuates the object starting at byte-address 100.

In conclusion, this subsection introduced a sharing preserving strategy, which, for completeness, includes side-metadata tables for slow-path lookups of forwarding and skip-over information. However, we expect that programs take these slow paths rarely, because: (1) heaps generally have a sufficient density of primitive data (ints, floats, strings) or indirections such that there is a high density of forwardable objects, and (2) skipping over already-evacuated values is necessary only when the nursery contains sharing. The latter case generally happens in programs that perform fine-grained allocations of small regions, with a high percentage of objects occupying location zero of their respective regions.

\(^6\)In future work, static analysis may assist in ruling out sharing and lessening this obligation.
3.5 Write Barrier
As explained in Section 3.3, because we allow old-to-young pointers, we need a write barrier on indirections written in the old generation. On every write of an indirection, we test the target address to see if it’s in the nursery, and if not we add it to a remembered set. Our write barrier currently compares the pointer against a \((\text{start}, \text{end})\) range of addresses for the nursery. Further optimizations to the “is in nursery?” predicate are possible in the future\(^7\).

One important optimization we perform is to prevent redundant chains of indirections, \textit{short circuiting} them, in the write-barrier. We perform an additional load to peek at the tag of the target to which the new indirection points, and if it is an indirection, we keep following it until we find a non-indirection tag as the target. As with other aspects of the design, this leverages the immutability of our heap to maintain the invariant.

4 Implementation Details
We implement our AMM system in the open-source Gibbon with a tag corresponding to a regular data constructor, (as opposed to a reserved tags described earlier) it retrieves the necessary layout information from the info-table.

**Pointer encoding:** At various points during collection, the collector needs to know metadata information of a region which houses an object that is the target of a pointer (indirection pointer or end-of-chunk pointer). For instance, if writing an old-to-old indirection pointer, the target region’s reference count needs to change; if promoting a chunk that ends with a link to a pretenured chunk, the target region’s set of chunks need to be updated. These metadata can be accessed via the footer of the target chunk. To get to the footer, we use a 64-bit pointer to store both: the address of the target object and the offset from there to the target chunk’s footer. The offset is stored in the 16 high-order bits. As a result, the maximum chunk size in our system is bound at 65K bytes \(2^{16}\).

**Optimizing default region size:** A performance anti-pattern with previous versions of Gibbon was to allocate a sizeable region of the default size, typically at least 1K, and then write only a single constant-sized object to it, such as one cell of a linked list. This wastes a lot of memory, and in GC-Gibbon, can also cause many more collections to occur. It is therefore profitable to identify certain regions with \textit{statically bounded maximum size}. We add such a static analysis on Gibbon’s LoCal intermediate language. When the compiler backend generates code for region allocation, it overrides the default size with the static bound if it is smaller. Implementing this requires analyzing all the locations that allocate to a particular region, and then inferring the sizes of objects written to these locations. The size of primitive types such as ints and floats is known a priori. Expressions that allocate a variably-sized serialized value (for example, using recursion) are inferred to have an \textit{unbounded} size.

5 Evaluation
In this section we evaluate our memory management system using a variety of benchmarks taken from previous literature, and two additional benchmarks—\textit{reverse} and \textit{treeupdate}—that stress the worse-case scenarios for our implementation. Besides prior Gibbon (referred to as Legacy-Gibbon in the rest of this section), we compare the performance of our implementation to GHC\(^10\), which is especially optimized to run functional programs which allocate lots of small objects, and Java, which has a highly optimized and mature garbage collector. For our experiments, we use a single-socket Intel E5-2699 18 core machine with 64GB of memory and running Ubuntu 18.04. We compile the C programs generated by our implementation using GCC 7.5.0 with all optimizations enabled (option \(-O3\)). For comparing against Legacy-Gibbon, we use its version 0.2 compiled from source. To ensure an

---

\(^7\)For example, we could allocate the nursery in a fixed portion of the virtual address space, so that a test on the pointer is sufficient for the is-in-nursery test, without any additional loads for (dynamic) nursery bounds.

\(^8\)https://github.com/iu-parfunc/gibbon/

\(^9\)Since we use GCC to compile the generated C programs because it usually produces more efficient code than Clang in our experience, especially for the switch-heavy tree traversal programs. Besides, getting meaningful link-time optimizations between code compiled using Clang and Rust is not trivial.

\(^10\)https://www.haskell.org/ghc/
with the same inputs. For GHC and Legacy-Gibbon, we hold which generally offers the same or better performance, but when computing the mean. We observed low variance in all (We give brief descriptions of all benchmark programs in Appendix C.1.) For GHC, we use strict datatypes in benchmarks, avoids performance complications due to laziness. All programs use the same algorithms\(^{11}\) and datatypes, and are run with the same inputs. For GHC and Legacy-Gibbon, we hold the size of the young generation constant at 4MB. For Java,

### Table 1. Run times of out-of-order and small-allocation benchmarks (in seconds and relative to GC-Gibbon).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>GC-Gibbon</th>
<th>Legacy-Gibbon</th>
<th>GHC</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>reverse</td>
<td>0.49</td>
<td>1.46</td>
<td>2.98</td>
<td>0.42</td>
</tr>
<tr>
<td>treeUpdate</td>
<td>0.77</td>
<td>4.17</td>
<td>5.41</td>
<td>0.37</td>
</tr>
<tr>
<td>coins</td>
<td>4.34</td>
<td>35.5</td>
<td>8.18</td>
<td>1.21</td>
</tr>
<tr>
<td>lcss</td>
<td>0.51</td>
<td>0.30</td>
<td>0.59</td>
<td>0.45</td>
</tr>
<tr>
<td>power</td>
<td>1.40</td>
<td>8.07</td>
<td>5.76</td>
<td>0.28</td>
</tr>
<tr>
<td>geomean</td>
<td>-</td>
<td>-</td>
<td>3.39×</td>
<td>0.46×</td>
</tr>
</tbody>
</table>

### Table 2. Run times of in-order allocation and bulk-traversal benchmarks (in seconds and relative to GC-Gibbon).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>GC-Gibbon</th>
<th>Legacy-Gibbon</th>
<th>GHC</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>buildKdTree</td>
<td>2.67</td>
<td>2.53</td>
<td>0.95</td>
<td>7.78</td>
</tr>
<tr>
<td>countCorr</td>
<td>1.77</td>
<td>1.77</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>allNearest</td>
<td>0.71</td>
<td>0.80</td>
<td>1.13</td>
<td>1.46</td>
</tr>
<tr>
<td>barnesHut</td>
<td>3.54</td>
<td>3.40</td>
<td>0.96</td>
<td>5.83</td>
</tr>
<tr>
<td>constFold</td>
<td>1.38</td>
<td>1.50</td>
<td>1.09</td>
<td>4.12</td>
</tr>
<tr>
<td>geomean</td>
<td>-</td>
<td>-</td>
<td>1.02×</td>
<td>2.19×</td>
</tr>
</tbody>
</table>

### 5.1 Evaluating Run Time and Memory Consumption

#### Run Time Performance.**

Tables 1 and 2 show the run time results of GC-Gibbon to Legacy-Gibbon, GHC, and Java on simple tree traversals from either of two classes: out-of-order or in-order.

For small out-of-order allocation benchmarks (Table 1), GC-Gibbon benefits from its fast bump-allocated young generation, whereas Legacy-Gibbon shows the overheads of malloc-based region allocations. In the case of reverse, both Gibbon versions need to allocate a new region per input element, and thus 8M regions are allocated in this instance. But despite this very high rate region allocation, GC-Gibbon is 7% faster than Java and only 14% slower than GHC. The lcss benchmark is surprisingly fast with Legacy-Gibbon. According to our initial observations, lcss’s allocation pattern seems to naturally have a stack-like behavior and benefits from Legacy-Gibbon’s region based memory management.

For in-order allocation and bulk-traversal benchmarks (Table 2), Legacy-Gibbon has a home-turf advantage. Importantly, this advantage is not harmed under GC-Gibbon. Key to this resilience is the pretenuring optimization described in Section 3.3—for these benchmarks, only the first chunk is allocated in the young generation and the rest are directly allocated in the old generation. Due to this, Legacy-Gibbon and GC-Gibbon also have similar memory usage for these benchmarks. The slowdowns observed here are primarily because GC-Gibbon’s pointer encoding mechanism, which puts an upper bound on the largest chunk that it can allocate, namely 65K, unlike Legacy-Gibbon which sets this upper bound to 1GB. However, both GC-Gibbon and Legacy-Gibbon outperform GHC and Java on these benchmarks.

For small allocations, our system is 3.79×, 0.46×, and 1.09× geomean faster than Legacy-Gibbon, GHC, and Java, respectively. For bulk-tree-traversals, our geomean speedup is...
Table 4. Run times in seconds of benchmarks run with different GC configurations (explained in Section 5.2).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Default</th>
<th>NoBurn</th>
<th>NoCompact</th>
<th>SimpleBarrier+SB</th>
<th>NoBurn+SB</th>
<th>NoCompact+SB</th>
</tr>
</thead>
<tbody>
<tr>
<td>reverse</td>
<td>0.49</td>
<td>0.43</td>
<td>11.8</td>
<td>0.49</td>
<td>0.44</td>
<td>11.9</td>
</tr>
<tr>
<td>treeUpdate</td>
<td>0.77</td>
<td>0.75</td>
<td>0.88</td>
<td>2.50</td>
<td>1.20</td>
<td>12.1</td>
</tr>
<tr>
<td>coins</td>
<td>4.34</td>
<td>4.32</td>
<td>4.31</td>
<td>10.45</td>
<td>10.3</td>
<td>10.4</td>
</tr>
<tr>
<td>lcss</td>
<td>0.51</td>
<td>0.53</td>
<td>0.54</td>
<td>0.52</td>
<td>0.52</td>
<td>0.53</td>
</tr>
<tr>
<td>power</td>
<td>1.40</td>
<td>1.34</td>
<td>1.36</td>
<td>1.38</td>
<td>1.40</td>
<td>1.44</td>
</tr>
</tbody>
</table>

1.02×, 2.19× and 1.5×. Overall, these results show that GC-Gibbon offers significant performance improvements compared to Legacy-Gibbon on small and out-of-order allocation benchmarks, without degrading the performance on bulk-traversal and allocation benchmarks. Legacy-Gibbon is extremely slow on certain workloads such as coins, power [17] or treeUpdate, thereby discounting its use entirely if any part of an application has allocation patterns like these.

Memory Consumption. Table 3 shows the memory allocation behavior for the benchmarks from the small-allocation group. GC-Gibbon has geometric 47% memory usage relative to Legacy-Gibbon. Aligning with the runtime results where lcss was GC-Gibbon’s weakest spot, the memory consumption of this benchmark is higher compared to Legacy-Gibbon.

Sensitivity to Size of Inputs. Figure 6 shows the run times of benchmarks using inputs of various sizes. The young-generation size is held constant at 4MB. All the variants have similar behavior, with Legacy-Gibbon being the slowest in most cases. The graphs for reverse and treeUpdate show how GC-Gibbon fixes especially poor asymptotics of Legacy-Gibbon.

For memory consumption, we observe close to constant improvement (as measured by the R column in Table 3) when varying the input size.

Other parameter sweep results are posted in Appendix C.2.

5.2 Evaluating Our Design Choices

To evaluate the effects of the design choices we made, we run the benchmarks that stress the collector in six different modes, each of which toggles a specific choice. All of these modes (except SimpleBarrier) are configuration flags provided to the collector. SimpleBarrier requires recompiling the mutator since the write-barrier is inlined into the mutator at compile time.

- Default: follow the design described in Section 3, with all optimizations enabled.
- NoBurn: disable the forwarding pointer mechanism (Section 3.4), and allow shared values to multiply. A benefit is that writing forwarding pointers, burning data, and maintaining side-metadata tables is not needed.

- NoCompact: disable compaction (pointer elimination). For each indirection pointer encountered during evacuation, if the target object is not already copied, it is put into a fresh region and a new indirection pointing to this fresh region is created.

- SimpleBarrier: disable elimination of redundant chains of indirections (Section 3.5). This makes the write-barrier more efficient by reducing the number of memory loads it performs, but makes the collection more expensive because of the overheads associated with evacuating indirections (due to forwarding, side-metadata tables, etc.).

- NoBurn+SB: combine NoBurn and SimpleBarrier.

- NoCompact+SB: combine NoCompact and SimpleBarrier.

The results are given in Table 4. At first glance, NoBurn mode looks like the best choice. It is fast because it touchess less memory in the young generation, as it does not have to write forwarding pointers or maintain side-metadata tables. But, since sharing is disabled, it is very easy to run into a pathological-worst-case that could cause the collector to copy an exponential amount of data, leading to inefficient space usage. None of the benchmarks we considered here trigger this behavior, however. With respect to NoCompact mode, reverse is 24× slower in this configuration since it has the highest number of indirections among these benchmarks—8M, one for each cons-cell. The effectiveness of the indirection-chain-eliminating write-barrier is demonstrated by treeUpdate and coins, both of which create long indirection chains and are 2-3× slower in this mode.

6 Related Work

The most closely related work to this paper is, of course, Vollmer et al.’s Gibbon compiler [29], which was summarized in Section 2. This paper presents a generational collector with copying collection for the young generation while keeping Gibbon’s reference-counted regions for the old generation.

Region-based Memory Management: The main motivation behind region-based memory management [2, 22, 25] was to bring some of the benefits of stack-based memory management (a technique common in imperative languages like Pascal and Algol) to higher-order functional languages, primarily Standard ML. In this context, “region types” are a feature of a type system that tracks what region of memory a value is allocated into, with the goal of safely deallocating all values in that region once it goes out of scope. This approach suffers from lack of prompt deallocation of memory (Section 1). Also, some common patterns of functional programming (e.g. tail recursive functions) can end up causing memory leaks [23].

Various attempts at extensions or optimizations to region systems have been proposed to address this, such as storage mode analysis, where a compiler inserts special instructions to reset the allocation pointer in a region if it can probe the region contains no live values, essentially allowing its
contents to be overwritten [24]. Elsman and Hallenberg [10] explored combining regions and copying collection in the MLKit system. While MLKit uses regions, each object within a region is traditional in every other way; the heap is not serialized like it is in Gibbon. Its collector does not face the challenges of copying serialized, partially-written objects, but also doesn’t benefit from the resulting compaction.

**Garbage Collection:** The literature on traditional garbage collection includes not only much work on tracing and reference counting independently, but also in combination. Our proposed collector is similar to Ulterior reference counting [3] which also has a copying young generation and a reference-counting old generation. This work was further extended in [20], which uses an efficient heap structure and reference counting, and also includes a backup tracing collector. More recent work [31] has shown the performance benefits of using an efficient heap structure and reference counting. The LXR collector brings together several optimizations and heuristics, and introduces an efficient remembered set and a low-overhead write barrier to make reference counting efficient, and is able to reclaim most memory without any copying. There is ample opportunity to improve our reference counting collector using these techniques.

**Compressed heap data:** Memory compression is available at the page level as an operating system feature, but it has also been explored in the language runtime, for example in Java for embedded devices [6]. More recent work on computing with serialized data [27], can be viewed as a form of compression.

7 Conclusions and Future Work

We presented a new approach to memory management for mostly serialized heaps, as found in the Gibbon compiler and its runtime system. This hybrid collector is able allocate objects and regions quickly and coalesce objects, which were scattered at the points of their allocation, into efficient, serialized representations.

This work is the first step in a new direction that invites further study and refinement. It is common in computer science to trade-off time and space using compression techniques, and these mostly serialized heaps point to opportunities to explore these tradeoffs more deeply in the context of language’s in-memory representations. More prosaic, there are additional optimizations to develop and apply to our system to further close the gap with traditional implementation techniques on their “home turf”, (i.e. the worst-case scenarios for Gibbon’s native representations). We also plan to extend the reference counting strategy in the old generation with an additional tracing collector. Finally, a major topic of future work is to scale the approach to the parallel setting, both for the mutator and the collector.

Acknowledgments

We would like to thank the anonymous reviewers for their suggestions and comments. This work was supported in part by National Science Foundation awards CCF-1908504, CCF-1919197, CCF-2216978, CCF-1909862, CCF-2119352, as well as by the Engineering and Physical Sciences Research Council award EP/X021173/1.

References


A Appendix: Sample Gibbon programs

In this section we present the details of some Gibbon programs, their translated LoCal IR code, and finally the C code that is generated by Gibbon.

A.1 mkList: constructing a linked list

```haskell
data List = Cons Int List | Nil
mkList :: Int → List
mkList 0 = Nil
mkList n = Cons n (mkList (n-1))
```

(a) `mkList`'s starting Haskell source code.

```localed
mkList :: ∀l r n. Int → List
mkList n = if n == 0 then Nil else
  let loc = l + 9 in
  let tail: List = mkList [l] (n-1) in
  Cons l n tail
```

(b) `mkList` compiled into LoCal IR by Gibbon

Figure 7

```c
typedef struct GibCursorProd3_struct {
  GibCursor field0; GibCursor field1; GibCursor field2;
} GibCursorProd3;

GibCursorProd3 mkList(GibCursor footer_r_289, GibCursor loc_288, GibInt n_30_97_141) {
  if ((loc_288 + 18) > footer_r_289) {
    gib_grow_region(&loc_288, &footer_r_289);
  }
  if (n_30_97_141 == 0) {
    *(GibPackedTag *) loc_288 = 1; // Nil
    GibCursor after_tag_573 = loc_288 + 1;
    return (GibCursorProd3) {footer_r_289, loc_288, after_tag_573};
  } else {
    GibCursor loc_335 = loc_288 + 9;
    *(GibPackedTag *) loc_288 = 0; // Cons
    GibCursor after_tag_582 = loc_288 + 1;
    *(GibInt *) after_tag_582 = n_30_97_141; // Element in the cons cell
    GibCursorProd3 tmp_struct_5 = mkList(footer_r_289, loc_335, (n_30_97_141 - 1));
    return (GibCursorProd3) {tmp_struct_5.field0, tmp_struct_5.field1, tmp_struct_5.field2};
  }
}
```

Figure 8. `mkList` translated to C by Gibbon.

This code operates as follows. The arguments of `mkList` include the number \( n \) and a symbolic location \( l' \), where the resulting list of length \( n \) will be allocated. This allocation technique is a form of destination-passing style. In LoCal, a location \( l' \) is an address within in a region \( r \) and type \( \tau@l' \) is assigned to a value of type \( \tau \) residing at a location \( l' \). If \( n \) is zero, `mkList` simply allocates `Nil` at location \( l' \). Otherwise, it starts constructing a cons cell. First it binds a location \( l'_1 \) that is 9 bytes past the location \( l' \) (one byte for the `Cons` tag and 8 bytes for the integer). Next, it recursively constructs a list of length \( (n-1) \) at location \( l'_1 \). Then it writes the `Cons` tag and the integer \( n \) starting at location \( l'_1 \), which completes the construction of this cons cell. Gibbon’s location inference algorithm places the cons cell and its tail within the same region because the allocations in `mkList` happen in order. Thus, the resulting list looks like the structure shown in Figure 2.
A.2 reverse: reversing a linked list

\[
\text{reverse :: } \text{List} \rightarrow \text{List} \\
\text{reverse } \text{Nil} \text{ acc } = \text{acc} \\
\text{reverse } (x:xs) \text{ acc } = \text{reverse } xs (x:acc)
\]

(a) Starting Haskell source code for accumulator-style list-reverse.

\[
\text{reverse} : \forall \; l' \; m' \; n'. \; \text{List}@l' \rightarrow \text{List}@m' \rightarrow \text{List}@n'
\]

\[
\text{reverse } \text{xs} \text{ acc } = \text{case } \text{xs} \text{ of} \\
\text{Nil } \rightarrow \text{Ind } n' \text{ (addrOf acc)} \\
\text{Cons } (y:\text{Int}@I_\gamma) \text{ (ys:List}@I_\gamma') \rightarrow \\
\text{let } \text{region } u \text{ in} \\
\text{let } \text{loc } o_1^u = \text{start } u \text{ in} \\
\text{let } \text{loc } o_2^u = o_1^u + 9 \text{ in} \\
\text{let } \text{cpy: List}@o_2^u = \text{Ind } o_2^u \text{ (addrOf acc)} \\
\text{let } \text{acc': List}@o_1^u = \text{Cons } o_1^u y \text{ cpy } \text{ in} \\
\text{reverse } [I_\gamma'; o_1^u n'] \text{ ys acc'}
\]

(b) reverse compiled into LoCal IR by Gibbon.

typedef struct GibCursorProd6_struct { GibCursor field0; GibCursor field1; GibCursor field2; ... } GibCursorProd6;

GibCursorProd6 reverse(GibCursor end_r_293, GibCursor end_r_294, GibCursor end_r_295, GibCursor loc_292, GibCursor xs_31_98_145, GibCursor acc_32_99_146)
{
  GibShadowstack *rstack = READ_SHADOWSTACK;
  GibShadowstack *wstack = WRITE_SHADOWSTACK;
  GibShadowstackFrame *frame;
  if ((loc_292 + 18) > end_r_295) {
    gib_grow_region(&loc_292, &end_r_295);
  }
  GibPackedTag tmpval_931 = *(GibPackedTag *) xs_31_98_145;
  GibCursor tmpcur_932 = xs_31_98_145 + 1;
  switch (tmpval_931) {
    case 0: {
      GibInt tmpval_937 = *(GibInt *) tmpcur_932;
      GibCursor tmpcur_938 = tmpcur_932 + sizeof(GibInt);
      gib_shadowstack.push(rstack, acc_32_99_146, end_r_294, Stk, PList_T);
      gib_shadowstack.push(rstack, tmpcur_938, end_r_293, Stk, PList_T);
      gib_shadowstack.push(wstack, loc_292, end_r_295, Stk, PList_T);
      GibChunk region_939 = gib_alloc_region(32);
      GibCursor r_356 = region_939.start;
      GibCursor loc_345 = r_356 + 9;
      *(GibPackedTag *) r_356 = 0;
      GibCursor after_tag_600 = r_356 + 1;
      *(GibInt *) after_tag_600 = tmpval_937;
      gib_indirection_barrier(loc_345, end_r_356, acc_32_99_146, end_r_294, PList_T);
      return reverse(end_r_293, end_r_356, end_r_295, loc_292, tmpcur_938, r_356);
    }
  }
case 1: {
    GibCursor jump_412 = xs.31.98.145 + 1;
    gib_indirection_barrier(loc_292, end_r.294, acc.32.99.146, end_r.294, PList_T);
    GibCursor end_592 = loc_292 + 9;
    return (GibCursorProd6) {end_r.293, end_r.294, end_r.295, jump_412, loc_292, end_592};
    break;
}
case GIB_INDIRECTION_TAG: {
    uintptr_t tagged_tmpcur_17 = *(uintptr_t *) tmpcur_932;
    GibCursor tmpcur_956 = GIB_UNTAG(tagged_tmpcur_17);
    uint16_t tmptag_958 = GIB_GET_TAG(tagged_tmpcur_17);
    GibCursor end_from_tagged_indr_447 = tmpcur_956 + tmptag_958;
    GibCursor jump_449 = tmpcur_932 + 8;
    GibCursorProd6 tmp_struct_16 = reverse(end_from_tagged_indr_447, end_r.294, end_r.295, loc_292, tmpcur_956, acc.32.99.146);
    return (GibCursorProd6) {end_r.293, tmp_struct_16.field1, tmp_struct_16.field2, jump_449, tmp_struct_16.field4, tmp_struct_16.field5};
}
case GIB_REDIRECTION_TAG: {
    uintptr_t tagged_tmpcur_19 = *(uintptr_t *) tmpcur_932;
    GibCursor tmpcur_971 = GIB_UNTAG(tagged_tmpcur_19);
    uint16_t tmptag_973 = GIB_GET_TAG(tagged_tmpcur_19);
    GibCursor end_from_tagged_indr_447 = tmpcur_971 + tmptag_973;
    return reverse(end_from_tagged_indr_447, end_r.294, end_r.295, loc_292, tmpcur_971, acc.32.99.146);
}
default: {
    printf("%s\n", "Unknown tag in: tmpval_931");
    exit(1);
}
}
A.3 treeUpdate

\[
\text{loop} :: \forall \forall \forall l r k s. \text{RNG} \to \text{Int} \to \text{Tree} @ l r \to \text{Tree} @ k s \\
\text{insert} :: \forall \forall l r k s. \text{Int} \to \text{Tree} @ l r \to \text{Tree} @ k s
\]

\[
\text{loop} [l r k s] \text{rng i tr} = \\
\text{if } i == 0 \text{ then } \text{Ind } k s (\text{addrOf } tr) \\
\text{else let } (n, \text{rng'}) = \text{next } \text{rng } 0 512 \text{ in} \\
\text{let } \text{region } t \text{ in} \\
\text{let } j t = \text{start } t \text{ in} \\
\text{let } \text{tr'} : \text{Tree} @ j t = \\
\text{if } n \% 2 == 0 \text{ then insert } [l' j' r k s] \text{ n tr} \\
\text{else delete } [l' j' r k s] \text{ (n-1) tr} \\
\text{in loop } \text{rng'} (i-1) \text{ tr'}
\]

\[
\text{insert} :: \forall \forall l r k s. \text{Int} \to \text{Tree} @ l r \to \text{Tree} @ k s
\]

\[
\text{insert} [l' j' r k s] \text{ n tr} = \text{case tr of} \\
\text{Null } \rightarrow \text{Leaf } k s n \\
\text{Node } (m: \text{Int} @ l m r) (x: \text{Tree} @ l x r) (y: \text{Tree} @ l y r) \rightarrow \\
\text{if } m < n \text{ then} \\
\text{let } k2 s = k1 s + 1 \text{ in} \\
\text{let } x' : \text{Tree} @ k2 s = \\
\text{let } k3 s = \text{after } \text{Tree} @ k2 s \text{ in} \\
\text{let } y' : \text{Tree} @ k3 s = \\
\text{insert } [l' j' r k3 s] \text{ n y in} \\
\text{Node } k1 s m x' y' \text{ else } ... -- \text{Insert into the left subtree.}
\]

\[
\text{Leaf } (m: \text{Int} @ l m r) \rightarrow ... 
\]

Figure 11. A LoCal program that repeatedly inserts a random number into a binary search tree.

B Appendix: Implementation Details

B.1 Reordering Tag Allocations

Even though allocations in LoCal happen in order, its formalism requires that a data constructor tag be written after all its fields are. However, this creates a problem for our collector, which might need to copy a value while it is still under-construction (Section 3.3). Without the tag present at the beginning of an object, the collector cannot infer what kind of an object it is copying. We bypass this by reordering the writes such that a data constructor tag always gets written before any of its fields. Not only does this help the collector, it also makes the mutator slightly more efficient.

B.2 Random-access via shortcut pointers

While indirection pointers enable allocation of values out-of-order, they do not enable reading values out-of-order. To this end, Gibbon uses untagged shortcut pointers to enable constant-time random-access to certain fields on a per-data-constructor, per-field basis. The reason shortcut pointers are necessary is that some programs need to “skip over” certain parts of a value to read it out-of-order, and there is no way to accomplish this if the value is fully serialized—the only way to access a particular part in it is to scan past all of the data that has been serialized before it. For example, to compile a program which fetches the rightmost leaf of a binary tree with the correct asymptotic complexity ($O(\log n)$), Gibbon stores the absolute address of the right subtree in each intermediate node so that it can be accessed directly without traversing the left subtree, which would make this a $O(n)$ operation. Figure 4 shows such a node.

Fortunately, shortcut pointers are innocuous with respect to garbage collection because they always point within the same region, and thus cannot change the lifetime of other regions. In principle, they require less space to store than in a normal pointer-based representation, because no pointer is needed for the leftmost non-scalar field, e.g. one pointer for a binary tree, instead of two. A possible implementation choice would be to store the integer size in bytes of packed fields which can be used to skip over them. For example, the address of the right subtree can be computed as (the address of the left subtree + size of the left subtree), given that all the fields are serialized side-by-side in a single region. However, the runtime representation of regions (chunked, growable, as described in the next section) makes this choice slightly less efficient in practice.

B.3 Info-table representation

Figure 12 gives the representation of the info-table used in GC-Gibbon’s runtime system.

C Appendix: Evaluation

C.1 Benchmark descriptions

- reverse: This is the standard accumulator style list-reverse program shown in Figure 3; it reverses a list containing 8M integers. The Java implementation is defined using a while loop rather than a recursive function.
type InfoTable = Vec<DatatypeInfo>;

enum DatatypeInfo {
    Scalar(usize),
    Packed(Vec<DataconInfo>),
}

struct DataconInfo {
    // Bytes before the first packed field.
    scalar_bytes: usize,
    // Number of shortcut pointer fields.
    num_shortcut: usize,
    // Field types of packed fields.
    field_tys: Vec<u32>,
}

Figure 12. The representation of GC-Gibbon’s info-table used in runtime system.

- **treeUpdate**: This is the complete version of the program given in Figure 11 in the Appendix. It starts with a very small search tree and repeatedly inserts and deletes numbers in it. The numbers are chosen from small range (0-512) to keep the size of the tree more or less constant, and the tree is updated 5M times. The Java implementation encodes the outer “update” loop as an actual while loop, but insert and delete are defined using recursion in the standard way.

- **coins**: This benchmark is taken from GHC’s NoFib benchmark suite. It computes the number of ways in which a certain amount of money can be paid by using the given set of coins. The input set of coins and their quantities are [(250, 55), (100, 88), (25, 88), (10, 99), (5, 122), (1, 177)], and the amount to be paid is 999.

- **lcss**: This benchmark computes the longest-common-substring using Hirschberg’s algorithm. Our implementation is taken from GHC’s NoFib benchmark suite. We provide as input two strings of length 3100 and 3000 respectively, such that the result has length 2100.

- **power**: This benchmark computes 20 elements of the power series \((ts = 1 :+\: ts^2)\), which is shown here assuming lazy evaluation. We use a slightly modified implementation that is suitable for a strict language.

- **buildKdTree** and **countCorr** and **allNearest**: buildKdTree constructs a kd-tree containing 1M 3-d points in the Plummer distribution. countCorr takes as input a kd-tree and counts the number of correlated (within a distance of 100 units) points for all 1M 3-d points. allNearest computes the nearest neighbor of all 1M 3-d points.

- **barnesHut**: Uses a quad tree to run an nbody simulation over 1M 2-d point-masses distributed uniformly within a square.

- **constFold**: This benchmark is taken from and implements constant folding for a language that supports integer arithmetic. It is run on synthetic syntax-tree which is a balanced binary tree of depth 26.

C.2 Parameter sweeps

In this section we discuss the results of three parameter sweeps for small out-of-order allocation benchmarks that stress the collector. Figure 13 shows the run times of benchmarks using different young-generation sizes. This shows the expected tradeoff between space and time; the wall-clock time gets better as the young generation gets bigger, due to fewer collections. Figure 14 shows the run times of benchmarks using different initial chunk sizes for GC-Gibbon. Using bigger initial chunks, and therefore growing the overall region at a faster rate, causes more collections to occur since the young generation fills up faster. It also leads to inefficient space usage since many of the larger chunks will be mostly empty.

Figure 13. Run times in seconds of benchmarks using young generations of various sizes. The red line marks the young-generation size used for measurements reported in Table 1 (4MB).

12https://gitlab.haskell.org/ghc/nofib
Figure 14. Run times in seconds of benchmarks using initial chunks of varying sizes. Our region size inference analysis (Section 4) causes GC-Gibbon to allocate constant-sized regions in the main workload of reverse and coins, thus their run times are constant. We report the measurements after disabling this optimization as "GC-Gibbon (region size not inferred)".
D Appendix: Evacuation Algorithm

```haskell
1  -- store type layouts per data constructor (static)
2  global info_table[]
3  -- map spans of from-space memory to to-space
4  global fwd_table[]
5  -- map address of value start to its end
6  global skip_table[]

7  fun evacuate_root(from_start, ty):
8      if in_oldgen?(from):
9          -- skip root
10     else if already_marked?(from):
11         update_root(fwd_addr(from_start))
12     else:
13         (to_start, to_end) := alloc.oldgen_region()
14         evacuate_object(from_start, ty, to_start)
15         update_root(to_start, to_end)
16
17     -- Returns a pointer into the to-space,
18     -- after the given value
19     fun skip_over(from):
20          -- We can only create a need for skipping if we process
21          -- at least one data constructor in an evacuate_value:
22          assert(not(zero_location?(from)))
23          return skip_table[from]
24
25     -- Returns a pointer in the to-space
26     fun fwd_addr(from):
27          -- For the first location in the region,
28          -- the region metadata lets us forward:
29          if zero_location?(from):
30              return footer_fwd_ptr(region(from))
31          else:
32              while offset < max.scan:
33                  let next = deref(from + offset)
34                  match(next):
35                      Forwarded(addr): return (addr - offset)
36                      Burned: offset += 1
37                      -- The interval-map source to target bytes
38                      -- by internally mapping entire
39                      -- (src..src+k) => (dst..dst+k) ranges efficiently:
40                      -- side metadata lookup = slow path
41              return fwd_table[from]

(a) Part of the evacuation algorithm. Global definitions, helpers, and
entrypoint to begin evacuation. See also figure 15b.
```

```haskell
1  -- Recursively evacuate the value at a given location,
2  -- with the given type. Because of the acyclic heap,
3  -- this will never recur back to the same location.
4  fun evacuate.object(from_start, ty, to_start):
5      let ty_stk = [ty]
6      let to = to_start
7      let from = from_start
8      let span.start = from_start
9      let span.bytes = 0
10
11     fun end.span():
12          if span.bytes > max.span:
13              let sz = from - span.start
14              fwd_table[span.start .. sz] := to-sz .. to
15          span.bytes := 0
16          span.start = from
17
18     -- One evacuate-and-burn session
19     -- on one contiguous interval:
20     while (from_ty = pop(ty_stk)):
21         while (chunk_redirection?(from)):
22             forward_obj(from, to)
23             from := deref(from)
24         end_span()
25         if indirection?(from):
26             -- start a separate interval in a new chunk:
27             to' = evacuate.object(deref(from), from_ty, to)
28             forward_obj(to, to)
29             from += TAGGED_INDIRECTION_SIZE
30             to := to'
31         end_span()
32     else if already.marked?(from):
33         to := write_indirection(to, fwd_addr(from))
34         if non_empty?(ty_stk):
35             from := skip_over(from)
36         else:
37             -- regular data-copying codepath
38             -- advance to/from cursors past
39             -- the data written:
40             (to',from') =
41             write.tag_and.scalars(to, from)
42             -- look up types of 0 or more
43             -- non-scalar data fields (children):
44             push.children(stk, info_table[from])
45             fwded = burn_or_forward_obj(from)
46             to := to'
47             from := from'
48             if fwded: end.span()
49             else: span.bytes += from' - from
50         if not(zero.location?(from.start)):
51             -- Populate for future slowpath lookups
52             -- one byte after the end
53             skip_table[from.start] := to
54     return to

(b) Continued from figure 15a, core burning and forwarding algorithm
for sharing maintenance.
```

Figure 15. Evacuation algorithm