Being *Lazy* When it *Counts*

Practical Constant-Time Memory Management for Functional Programming

Chun Kit Lam[0000−0002−8856−9095] and Lionel Parreaux[0000−0002−8805−0728]

HKUST {cklamaq,parreaux}@ust.hk

Abstract. Functional programming (FP) lets users focus on the business logic of their applications by providing them with high-level and composable abstractions. However, both automatic memory management schemes traditionally used for FP, namely tracing garbage collection and reference counting, may introduce latencies in places that can be hard to predict, which limits the applicability of the FP paradigm.

We reevaluate the use of lazy reference counting in single-threaded functional programming with guaranteed constant-time memory management, meaning that allocation and deallocation take only a bounded and predictable amount of time. This approach does not leak memory as long as we use uniform allocation sizes. Uniform allocation sizes were previously considered impractical in the context of imperative programming, but we find it to be surprisingly suitable for FP.

Preliminary benchmark results suggest that our approach is practical, as its performance is on par with Koka's existing state-of-the-art implementation of reference counting for FP, sometimes even outperforming it. We also evaluate the effect of different allocation sizes on application performance and suggest ways of allowing large allocation in non-mission-critical parts of the program via Koka's effect system.

We believe this potentially opens the door to many new industrial applications of FP, such as its use in real-time embedded software. In fact, the development of a high-level domain-specific language for describing latency-critical quantum physics experiments was one of the original use cases that prompted us to initiate this work.

1 Introduction

Functional programming allows software developers to design applications at a very high level of abstraction. It lets them focus on immutability and function composition to design programs in a way that can often be described as "correct by construction".

For example, the *functional reactive programming* (FRP) pattern [10, 15, 22], as found in popular languages like Haskell and Elm, uses immutable data structures and higher-order functions to let users *declaratively* specify reactive designs, such as animations, graphical user interfaces, and video games.

In practice however, a major problem with FRP, as well as with a number of other functional design patterns, is that it is extremely *memory-intensive* — on

each event, the old immutable states of objects that change are discarded and reconstructed from scratch, and new closures are allocated to register actions to be performed when new events occur. This means that typical FRP programs continuously allocate and free lots of memory, which can translate to jerky animations and small pauses in graphical user interface rendering.

Some FRP implementations allow users to avoid needless recomputation by telling parent components whether the state has changed [7]. This enables the runtime to reuse computation results for unaffected parts of the GUI, which reduces memory allocator pressure and makes the program less memory-intensive. However, the effectiveness of this kind of optimization depends not only on the GUI layout design, but also programmer efforts and discipline. Beginner users may not understand the need for such optimization, or may not know for sure if the layout is truly unchanged.

Traditional Memory Management for Functional Programming Functional programming languages typically rely on some kind of garbage collection for automatic memory management. They often rely on generational, copying garbage collectors [9] to efficiently process short-lived objects while performing tracing garbage collection for older generations.

Reference counting, one of the oldest memory management techniques [5] [24], was historically considered less practical than garbage collectors in this context, because frequent reference count updates can have a large performance impact and because reference counting cannot handle cycles [14]. However, there has been a recent surge in the popularity of reference counting for general-purpose memory management. Shahriyar et al. [20] analyzed the overhead of reference counting compared to tracing garbage collectors and introduced several strategies for improving its performance. Ullrich and de Moura [21] showed that destructive updates enabled by precise reference counting can provide significant performance improvements for functional programming languages. Reinking et al. [19] introduced Perceus, an algorithm for precise reference counting with destructive updates and specialization, and implemented the algorithm on Koka, a functional programming language with a type and effect system. These papers showed that functional programming languages using reference counting can have good performance too and can compete with state-of-the-art garbage collectors.

On the other hand, latency is also a key metric for system performance, sometimes even more important than throughput, and it has not been directly addressed by these recent works. Indeed, *eager* forms of reference counting can often lead to garbage collection pauses, sometimes even longer than those of tracing garbage collectors [3]. For example, consider an application that represents a news feed as an infinitely-scrollable view on which various widgets can appear. Imagine that a particular drawing-board widget interacts with user input by letting the user draw shapes in it. Each of these shapes will be represented as some vector-graphics object in a collection of shapes that have been drawn so far, which could grow very large. Thus, when the user finally loses interest and scrolls past the widget, its deallocation in a normal reference counting implementation will require recursively deallocating an unbounded number of vector-graphic

objects, which is likely to cause a small pause in the scrolling animation, making it appear jerky.

Constant-Time Memory Management for Functional Programming In this paper, we revisit the old idea of *lazy* reference counting, which avoids the long delays introduced by cascading deallocations. Our technique works by deferring the deallocation of an object's fields until the memory is needed by further allocations, which can progressively reuse memory by traversing the graph of objects no longer in use. This effectively gives us *constant-time* allocation and deallocation procedures, which we refer to as *constant-time memory management*.

There are naturally a number of limitations and caveats to this approach. First, in order to be truly (non-amortized) constant-time without the possibility of leaking unbounded amounts of memory, we need all allocations to share the same size. While it means that the compiler must *split up* large objects into multiple allocations, we argue that this approach is eminently practical in the context of functional programming languages. Indeed, algebraic data type objects tend to be on the smaller side, and we show that the performance hit associated with splitting larger objects in this context varies from moderate to small or insignificant.

Second, like most reference counting approaches, we do not handle cycles in the reference graph. But we argue that in the context of functional programming, where most data is immutable and tree-like, cycles can normally be avoided. Depending on the compilation strategy and language features, cycles may still be introduced in the presence of laziness, recursive closures, or OCaml-style cyclic values. With a slight loss of expressiveness, it is possible to design pure functional programming languages where cycles cannot be constructed, as exemplified by Lean [21]. We argue that the loss of expressiveness is acceptable. On one hand, while it is true that recursive closures are traditionally implemented through cyclic values, other implementation strategies exist, used for example in languages like Rust and Koka. On the other hand, while some languages use laziness to allow conveniently constructing cyclic values, it is possible to design more restricted languages where laziness is still supported but where cyclic values cannot be constructed. Third, our approach is currently only applicable to sequential mutators. Due to the lazy nature of our approach, it is impossible in general to know the amount of memory actually in use by the program at any given time. This makes some approaches to concurrent memory management with thread-local free lists impractical.

In turn, our approach also has large advantages. While the uniform allocation size does create *internal* fragmentation (which could lead to up to 2x memory usage overhead in the worst case), it also means that we are effectively free from *external* fragmentation, whose worst-case overhead can become vastly higher than 2x, and which often cripples long-running systems in practice, leading to degraded performance and even system failure [17].

We formalized our design, nicknamed CTRC (for *constant-time reference counting*), and implemented it inside the Koka programming language, leveraging its existing reference counting optimizations as well as its type-and-effect system.

The allocator adopts the approach of Leijen et al. [13] with page-local free lists to improve memory locality.

Our experimental results show that this approach can achieve throughput comparable with Koka using the state-of-the-art mimalloc allocator, while providing constant-time guarantees for both allocation and deallocation, even for cases with larger objects that require splitting into several allocations. A major advantage of our approach is that it is extremely simple to implement: the basic runtime is about 160 lines of C code (the entirety of which fits into four pages of Appendix B), and does not make use of any existing system allocator beyond the operating system's memory page allocation routines.

This could enable the implementation of efficient automatic memory reclamation for resource-constrained embedded systems, which could have future applications in the domain of hard real-time systems.

Contributions

- Although similar solutions were proposed in the past in slightly different contexts back in 1963 [24], we present a refreshed and practical solution to *constant-time* memory management for functional programming, called CTRC (Section 2). CTRC was implemented by adapting the existing Koka runtime system and compiler. Our implementation is extremely simple, which we consider to be a major selling point.
- We formalize this new approach and show that it is sound and prevents memory leaks, in the sense that it does not increase the peak memory usage of programs (Section 2.4 and Appendix A). We also show that CTRC can be applied only locally, to those parts of a program that are latency-sensitive.
- We experimentally justify the practicality of CTRC, showing that on our benchmark programs, its space and time overheads are small when compared to Perceus, a state-of-the-art reference-counting implementation (Section 3). We also discuss the source of the overhead, and suggest approaches for programmers to reduce the overhead.

2 Presentation of Constant-Time Reference Counting

In this section, we present our basic idea of *constant-time reference counting* for Functional Programming (CTRC).

2.1 Eager and Lazy Reference Counting

Reference counting is typically implemented by attaching an integer reference count to each object, indicating the number of pointers pointing to the current object. When pointers or variables are modified, the reference count of the referred object is updated. As the reference count of an object becomes zero, it is deleted and the reference counts of its fields are decremented. For example, when deallocating a linked list, the reference count of the first node becomes

zero, which recursively decrements the subsequent nodes and deletes them. We shall refer to this type of reference counting as *eager* reference counting.

Eager reference counting allows immediate garbage collection with low pause times compared to tracing garbage collectors, as the tracing part is done incrementally by tracking reference counts rather than a separate tracing phase. However, reference counting can still have unbounded pause time as a deallocation event can trigger a chain of deallocation.

By contrast, CTRC performs reference counting *lazily*. Instead of updating reference counts for the fields of deallocated objects immediately, the updates are postponed by storing the deallocated objects in a free list. When there is a new allocation request, the fields of the object are freed as the object allocation is being reused. This breaks the chain of deallocation mentioned previously, making the allocation and deallocation of objects constant-time.

The use of constant-time reference counting mandates using only a single allocation size, as using multiple allocation sizes may result in memory leaks, which we discuss in Section 2.2. In general, large objects can always be split up into smaller segments to fit the single allocation size requirement. Although using a constant allocation size was traditionally considered impractical in previous literature, our experiments show that by selecting a suitable size, the space and performance penalty of splitting large objects is limited. While CTRC requires a constant allocation size, there can be cases in which the program requires large contiguous arrays, for example when interacting with graphics APIs. We show that one can make use of an effect system to isolate the parts of a program that need to perform variable-sized memory allocation, while maintaining the constant-time guarantee for all other parts of the program.

2.2 Allocation Size

We now discuss why it is necessary to have a single allocation size for constant-time memory management, overhead associated with this strategy, and criteria for choosing allocation size.

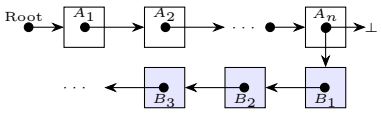


Fig. 1: Linked List with 2 Size Classes.

In the case of having multiple constant allocation sizes, one either has to give up constant time guarantee or potentially suffer from memory leaks [3]. Consider the example shown in Figure 1, where there are two size classes, with small and large objects denoted by white and pale blue boxes respectively. The last element of the small-object linked list contains a unique reference to a linked list with large objects. Assume that the program loses the unique reference to A_1 at some point, causing all the small objects A_i and large objects B_i to become unreachable. For normal reference counting schemes, the entire linked list together with the large-object linked list is deallocated. However, for constant

time memory management schemes, the collector can only visit and deallocate a bounded number of objects per step. Hence, if the small-object linked list is sufficiently long, the memory collector can only deallocate the small objects, which cannot be used to fulfill allocation for large objects. The allocator has to allocate additional memory to fulfill large object allocations, even though there are unused large object allocations that could be reused, causing a memory leak.

To avoid such an issue, our implementation only allows a single allocation size, and large objects are split into segments.[1] For imperative programming languages, this approach is considered infeasible as it is more common to have arrays and objects with a large number of fields. However, for functional programming languages, it is common to use data structures such as linked lists and other tree-shaped linked data structures, whose nodes are usually small. This makes our approach feasible, as shown by our experiments (Section 3).

There are multiple considerations when choosing the allocation size, including memory overhead and architecture-specific details. If the allocation size is a lot larger than the average object size, there will be severe internal fragmentation, which wastes memory and memory bandwidth, which can cause performance degradation. However, if the allocation size is too small, large objects are split into multiple cells, and access to certain fields involves multiple pointer indirections, making the access slow. In addition, each cell has to store metadata such as reference counts, so the overhead increases when large objects are split into multiple cells. One should also consider architecture-specific details, such as the cache line size and alignment requirements when choosing the allocation size. This is to make sure allocations satisfy alignment requirements when packed together, and require minimal cache access when accessing an object.

In our experiments, we choose 32 bytes as the size of each allocation, with a header occupying 8 bytes. The target CPU architecture includes x86-64 and aarch64, which uses 64-byte cache lines. Our objects should align to cache line boundaries to avoid false-sharing, so it is natural to choose 64 bytes as the allocation size. However, our early experiments found that the typical object sizes are small, so using an allocation size of 32 bytes can improve memory and cache utilization, while still being aligned to cache line boundaries.

Let the object size be n bytes, it requires at most $\lceil \frac{n}{16} \rceil$ segment, where every non-leaf segment contains a header and a pointer to the next segment. The worst-case memory usage is four times the optimal memory usage when metadata occupies 8 bytes. However, the actual worst-case memory usage is usually much smaller. For example, if the smallest allocations are 32-bit integers which occupy 4 bytes each, the object size is 12 bytes together with the metadata, and the memory overhead is 2.66 times the optimal memory usage. Note that this worst-case memory usage is independent of the allocation and deallocation pattern, which makes it simple to reason about statically with a tight upper bound.

[1] Note that since object sizes are a constant of the program, field accesses still take constant time, even for split objects.

2.3 Implementation in Koka

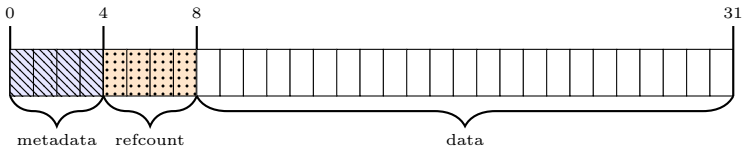


Fig. 2: Cell Data Layout.

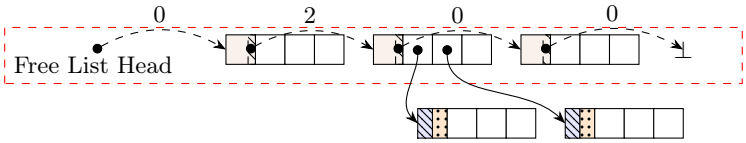


Fig. 3: Free List Layout.

We implemented the CTRC memory management scheme by extending Koka's C runtime and modifying the Koka compiler to split large objects into constant-size segments. In the following, we shall use *cell* to denote a fixed-size memory allocation (32 bytes, including a header of 8 bytes) and *drop* to denote decreasing the reference count of an object and *deallocating* it when the reference count is 0. Figure 2 shows a simplified view of the memory layout for each cell, which consists of a 32-bit metadata, 32-bit reference count `refcount`, and 24 bytes of actual storage which is capable of storing three 64-bit pointers or integers. The metadata encodes information including object type for pattern matching, the number of pointers in the current cell, and additional data for other extensions such as the "eagerly-deallocating-allocation" effect referred to in Section 2.5.

The free list is maintained as an intrusive linked list where the 8-byte header acts as the pointer to the next cell. As cells are aligned to 32 bytes boundary, the 5 least significant bits of the free list pointers can be used for storing some of the metadata, which includes information such as the segment's pointer count, that has been replaced by the intrusive pointer and can no longer reside in the object. Note that this does not apply to pointers pointing to live objects, i.e. the normal pointers, as the reference count must be stored in the object and 5 bits are not enough for other metadata. The linked list is maintained with a last-in-first-out (LIFO) order. As the last deallocated cell is likely to reside in the cache, reusing it first can increase the chance of getting a cache hit. Figure 3 shows an example free list layout. The cells inside the red dashed rectangle are inside the free list, note that their headers are different from those outside the free list. Pointers pointing to other free list cells contain metadata, such as the next cell pointer counts on top of the dashed arrows. Pointers pointing to live cells, denoted by solid arrows, are dropped when the free list cell is being reused.

In case the object occupies more than 24 bytes, the compiler splits the object into multiple segments. Each segment occupies a single cell, with pointers pointing to other segments in a tree-like fashion. Note that in our prototype implementation, the objects are split up in a linked-list-like fashion, i.e. each segment can only point to one other segment, for simplicity. The runtime treats each segment inside the free list as individual objects.

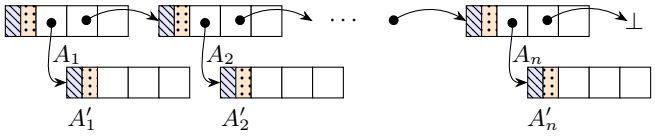


Fig. 4: Linked List Example.

Consider a linked list example, where each node contains a pointer to the next node and 3 64-bit integers. As each node is larger than 32 bytes, it is split into 2 segments. The example memory layout is shown in Figure 4. The i-th node is split into two segments A_i and A'_i, where A_i contains pointer to A'_i, pointer to the next node A_{i+1}, and 1 64-bit integer. A'_i contains the remaining 2 64-bit integers. When A_1 is deallocated, it is put into the free list. When there is another memory allocation and A_1 is removed from the free list, the allocator puts A'_1 into the free list as A_1 must contain the unique reference to A'_1, A_2 is dropped at this point but not necessarily deallocated as there can still be other references to A_2.

2.4 Soundness and Garbage-Free Guarantee

Due to space limitation, we provide the details of our formalization in Appendix A. We define the "*baseline semantics*" to be the typical operational semantics for untyped lambda calculus with explicit binding, pattern matching, and an interpretation of *dup* and *drop* as *no-op* instructions. The baseline semantics is the observable behavior of the program and is independent of the underlying memory management strategy. We refer to the semantics that manage the heap and perform eager reference counting as the "*eager semantics*". Similarly, we define the semantics of CTRC that perform lazy reference counting as the "*lazy semantics*". In addition to the heap, the lazy semantics also has the concept of a free list, which is a list of dropped cells that can be reused. Instead of getting a fresh memory location in the heap when performing allocation, the lazy semantics attempts to get a cell from the free list and drop its fields when the free list is non-empty. Also, instead of recursively dropping an object when the reference count becomes zero, the lazy semantics keeps the object in the heap and adds it to the free list.

Our formalization reasons about program traces in different semantics. We show that:

1. The eager semantics and lazy semantics both simulate the baseline semantics.

This makes sure the behavior is correct when allocation does not return memory that is still being referenced. Also, this provides the notion of time for other parts of the formalization.

2. At any point in the execution, the reference count of an object in the eager semantics x_{k} is smaller than that in the lazy semantics x_{c}, i.e. $x_{\mathrm{k}} \leq x_{\mathrm{c}}$.
   This makes sure we never put things that are still being referenced in the free list, so allocation does not return memory that is still being referenced, provided the eager semantics is sound.
3. When the points-to graph is acyclic and the free list is empty, we have $x_{\mathrm{k}} = x_{\mathrm{c}}$ for every object.
   This means that when we require additional memory from the system to fulfill allocation, the current heap is garbage-free, provided that the execution under the eager semantics is garbage-free at this point.

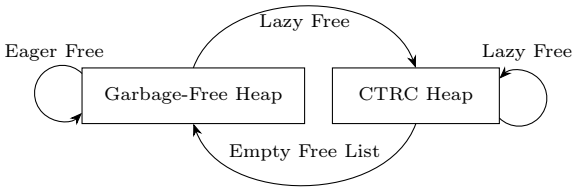


Fig. 5: Relationship between Eager free and Lazy free.

The proof is based on simulation, which means that we do not have to prove the soundness and garbage-free properties for program transformations under the lazy semantics. If the program transformation preserves soundness and garbage-free properties in the eager semantics, the same guarantees hold in the lazy semantics. Figure 5 shows the relationship between the eager and lazy semantics.

2.5 Eagerly-Deallocating-Allocation Effect

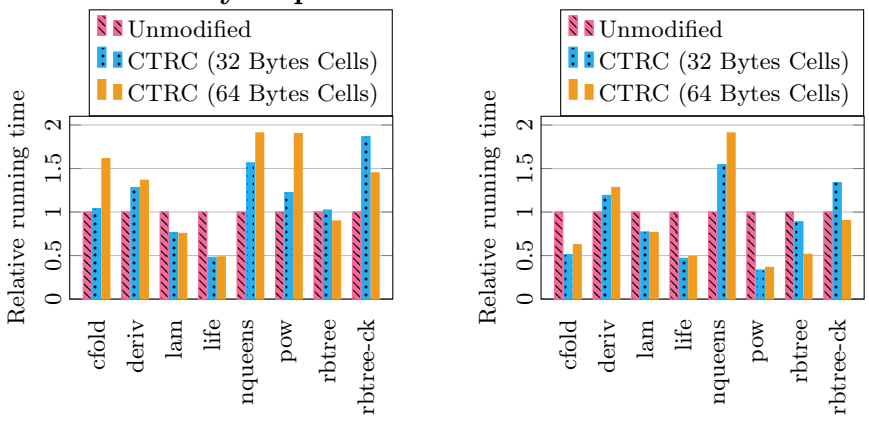
While functional programming typically uses small objects, some data structures require large contiguous memory allocation to be efficient, such as B-Trees [6] which benefit from a larger branching factor, and hash tables, which benefit from being able to perform random accesses in an array. However, increasing the fixed allocation size to satisfy these use cases causes a large increase in memory usage, and may impact performance due to worse cache utilization. Ideally, there should be a way of using variable-sized memory together with fixed-size constant-time memory management.

Eagerly-Deallocating-Allocation Effect (EDA), is an extension to CTRC that allows users to use variable-sized memory, without sacrificing the constant-time guarantee for the whole program. The idea is to utilize the effect system to mark parts of the program that perform variable-sized memory allocations which take non-constant time. Users can use the effect system to prohibit calling functions that perform variable-sized memory allocations in timing-sensitive regions of the program. Note that memory deallocation is still constant-time in all cases, and users can still perform read and write operations on those variable-sized

allocations from all parts of the program. Only variable-sized allocations take unbounded time, as they deallocate everything in the eager free list to recover space.

The runtime segregates the free list into a lazy free list and an eager free list. The eager free list contains all large allocations and objects that can transitively reach objects in the eager free list. For implementation, one can use 1 bit in the metadata to determine whether or not the object can transitively reach large objects. Deallocation operations put the object into the corresponding free list depending on this bit, and each operation still takes constant time. If there is insufficient memory to satisfy a variable-sized allocation, the allocator first tries to empty the eager free list, and then the lazy free list, to try to get enough space to fulfill the allocation. When both free lists are empty, the system should be garbage-free, so emptying the two free lists can make sure the system does not use more memory than necessary. The eager free list is emptied first to free large allocations, which are more likely to be able to satisfy the large allocation request. For normal fixed-size allocation, the allocator tries to reuse cells in the lazy free list. When the lazy free list is empty, the allocator uses cells in the eager free list, splitting large allocations when necessary. The allocator avoids splitting large allocations when possible, as splitting such allocation to fulfill small allocation requests may cause fragmentation.

3 Preliminary Experiments



(a) With reuse (b) Without reuse

Fig. 6: Relative Running Time

In this section, we discuss the initial benchmarks of CTRC, implemented by modifying the compiler and runtime system of Koka, versus Koka with the mimalloc [13] memory allocator.

3.1 Experimental Setup

Our CTRC implementation is extremely simple, the allocator runtime contains about 150 lines of C code that *does not depend on the system memory allocator*.

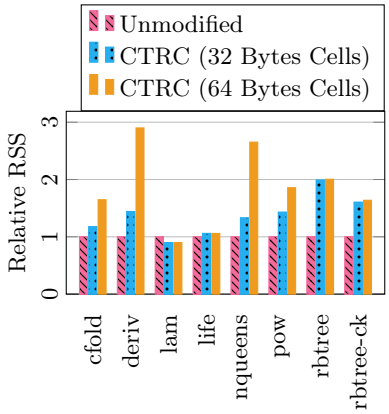


Fig. 7: Relative RSS (lower is better).

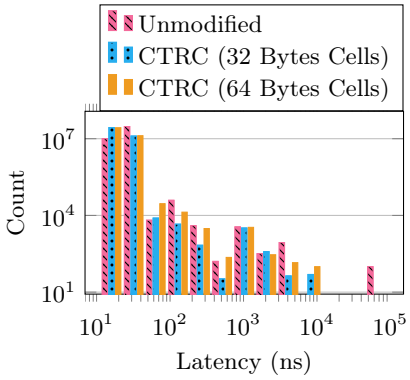


Fig. 8: Histogram of statement latencies (Section 3.3).

We included the implementation of the allocator in Appendix B, which uses mmap for allocating new pages but can be modified to use memory from a static buffer, and is simple to port to embedded systems. We modified the Koka compiler[2] to limit the size of each allocation and split the object when necessary. In the benchmark, we compare the performance of different cell sizes (32 bytes and 64 bytes) as cell size impacts both the performance and memory usage of the application.

Note that neither of the CTRC implementations used here supports hybrid reference counting, which is the ability to mix different eager and lazy allocation styles within the same program, as described in Section 2.5. We anticipate that adding support for hybrid reference counting would be straightforward and would not significantly alter these results.[3]

We run the benchmarks that were included in the work by Reinking et al. [19], as well as a few benchmarks adapted from NoFib [16], a Haskell benchmark suite, that stress memory allocation. Each benchmark is run in a loop 100 times in Koka, to avoid measuring only the startup and termination overhead of the runtime. All the benchmarks are run on a desktop computer with Intel i5-13600KF with 32GiB of RAM, running NixOS 23.11 with Linux 6.5.12 Xanmod kernel. The benchmarks are compiled with -O3 optimization using GCC 11.3.0. The benchmark run is pinned on a performance core with `taskset`, with SMT and turbo boost disabled to ensure all the code is run at the same CPU frequency. The relative execution time and memory usages are given in Figure 6a, Figure 6b and Figure 7. By default, Koka compiles with reuse optimization [19] enabled, which reduces the number of deallocations by performing in-place updates when the reference to the data structure is unique. Figure 6b shows the benchmark result

[2] Commit hash: `b167030`

[3] The only change needed for hybrid reference counting in the CTRC allocator implementation is the addition of a check for the dirty bit stored in object headers.

in which the reuse optimization is disabled, which demonstrates the performance when such reuse optimization is inapplicable or not implemented, for example when run in an interpreter.

3.2 Performance and Memory Usage

From Figure 6a, CTRC with a 32-byte cell size has similar performance compared with Koka unmodified, except in the N-queens (nqueens) and red-black tree (rbtree-ck) benchmarks. For nqueens, this is because it mostly uses integer cons lists, where each node only occupies 16 bytes in the 32-byte cell. Every access fetches some unused memory, which under-utilizes the memory bandwidth. This behavior is also apparent in most benchmarks when increasing the cell size from 32 bytes to 64 bytes: these benchmarks run slower in addition to using more memory. For rbtree-ck, the slowness is caused by the compiler splitting the left-child and right-child pointers into two segments. Tree traversal requires one additional pointer indirection, making the running time slower. Switching the cell size to 64 bytes removes the need for pointer indirection, which makes the running time faster in this case.

For the lambda evaluation (lam) and game of life (life) benchmarks, they are significantly faster compared with Koka unmodified because they involve deallocating large collections. For example, the life benchmark allocates and deallocates a large grid, and the lazy deallocation approach used by CTRC provides better temporal locality. CTRC with a 32-byte cell size has on average 8.8M L1 d-cache misses per iteration, while unmodified Koka has on average 26M L1 d-cache misses per iteration for the life benchmark. Similar behavior occurs when the reuse optimization is disabled, because in-place updates become deallocation and allocation of the same large collection, so CTRC becomes faster than unmodified Koka in these cases.

CTRC generally uses more memory compared to the baseline. For some benchmarks, 64-byte cell size can have a relatively high memory overhead because the object size is small (e.g. 16 bytes), wasting the remaining 48 bytes. However, the advantage of CTRC is that the maximal memory overhead can be determined statically, and is independent from the allocation/deallocation pattern.

Programmers can reduce the performance and memory overhead of CTRC by changing the datatypes to pack more data into each cell. For example, instead of using the usual cons list definition, one can use the following:

```
alias i32 = int32
type i32-list
  Nil
  Cons1{h1: i32;                                t: i32-list}
  Cons2{h1: i32; h2: i32;                       t: i32-list}
  Cons3{h1: i32; h2: i32; h3: i32;              t: i32-list}
  Cons4{h1: i32; h2: i32; h3: i32; h4: i32;     t: i32-list}
```

When compiled, the pointer `t` uses 8 bytes in 64-bit systems, and the maximal of 4 32-bit integer fields occupy 16 bytes in total. Together with the 8-byte

metadata per cell, this fully utilizes the 32-byte cell size. Note that these transformations will make the code more complicated, which may cause additional overhead if the program bottleneck is not caused by memory bandwidth limitation or cache misses. For example instead of simply using cons cells to add an element to the start of the list, the code now should check for the variant of the first cell and change the cell type accordingly. Also, the code transformation requires knowing the size of the fields, which may make it hard to apply the optimization for polymorphic types. For example, we can only unroll two fields if the data type used is a pointer instead of 32-bit integers.

There are also opportunities for data structure inlining [8], which is a well-known approach to optimizing program performance and memory footprint. Bruno et al. [4] uses value semantics to determine if data structures are eligible for inlining, where data objects with value semantics are not used for reference comparison and do not require atomic field access. In the context of functional programming, objects automatically have value semantics, so inlining can be applied to most objects except those behind reference cells. The problem is how to pack the objects such that fields accessed together are placed in the same cell, and how to maximize the utilization of cell size, which we leave as future work.

3.3 Latency Measurements

In the context of embedded programming, it is common to implement cooperative scheduling by explicitly yielding program control. This does not require a complicated program runtime and is more efficient, but requires careful coordination to meet latency requirements. Unbounded latency caused by recursive drop is particularly problematic in this scenario, because the programmer may think that every statement in the source language corresponds to a bounded number of steps in the machine execution. As the famous saying goes, "Any sufficiently complicated C or Fortran program contains an [...] implementation of half of Common Lisp", we implemented a simple lambda calculus interpreter to simulate the workload of complicated embedded programs and measured the latency per statement, to simulate the latency of cooperative scheduling. The result is shown in Figure 8, where "Count" is the number of statements that require a certain amount of time to execute.

From the figure, although most statements have low latency, unmodified Koka can occasionally get latency spikes of around 10 µs or higher, which are caused by deallocating large data structures. The latencies of around 1µs for unmodified Koka and CTRC are caused by page faults, and are unavoidable when running with virtual memory. This shows that the unbounded latency for eager reference counting can have a measurable impact, and is not only a theoretical problem, while CTRC mitigates the issue of unbounded latency in both memory allocation and deallocation.

4 Related Work and Conclusion

We now discuss related work and conclude.

4.1 Related Work

Reinking et al. [19] introduced a new algorithm for optimizing reference counting with memory reuse and specialization. Their work showed that reference counting can achieve comparable performance with state-of-the-art memory management systems, and sometimes even out-performing in terms of efficiency, while maintaining low memory usage and reasonable pause time. Our implementation is based on their work on the Koka compiler, which benefits from their reference counting optimization. Some of the optimization, such as reuse analysis, becomes more efficient in CTRC due to the constant allocation size guarantee.

Leijen et al. [13] implemented mimalloc, a fast memory allocator developed for Koka and Lean. Their implementation uses free list sharding to increase locality and reduce fragmentation. Free list sharding can be implemented in constant time for our approach, but experiments showed no consistent performance improvement due to worse temporal locality compared with a stack-like approach (LIFO). Our implementation achieves competitive performance when compared with Koka together with mimalloc, and the latter was already shown to be competitive with existing functional programming language implementations like GHC and OCamlc, which uses traditional tracing GC, as well as Swift, which uses reference counting. But while mimalloc is implemented in about 8k lines of code, our prototype implementation just takes around 150 lines of C code due to having fewer features and supporting Unix only.

Lazy reference counting is not a new concept. Back in 1963, Weizenbaum [24] introduced a list processing system that used lazy reference counting, and provided a simple implementation of the processing system in FORTRAN. However, due to various limitations of reference counting, such as the performance impact caused by frequent reference count updates and the inability to deal with cycles, tracing garbage collection is still the preferred way for performing automatic memory management in most high-level languages. Joisha [11, 12] used the idea of lazy reference counting to bound pause time in garbage collection. Instead of immediately collecting all the garbage, the deallocator maintains a list of zombie objects that have a reference count of zero but are yet to be reclaimed. This avoids triggering unbounded pause time when deallocating linked lists. However, as the system uses variable allocation size, the runtime may need to eagerly process all the zombie objects when there is insufficient space to serve certain allocation requests.

Boehm [3] analyzed the upper bound on the memory usage of lazy reference counting when multiple size classes are used. When the maximum and minimum cell size are $s_{\max}$ and $s_{\min}$ respectively and the number of live object is N, the space bound is $\frac{s_{\max}}{s_{\min}}N$. This upper bound is not better than using only one size class and promoting all small objects to the largest size class.

Puaut [18] evaluated the performance of various dynamic memory allocation approaches, comparing their analytical worst-case allocation time with their actual observed worst-case allocation time. Those algorithms tend to work well in practice and the observed worst-case allocation time is not too large compared with the average allocation time. However, their analytical worst-case time can

be very large due to the variable allocation sizes they support, and are unsuitable for hard real-time applications. On the other hand, our work allows for simple implementation, high throughput, and low analytical worst-case allocation time in the portions of the program that are statically proven by the type system not to have large allocation effects.

Bruno et al. [4] showed that object inlining can provide large improvements to throughput and reduce memory footprint. It may be possible to perform similar object inlining in CTRC to reduce the memory overhead of constant allocation size, and potentially improve performance by improving cache utilization.

Blelloch and Wei [2] gave a wait-free implementation for fixed-size allocation and free that is linearizable. This can potentially be used to implement the concurrency extension of CTRC, which requires balancing the global and thread-local heaps.

Blackburn and McKinley [1] introduced a hybrid garbage collector that combined both generational collector and reference-counting collector for high throughput and low maximum pause time. Their implementation divided the heap into an immortal part, a reference counted space for mature objects and a nursery space for short-lived objects with a high mutation rate. By deferring reference count updates for short-lived objects, the system's throughput can be improved. While they added a *time cap* parameter to limit the time spent on each garbage collection phase, our approach is inherently lazy and does not require tuning such ad hoc parameters. In addition, as shown in Reinking et al. [19], the throughput of reference counting with compile-time optimization is on par with or even better than state-of-the-art garbage collectors, without the complexity of a hybrid garbage collector.

Wan et al. [23] designed a statically-typed language called Real-Time FRP, that statically bounds the time and space cost of each execution step, which avoids unbounded allocation size and infinite recursion in the computation. However, in their formalism, the time cost of allocation and deallocation are not considered, as they only bound the term size. But allocation times can be significant in practice and could make their approach not truly real-time.

4.2 Conclusion

In this paper, we presented CTRC, a lazy reference counting system that provides a constant-time guarantee for memory allocation and deallocation operations. We also presented extensions for supporting a hybrid memory management strategy by utilizing a type-and-effect system, and discussed challenges and potential solutions for handling multithreaded allocation with memory sharing. While extremely concise, our implementation in Koka is competitive with the Koka runtime using mimalloc, a state-of-the-art memory allocator optimized for functional programming languages, which shows that our approach is practical and does not suffer from any significant performance penalty. We would like to experiment with other optimization techniques, such as object inlining [4] and profile-guided optimizations, to further reduce the performance and memory overhead of splitting objects. We leave as future work how to handle multithreaded allocation efficiently.

References

[1] Blackburn, S.M., McKinley, K.S.: Ulterior reference counting: fast garbage collection without a long wait. In: Crocker, R., Jr., G.L.S. (eds.) Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA, pp. 344–358, ACM (2003), https://doi.org/10.1145/949305.949336, URL https://doi.org/10.1145/949305.949336

[2] Blelloch, G.E., Wei, Y.: Concurrent fixed-size allocation and free in constant time (2020), https://doi.org/10.48550/ARXIV.2008.04296, URL https://arxiv.org/abs/2008.04296

[3] Boehm, H.J.: The space cost of lazy reference counting. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, p. 210–219, POPL '04, Association for Computing Machinery, New York, NY, USA (2004), ISBN 158113729X, https://doi.org/10.1145/964001.964019, URL https://doi.org/10.1145/964001.964019

[4] Bruno, R., Jovanovic, V., Wimmer, C., Alonso, G.: Compiler-assisted object inlining with value fields. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021, pp. 128–141, ACM (2021), https://doi.org/10.1145/3453483.3454034, URL https://doi.org/10.1145/3453483.3454034

[5] Collins, G.E.: A method for overlapping and erasure of lists. Commun. ACM **3**(12), 655–657 (dec 1960), ISSN 0001-0782, https://doi.org/10.1145/367487.367501, URL https://doi.org/10.1145/367487.367501

[6] Comer, D.: Ubiquitous b-tree. ACM Computing Surveys (CSUR) **11**(2), 121–137 (1979)

[7] Czaplicki, E., Chong, S.: Asynchronous functional reactive programming for guis. In: Boehm, H., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013, pp. 411–422, ACM (2013), https://doi.org/10.1145/2491956.2462161, URL https://doi.org/10.1145/2491956.2462161

[8] Dolby, J.: Automatic inline allocation of objects. In: Chen, M.C., Cytron, R.K., Berman, A.M. (eds.) Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI), Las Vegas, Nevada, USA, June 15-18, 1997, pp. 7–17, ACM (1997), https://doi.org/10.1145/258915.258918, URL https://doi.org/10.1145/258915.258918

[9] Doligez, D., Leroy, X.: A concurrent, generational garbage collector for a multithreaded implementation of ML. In: Deusen, M.S.V., Lang, B. (eds.) Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993, pp. 113–123, ACM Press (1993), https://doi.org/10.1145/158511.158611, URL https://doi.org/10.1145/158511.158611

[10] Elliott, C., Hudak, P.: Functional reactive animation. In: International Conference on Functional Programming (1997), URL http://conal.net/papers/icfp97/

[11] Joisha, P.G.: Compiler optimizations for nondeferred reference: Counting garbage collection. In: Proceedings of the 5th International Symposium on Memory Management, p. 150–161, ISMM '06, Association for Computing Machinery, New York, NY, USA (2006), ISBN 1595932216, https://doi.org/10.1145/1133956.1133976, URL https://doi.org/10.1145/1133956.1133976

[12] Joisha, P.G.: Overlooking roots: A framework for making nondeferred reference-counting garbage collection fast. In: Proceedings of the 6th International Symposium on Memory Management, p. 141–158, ISMM '07, Association for Computing Machinery, New York, NY, USA (2007), ISBN 9781595938930, https://doi.org/10.1145/1296907.1296926, URL https://doi.org/10.1145/1296907.1296926

[13] Leijen, D., Zorn, B., de Moura, L.: Mimalloc: Free list sharding in action. In: Lin, A.W. (ed.) Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings, Lecture Notes in Computer Science, vol. 11893, pp. 244–265, Springer (2019), https://doi.org/10.1007/978-3-030-34175-6_13, URL https://doi.org/10.1007/978-3-030-34175-6_13

[14] McBeth, J.H.: Letters to the editor: On the reference counter method. Commun. ACM **6**(9), 575 (sep 1963), ISSN 0001-0782, https://doi.org/10.1145/367593.367649, URL https://doi.org/10.1145/367593.367649

[15] Nilsson, H., Courtney, A., Peterson, J.: Functional reactive programming, continued. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, p. 51–64, Haskell '02, Association for Computing Machinery, New York, NY, USA (2002), ISBN 1581136056, https://doi.org/10.1145/581690.581695, URL https://doi.org/10.1145/581690.581695

[16] Partain, W.: The nofib benchmark suite of haskell programs. In: Launchbury, J., Sansom, P.M. (eds.) Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, UK, 6-8 July 1992, pp. 195–202, Workshops in Computing, Springer (1992), https://doi.org/10.1007/978-1-4471-3215-8_17, URL https://doi.org/10.1007/978-1-4471-3215-8_17

[17] Powers, B., Tench, D., Berger, E.D., McGregor, A.: Mesh: Compacting memory management for c/c++ applications. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, p. 333–346, PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019), ISBN 9781450367127, https://doi.org/10.1145/3314221.3314582, URL https://doi.org/10.1145/3314221.3314582

[18] Puaut, I.: Real-time performance of dynamic memory allocation algorithms. In: 14th Euromicro Conference on Real-Time Systems (ECRTS 2002), 19-21 June 2002, Vienna, Austria, Proceedings, pp. 41–49, IEEE Computer Society (2002), https://doi.org/10.1109/EMRTS.2002.1019184, URL https://doi.org/10.1109/EMRTS.2002.1019184

[19] Reinking, A., Xie, N., de Moura, L., Leijen, D.: Perceus: garbage free reference counting with reuse. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021,

pp. 96–111, ACM (2021), https://doi.org/10.1145/3453483.3454032, URL https://doi.org/10.1145/3453483.3454032

[20] Shahriyar, R., Blackburn, S.M., Frampton, D.: Down for the count? getting reference counting back in the ring. In: Proceedings of the 2012 International Symposium on Memory Management, p. 73–84, ISMM '12, Association for Computing Machinery, New York, NY, USA (2012), ISBN 9781450313506, https://doi.org/10.1145/2258996.2259008, URL https://doi.org/10.1145/2258996.2259008

[21] Ullrich, S., de Moura, L.: Counting immutable beans: Reference counting optimized for purely functional programming. In: Proceedings of the 31st Symposium on Implementation and Application of Functional Languages, IFL '19, Association for Computing Machinery, New York, NY, USA (2021), ISBN 9781450375627, https://doi.org/10.1145/3412932.3412935, URL https://doi.org/10.1145/3412932.3412935

[22] Wan, Z., Hudak, P.: Functional reactive programming from first principles. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, p. 242–252, PLDI '00, Association for Computing Machinery, New York, NY, USA (2000), ISBN 1581131992, https://doi.org/10.1145/349299.349331, URL https://doi.org/10.1145/349299.349331

[23] Wan, Z., Taha, W., Hudak, P.: Real-time FRP. In: Pierce, B.C. (ed.) Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001, pp. 146–156, ACM (2001), https://doi.org/10.1145/507635.507654, URL https://doi.org/10.1145/507635.507654

[24] Weizenbaum, J.: Symmetric list processor. Commun. ACM **6**(9), 524–536 (sep 1963), ISSN 0001-0782, https://doi.org/10.1145/367593.367617, URL https://doi.org/10.1145/367593.367617

A Formalization

In this section, we present the formal operational semantics of CTRC, prove its soundness, and show that it is garbage-free when the free list is empty (and the free list is always used for new allocations when non-empty).

Expressions

$$
\begin{array}{lll}
e ::= x \mid v \mid e\ e & & \text{(variable, value, application)} \\
\quad \mid \text{val } x = e; e & & \text{(bind)} \\
\quad \mid \text{dup } x;\ e & & \text{(duplicate)} \\
\quad \mid \text{drop } x;\ e & & \text{(drop)} \\
\quad \mid \text{match } x\ \{\ \overline{p_i \to e_i}^{\,n}\ \} & & \text{(match expr)} \\
v ::= \lambda^{\overline{y_i}^{\,n}} x.\ e & & \text{(function capturing } \overline{y_i}^{\,n}\text{)} \\
\quad \mid C\ \overline{v_i}^{\,n} & & \text{(constructor of arity } n\text{)} \\
p ::= C\ \overline{b_i}^{\,n} & & \text{(pattern)} \\
b ::= x \mid \_ & & \text{(binder or wildcard)}
\end{array}
$$

Syntactic Shorthands

$$
\begin{array}{lll}
e_1; e_2 & ::= \text{val } x = e_1; e_2\ x \notin \text{fv}(e_2) & \\
\lambda x.\ e & ::= \lambda^{\overline{y_i}^{\,n}} x.\ e & \overline{y_i}^{\,n} = \text{fv}(e) \\
\text{dropf } \lambda^{\overline{y_i}^{\,n}} x.\ e & ::= \overline{\text{drop } y_i}^{\,n} & \\
\text{dropf } C\ \overline{x_i}^{\,n} & ::= \overline{\text{drop } x_i}^{\,n} &
\end{array}
$$

Evaluation Judgments

$$
\begin{array}{rl}
e \longrightarrow & e' \text{ Baseline Semantics} \\
H \mid e \longrightarrow_{\mathrm{k}} H' \mid e' & \text{Reference Koka Semantics} \\
H; F \mid e \longrightarrow_{\mathrm{c}} H'; F' \mid e' & \text{New CTRC Semantics}
\end{array}
$$

Heap and Free List

$$
\begin{array}{ll}
\textit{(Heap)} & H : \quad x \to (\mathbb{N}^+, v) \\
\textit{(Free List)} & F ::= \varnothing \mid F, x
\end{array}
$$

Fig. 9: Syntax of λ^1.

A.1 Syntax

Figure 9 shows the syntax of λ^1 which is the same presented by Reinking et al. [19]. It is an untyped lambda calculus extended with explicit binding, pattern matching, as well as duplicate and drop instructions. Note that the duplicate and drop instructions are added by the compiler into the compiled program and are not written by the user. Constructors with fields $x_1, x_2, \ldots, x_n$ are denoted as $C\ \overline{x_i}^n$. Functions with parameter x, body e, free variables $y_1, y_2, \ldots, y_n$ are denoted as $\lambda^{\overline{y_i}^n} x.\ e$.

$$E ::= \Box \mid E\ e \mid x\ E \mid \text{val } x = E;\ e \mid C\ x_1 \ldots x_i\ E\ v_j \ldots v_n \qquad \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}\,[\text{EVAL}]$$

$$\begin{array}{llll}
(\textit{app}) & (\lambda x.\ e)\ v & \longrightarrow & e[x := v] \\
(\textit{bind}) & \text{val } x\ =\ v;\ e & \longrightarrow & e[x := v] \\
(\textit{match}) & \text{match}\ \ (C\ \overline{v_i}^n)\{\overline{p_j \to e_j}^m\} & \longrightarrow & e_j[\overline{x_i := v_i}^n] \text{ with } p_j = C\ \overline{x_i}^n \\
(\textit{no-op-1}) & \text{drop}\ \ x;\ e & \longrightarrow & e \\
(\textit{no-op-2}) & \text{dup}\ \ x;\ e & \longrightarrow & e
\end{array}$$

Fig. 10: Baseline Semantics.

We also define a few syntactic shorthands to simplify the presentation. We define sequence $e_1; e_2$ as binding e_1 to an unused variable x, functions are written as $\lambda x.\ e$ when the free variables are not important, and define dropf v for functions and constructors. dropf is used for dropping the fields of constructors and free variables of functions. It is a syntactic shorthand because it can be expanded into a fixed number of drops.

There are three different evaluation judgments, corresponding to different operational semantics.

Baseline Semantics The baseline semantics is the typical operational semantics that does not model memory management. Note that the syntax for match expression is modified to match $e\ \{\overline{p_i \to e_i}^n\}$, as the variable being matched is replaced with a value. The evaluation rules for the baseline semantics are shown in Figure 10, where the *app* rule is function application, the *bind* rule is variable binding, and the *match* rule is pattern matching. drop and dup instructions are ignored, as the baseline semantics does not model memory management and these two instructions are only for reference counting. In this paper, the baseline semantics serves as the baseline for program behavior, where the other two operational semantics should simulate. The *simulation* relation is shown with the simplified program trace defined below.

Reference Koka Semantics The reference Koka semantics, which we shall later refer to as the *eager* semantics, models memory management with

reference counting. The heap H is a mapping from variable to reference count and value. The evaluation judgment $H \mid e \longrightarrow_{\mathrm{k}} H' \mid e'$ reads as follows: given a heap H, the expression e is evaluated to e' with the heap updated to H'. This semantics is discussed in detail below.

New CTRC Semantics The new CTRC semantics, which can also be called the *lazy* semantics, models memory management with lazy reference counting. Instead of just a heap, the semantics also includes a free list F which stores reusable allocations. The evaluation judgment $H; F \mid e \longrightarrow_{\mathrm{c}} H'; F' \mid e'$ reads as follows: given a heap H and free list F, the expression e is evaluated to e', with the heap updated to H' and the free list updated to F'.

A.2 Reference Koka Semantics

$$E ::= \Box \mid E\ e \mid x\ E \mid \mathrm{val}\ x = E;\ e \mid C\ x_1 \ldots x_i\ E\ v_j \ldots v_n$$

$$\frac{H \mid e \longrightarrow_{\mathrm{k}} H' \mid e'}{H \mid E[e] \longrightarrow_{\mathrm{k}} H' \mid E[e']}\ [\textsc{eval}]$$

(new_{k})	H	$\mid v$	$\longrightarrow_{\mathrm{k}}$	$H, z \mapsto^1 v$	$\mid z$ fresh z
(app_{k})	H	$\mid f\ z$	$\longrightarrow_{\mathrm{k}}$	H	$\mid \boxed{\overline{\mathrm{dup}\ y_i;}^n\ \mathrm{drop}\ f;}\ e[x := z]$ with $(f \mapsto^m \lambda \overline{y_i}^n\ x.\ e) \in H$
$(match_{\mathrm{k}})$	H	$\mid \mathrm{match}\ x\ \{\overline{p_i \to e_i}^n\}$	$\longrightarrow_{\mathrm{k}}$	H	$\mid \boxed{\overline{\mathrm{dup}\ y_j;}^m\ \mathrm{drop}\ x;}\ e[\overline{x_j := y_j}^n]$ with $p_i = C\ \overline{x_i}^n \wedge (x \mapsto^n C\ \overline{y_j}^n) \in H$
$(bind_{\mathrm{k}})$	H	$\mid \mathrm{val}\ x = y;\ e$	$\longrightarrow_{\mathrm{k}}$	H	$\mid e[x := y]$
(dup_{k})	$H, x \mapsto^n v$	$\mid \boxed{\mathrm{dup}\ x;}\ e$	$\longrightarrow_{\mathrm{k}}$	$H, x \mapsto^{n+1} v$	$\mid e$
$(drop_{\mathrm{k}})$	$H, x \mapsto^{n+1} v$	$\mid \boxed{\mathrm{drop}\ x;}\ e$	$\longrightarrow_{\mathrm{k}}$	$H, x \mapsto^n v$	$\mid e$ if $n \geq 1$
$(free_{\mathrm{k}})$	$H, x \mapsto^1 v$	$\mid \boxed{\mathrm{drop}\ x;}\ e$	$\longrightarrow_{\mathrm{k}}$	H	$\mid \boxed{\mathrm{dropf}\ v;}\ e$

Fig. 11: Reference Koka semantics for λ^1.

Figure 11 shows the original[4] Koka reference-counted heap semantics. The evaluation context E uniquely determines where to apply an evaluation step. The $\boxed{\textit{boxes}}$ are tools we introduced to facilitate the proofs below. When the boxed instructions are removed, the semantics become the baseline semantics without reference counting, so the boxed instructions can be viewed as internal routines of the memory management scheme. Note that drop and dup instructions may be added by the compiler, i.e. they exist statically in the program, and are not boxed. They can also be internal routines of the memory management scheme, which are introduced on the right-hand side of evaluation rules and are boxed. We use a ⸢*dashed*⸣ box to denote instructions that can either be normal or boxed.

Values are allocated in the heap with rule new_{k} and evaluated to the variable pointing to the allocation. The freshly allocated variable has a reference count of 1.

[4] slightly modified to merge rules related to lambda and constructor for cleaner presentation

Function application with rule app_{k} duplicates the captured values of the function, drop the function allocation itself, and then perform the actual application via substitution. Similarly, for pattern matching, rule $match_{\mathrm{k}}$ duplicates the fields of the constructor, drop the constructor object itself, and substitute the fields to the pattern in the matched case. The $drop_{\mathrm{k}}$ and dup_{k} rules update the reference count of the target variable. When the reference count reaches 1, the drop instruction instead is evaluated according to $free_{\mathrm{k}}$, which drops the fields of the value and deallocates the allocation.

We define *simplified program trace* as the sequence of program states when executed according to some operational semantics, excluding the heap, free list and all steps that have boxed instructions. The simplified program trace corresponds to the execution trace of the baseline semantics, and should be the same for both the reference Koka semantics and the CTRC semantics.

For example, the full program-trace of $\mathrm{val}\ x = C_1;\ \mathrm{dup}\ x;\ \mathrm{val}\ y = \lambda^x z.\ x;\ \mathrm{drop}\ x;\ \mathrm{drop}\ y;\ \lambda x.\ x$ according to the reference Koka semantics is

1.	$\varnothing$	$\mathrm{val}\ x = C_1;\ \mathrm{dup}\ x;\ \mathrm{val}\ y = \lambda^x z.\ x;\ \mathrm{drop}\ x;\ \mathrm{drop}\ y;\ \lambda x.\ x$
2. (new_{k})	$u \mapsto^1 C_1$	$\mathrm{val}\ x = u;\ \mathrm{dup}\ x;\ \mathrm{val}\ y = \lambda^x z.\ x;\ \mathrm{drop}\ x;\ \mathrm{drop}\ y;\ \lambda x.\ x$
3. ($bind_{\mathrm{k}}$)	$u \mapsto^1 C_1$	$\mathrm{dup}\ u;\ \mathrm{val}\ y = \lambda^u z.\ u;\ \mathrm{drop}\ u;\ \mathrm{drop}\ y;\ \lambda x.\ x$
4. (dup_{k})	$u \mapsto^2 C_1$	$\mathrm{val}\ y = \lambda^u z.\ u;\ \mathrm{drop}\ u;\ \mathrm{drop}\ y;\ \lambda x.\ x$
5. (new_{k})	$u \mapsto^2 C_1, w \mapsto^1 \lambda^u z.\ u$	$\mathrm{val}\ y = w;\ \mathrm{drop}\ u;\ \mathrm{drop}\ y;\ \lambda x.\ x$
6. ($bind_{\mathrm{k}}$)	$u \mapsto^2 C_1, w \mapsto^1 \lambda^u z.\ u$	$\mathrm{drop}\ u;\ \mathrm{drop}\ w;\ \lambda x.\ x$
7. ($drop_{\mathrm{k}}$)	$u \mapsto^1 C_1, w \mapsto^1 \lambda^u z.\ u$	$\mathrm{drop}\ w;\ \lambda x.\ x$
8. ($free_{\mathrm{k}}$)	$u \mapsto^1 C_1$	$\boxed{\mathrm{drop}\ u;}\ \lambda x.\ x$
9. ($free_{\mathrm{k}}$)	$\varnothing$	$\lambda x.\ x$
10. (new_{k})	$y \mapsto^1 \lambda x.\ x$	y

Each row above shows the rule used to arrive at the current state, current heap and the resulting expression. The *simplified program trace* contains states $1-7, 9-10$. State 8 is excluded from the simplified trace because it contains boxed instructions.

A.3 New CTRC Semantics

We define the operational semantics for constant-time reference-counted heap in Figure 12, i.e. the *lazy* semantics. The reference count in the heap can now be zero, indicating the value is no longer reachable and is added to the free list. The free list, which is denoted by F, contains a list of memory locations that the program can reuse.

The major differences between the reference Koka semantics and the CTRC semantics are the allocation and deallocation rules. When the free list is empty, allocation requests are met by requesting more memory from the system according to rule new_{c}, which is the same as the rule new_{k} in the reference Koka semantics. When the free list is non-empty, however, the first entry in the free list is used to meet the request, and the fields in the original value of the entry are dropped according to the rule $newr_{\mathrm{c}}$, where the r suffix stands for reuse.

For the previous example, the program trace is

1.		$\varnothing$	$;\varnothing$	$\mid \text{val } x = C_1;\ \text{dup } x;\ \text{val } y = \lambda^x z.\ x;\ \text{drop } x;\ \text{drop } y;\ \lambda x.\ x$
2.	(new_c)	$u \mapsto^1 C_1$	$;\varnothing$	$\mid \text{val } x = u;\ \text{dup } x;\ \text{val } y = \lambda^x z.\ x;\ \text{drop } x;\ \text{drop } y;\ \lambda x.\ x$
3.	$(bind_c)$	$u \mapsto^1 C_1$	$;\varnothing$	$\mid \text{dup } u;\ \text{val } y = \lambda^u z.\ u;\ \text{drop } u;\ \text{drop } y;\ \lambda x.\ x$
4.	(dup_c)	$u \mapsto^2 C_1$	$;\varnothing$	$\mid \text{val } y = \lambda^u z.\ u;\ \text{drop } u;\ \text{drop } y;\ \lambda x.\ x$
5.	(new_c)	$u \mapsto^2 C_1, w \mapsto^1 \lambda^u z.\ u$	$;\varnothing$	$\mid \text{val } y = w;\ \text{drop } u;\ \text{drop } y;\ \lambda x.\ x$
6.	$(bind_c)$	$u \mapsto^2 C_1, w \mapsto^1 \lambda^u z.\ u$	$;\varnothing$	$\mid \text{drop } u;\ \text{drop } w;\ \lambda x.\ x$
7.	$(drop_c)$	$u \mapsto^1 C_1, w \mapsto^1 \lambda^u z.\ u$	$;\varnothing$	$\mid \text{drop } w;\ \lambda x.\ x$
8.	$(free_c)$	$u \mapsto^1 C_1, w \mapsto^0 \lambda^u z.\ u$	$;w$	$\mid \lambda x.\ x$
9.	$(newr_c)$	$u \mapsto^1 C_1, w \mapsto^1 \lambda x.\ x$	$;\varnothing$	$\mid \boxed{\text{drop } u;}\ w$
10.	$(free_c)$	$u \mapsto^0 C_1, w \mapsto^1 \lambda x.\ x$	$;u$	$\mid w$

Each row above shows the rule used to arrive at the current state, current heap, current free list, and the expression being evaluated. The simplified program trace is the above trace excluding step 9. Note that the full trace for both the eager semantics and the lazy semantics are very similar, except in the last few steps where they treat free and allocation differently. For the eager semantics, step 8 recursively drops the field of w, while the lazy semantics just put w into the free list. The field of w is dropped when there are new allocation requests, which happens in step 9 above. When the states involving boxed instructions are removed, the simplified traces for both semantics are the same and correspond to the baseline semantics.

$$E ::= \Box \mid E\ e \mid v\ E \mid \text{val } x = E;\ e \mid C\ x_1 \ldots x_i\ E\ v_j \ldots v_n$$

$$\frac{H; F \mid e \longrightarrow_c H'; F' \mid e'}{H; F \mid E[e] \longrightarrow_c H'; F' \mid E[e']}\ [\text{EVAL}]$$

(new_c)	$H; \varnothing$	$\mid v$	$\longrightarrow_c H, z \mapsto^1 v; \varnothing$	$\mid z$	fresh z
$(newr_c)$	$H, x \mapsto^0 v; F, x$	$\mid v'$	$\longrightarrow_c H, x \mapsto^1 v'; F$	$\mid \boxed{\text{dropf } v;}\ x$	
(app_c)	$H; F$	$\mid f\ z$	$\longrightarrow_c H; F$	$\mid \boxed{\overline{\text{dup } y_i;}^m \text{drop } f;}\ e[x := z]$	
			with $(f \mapsto^n \lambda^{\overline{y_i}^n} x.\ e) \in H$		
$(match_c)$	$H; F$	$\mid \text{match } x\ \{\overline{p_i \to e_i}^m\}$	$\longrightarrow_c H; F$	$\mid \boxed{\overline{\text{dup } y_j;}^n \text{drop } x;}\ e[\overline{x_j := y_j}^n]$	
			with $p_i = C\ \overline{x_i}^n \wedge (x \mapsto^n C\ \overline{y_j}^n) \in H$		
$(bind_c)$	$H; F$	$\mid \text{val } x = y;\ e$	$\longrightarrow_c H; F$	$\mid e[x := y]$	
(dup_c)	$H, x \mapsto^n v; F$	$\mid \boxed{\text{dup } x;}\ e$	$\longrightarrow_c H, x \mapsto^{n+1} v; F$	$\mid e$	
$(drop_c)$	$H, x \mapsto^{n+1} v; F$	$\mid \boxed{\text{drop } x;}\ e$	$\longrightarrow_c H, x \mapsto^n v; F$	$\mid e$	if $n \geq 1$
$(free_c)$	$H, x \mapsto^1 v; F$	$\mid \boxed{\text{drop } x;}\ e$	$\longrightarrow_c H, x \mapsto^0 v; F, x$	$\mid e$	

Fig. 12: Constant-time heap semantics for λ^1

A.4 Metatheory

In this section, we prove the correctness of the CTRC semantics. We show that the simplified program trace for the reference Koka semantics and the CTRC

semantics are equal. From this, we derive that the CTRC semantics never reuse memory before the reference Koka semantics drop them. By the soundness of the reference Koka semantics, the CTRC is also sound because it cannot cause memory corruption. We then prove that the system is garbage-free when the free list is empty. As CTRC would not request memory from the system when the free list is non-empty, it would not allocate more memory than needed. This property is also one that enables the eager-deallocating-allocation effect extension to work (see Section 2.5). At last, we show that each memory instructions of CTRC only perform a statically-bounded number of steps, which provides the constant-time guarantee as promised.

Lemma 1. *The eager semantics and lazy semantics simulate the baseline semantics.*

Proof. Boxed instructions do not add any non-boxed instructions when evaluated, so for the simplified program trace, we can safely remove them from the rules. The resulting rules are the same for both semantics, so their simplified program traces are the same.

With the simulation relation, we can define time in program execution by the position in the simplified trace, i.e. according to the baseline semantics. We denote the reference count of variable x at a certain time when executed according to the eager semantics and the lazy semantics by x_k and x_c respectively.

Lemma 2. *At any point in the program execution, we have $x_\mathrm{k} \leq x_\mathrm{c} - x_f$, where x_f is the number of times x occurs as a field of variables that are freed in the eager semantics but not reused in the lazy semantics.*

Proof. First, notice that if the proposition holds, the lazy execution never reuses memory before the eager execution deallocates the variable. This is because in order for the lazy execution to reuse memory, it has to execute the new_c rule, whereas the new_k rule of the eager semantics does not deallocate anything. Let x'_k and x'_c be the reference count after this step, we know that $x'_\mathrm{c} = 0$ because we deallocate in this step, and $x_\mathrm{k} = x'_\mathrm{k}$ as the eager semantics do not deallocate in this step, we have $x_\mathrm{k} = x'_\mathrm{k} \leq x'_\mathrm{c} = 0$ so x is already deallocated in the eager execution.

Now we prove the proposition by induction on the evaluation rules.

Case $H = \varnothing$**.** Initially, the free list and heap are empty, so the proposition holds trivially.

Case *new***.** For allocation expressions, the new_k and $new_\mathrm{c}/newr_\mathrm{c}$ rules are executed. In both semantics, the newly allocated value has $x_\mathrm{k} = x_\mathrm{c} = 1$ and is not freed in the eager semantics, so the proposition holds for the newly allocated value.

We now prove that the proposition still holds for all the original fields of the value being reused. Notice that for the original eager semantics, it cannot perform deallocation when evaluating allocation, so f_k is not changed. For $newr_\mathrm{c}$, the fields of the old deallocated value *old* are dropped, so reference

count f_c for the field f is decremented n times, where n is the number of occurrences of the variable in the fields of the deallocated object. f_f is also decremented by n, because *old* is now reused in the lazy semantics, and its fields no longer contribute to f_f, so the inequality still holds for f.
For other objects y, as the lazy semantics does not perform recursive drop, the reference counts are not being changed. Also, as the eager semantics does not perform reference count update in the case of allocation, except for the newly allocated value, y_f will not change, and the inequality still holds.

Case $free$ For $free_k$ and $free_c$ rules, it is easy to see that $free_k$ decrements the reference count x_k of every field x of the deallocated value by n, while $free_c$ causes x_f to increase by n and no change in x_c. So the proposition still holds for all fields of the deallocated value.

Other cases For other rules, both semantics have the same behavior so they do not affect the invariant.

Corollary 1. *The lazy semantics is sound, i.e. it only reuses garbage that would have been deallocated in the eager semantics.*

We now prove the garbage-free property for this lazy semantics, and the proof also shows that one can perform garbage collection and get to the same state as in the eager semantics.

Lemma 3. *When the points-to graph is acyclic and the free list is empty,* $v_k = v_c$.

Proof. Note that we only have to count the number of drop calls because dups are treated the same in both semantics, and drops are commutative so order does not matter.

By induction on the longest distance from the root set in the points-to graph. If the longest distance is zero, this holds because the reference count can only be n, where n is the number of dup and drop calls, as there are no references to the variable. For the induction case, note that every pointer pointing to the current object has a strictly smaller longest distance, and the induction hypothesis holds for them. If the pointer is from some garbage, by the induction hypothesis the reference count of the garbage is the same as in the eager semantics. Because the eager semantics is garbage-free, the reference count of the garbage has 0 reference count, which should already be dropped and added to the free list. As the free list is empty, the memory is already being reused by the ($newr_c$) rule and the fields are dropped. Hence, the reference count of the current object is equal to the number of live objects pointing to it, which is the same as in the eager semantics.

Note that the proof requires an acyclic heap, which is also a property required for reference counting to work. For functional programming languages without mutation, with suitable compilation strategy, programs can guarantee to have no reference cycles.

Corollary 2. *Acyclic heaps are garbage-free when the free list is empty.*

The relationship between eager reference counting and lazy reference counting is shown in Figure 5. The heap is originally garbage-free as there is no allocation. When users perform deallocation, eager deallocation removes all garbage associated with the object, while lazy deallocation turns the heap into the CTRC heap. When the user empties the free list of the CTRC heap, the heap becomes garbage-free again.

Theorem 1 (Constant-time memory management). *Each memory management instruction takes constant time with the CTRC semantics.*

Proof. There are three cases to consider:

Case dup This instruction is evaluated according to dup_c in 1 step.

Case drop This instruction can be evaluated according to $drop_c$ or $free_c$, where both of them can be evaluated in 1 step. $free_c$ requires appending a variable to the free list, which can be implemented in constant time with a linked list.

Case Allocation There are two cases for allocation, depending on whether the free list is empty.

Subcase Empty free list Allocation is evaluated according to rule new_c, which requests memory from the system in 1 step.

Subcase Non-empty free list Allocation reuses an allocation from the free list and drops all its fields according to rule $newr_c$. As we assume the number of fields is statically bounded, and each drop instruction takes a statically-bounded amount of CPU operations, the whole operation takes a statically-bounded amount of CPU operations.

Note that the formalization is different from the actual implementation, we do not distinguish between objects and segments. The compiler is responsible for splitting objects into segments, satisfying the constant size requirement. We do not model this compiler transformation because there can be many different implementations, and our operational semantics do not depend on such details. As the size is bounded, the number of fields of each object is also bounded.

B CTRC Allocator Source Code

In this appendix, we present our implementation of basic CTRC (without the locality optimization).

The `defer_drop` function is used for deallocating objects, and the `get_block` function is used for allocating new objects.

Header initialization and reference-count updates are handled in the Koka runtime.

```
1 #include "kklib.h"
2 #include <sys/mman.h>
3
4 #define STACK_NODE_PAGES 1ull
5 #define NUM_CELLS_PER_PAGE ((4096 * STACK_NODE_PAGES /
      SMALL_BLOCK) - 1)
```

```
#define MAGIC_BITS 0xCA
#define SPLIT_BIT 0x8

typedef union ctrc_cell_s {
  struct {
    kk_header_t header;
    uint8_t data[SMALL_BLOCK - 8];
  };
  struct {
    union ctrc_cell_s *next;
  };
} ctrc_cell_t;
_Static_assert(sizeof(ctrc_cell_t) == SMALL_BLOCK, "ctrc_cell_t???");

typedef struct ctrc_page_s {
  ctrc_cell_t *free_ptr;
  ctrc_cell_t *drop_ptr;
  struct ctrc_page_s *next_page;
  uint64_t free_counter;
  ctrc_cell_t cells[NUM_CELLS_PER_PAGE];
} ctrc_page_t;
_Static_assert(sizeof(ctrc_page_t) == 4096 * STACK_NODE_PAGES,
               "ctrc_page_t???");

static ctrc_page_t *last_page = NULL;

static inline void drop_cell(ctrc_cell_t *cell) {
  // find page
  size_t page_addr = (size_t)cell & ~(((size_t)
    STACK_NODE_PAGES * 4096) - 1);
  ctrc_page_t *page = (ctrc_page_t *)page_addr;
  __builtin_prefetch(page, 1);

  if (kk_unlikely(cell->header._field_idx != MAGIC_BITS))
    return;
  kk_ssize_t scan_fsize = cell->header.scan_fsize;
  // avoid corrupting the pointer part
  scan_fsize &= (SMALL_BLOCK - 1);
  bool new_page = page->drop_ptr == NULL && page->free_ptr ==
     NULL &&
                  page->free_counter == 0;
  if (new_page) {
    page->next_page = last_page;
    last_page = page;
  }
  if (scan_fsize == 0) {
    cell->next = page->free_ptr;
    page->free_ptr = cell;
```

```
52    } else {
53      cell->next = (ctrc_cell_t *)((size_t)page->drop_ptr |
        scan_fsize);
54      page->drop_ptr = cell;
55    }
56  }
57
58  void force_free(kk_block_t *block) {
59    // ignore
60    if (kk_unlikely(block->header._field_idx != MAGIC_BITS))
61      return;
62    if (block->header.scan_fsize & SPLIT_BIT) {
63      // splitted object
64      kk_box_t box = kk_block_field(block, 0);
65      kk_assert(kk_box_is_ptr(box));
66      kk_block_t *next_block = kk_ptr_unbox(box);
67      // ignore other ptrs...
68      next_block->header.scan_fsize &= SPLIT_BIT;
69      drop_cell((ctrc_cell_t *)next_block);
70    }
71    block->header.scan_fsize = 0;
72    drop_cell((ctrc_cell_t *)block);
73  }
74
75  static void drop_fields(kk_block_t *block) {
76    kk_ssize_t scan_fsize = block->header.scan_fsize & (
        SMALL_BLOCK - 1);
77    if (scan_fsize & SPLIT_BIT) {
78      scan_fsize = (scan_fsize & (~SPLIT_BIT)) + 1;
79      if (scan_fsize == 1) {
80        force_free(kk_ptr_unbox(kk_block_field(block, 0)));
81        return;
82      }
83    }
84    kk_context_t *context = kk_get_context();
85    for (kk_ssize_t i = 0; i < scan_fsize; i++) {
86      kk_box_drop(kk_block_field(block, i), context);
87    }
88  }
89
90  static ctrc_page_t *mmap_cache;
91  static size_t mmap_cache_count = 0;
92  static bool use_htlb = true;
93
94  static ctrc_page_t *alloc_blocks() {
95    if (kk_unlikely(mmap_cache_count-- == 0)) {
96      unsigned long long size =
97          use_htlb ? (32ull * 1024ull * 1024ull) : (64ull *
        1024ull);
98      mmap_cache =
```

```
99          mmap(NULL, size, PROT_WRITE | PROT_READ,
100              MAP_PRIVATE | MAP_ANONYMOUS | (use_htlb ?
       MAP_HUGETLB : 0), -1, 0);
101       if (kk_unlikely(mmap_cache == MAP_FAILED)) {
102         if (use_htlb) {
103           use_htlb = false;
104           mmap_cache_count++;
105           return alloc_blocks();
106         }
107         fprintf(stderr, "allocation␣error:␣%s\n", strerror(
       errno));
108         exit(1);
109       }
110       madvise(mmap_cache, size, MADV_POPULATE_WRITE |
       MADV_WILLNEED);
111       mmap_cache_count = size / sizeof(ctrc_page_t) - 1;
112     }
113     ctrc_page_t *page = mmap_cache++;
114     page->drop_ptr = NULL;
115     page->free_ptr = NULL;
116     page->free_counter = NUM_CELLS_PER_PAGE;
117     return page;
118   }
119
120   static ctrc_cell_t *pop_free() {
121     if (kk_unlikely(last_page == NULL)) {
122       ctrc_page_t *page = alloc_blocks();
123       page->next_page = last_page;
124       last_page = page;
125     }
126     ctrc_cell_t *result = last_page->free_ptr;
127     __builtin_prefetch(result, 1);
128     bool need_drop = false;
129     if (result == NULL) {
130       if (last_page->free_counter > 0) {
131         result = &last_page->cells[--last_page->free_counter];
132       } else {
133         result = last_page->drop_ptr;
134         size_t next_ptr = (size_t)result->next;
135         result->header.scan_fsize = next_ptr & (SMALL_BLOCK -
       1);
136         last_page->drop_ptr = (ctrc_cell_t *)(next_ptr & ~(
       SMALL_BLOCK - 1));
137         need_drop = true;
138       }
139     } else {
140       last_page->free_ptr = result->next;
141     }
142     if (kk_unlikely(last_page->drop_ptr == NULL && last_page->
       free_ptr == NULL &&
```

```
143                   last_page->free_counter == 0)) {
144     last_page = last_page->next_page;
145     __builtin_prefetch(last_page, 0);
146   }
147   if (need_drop)
148     drop_fields((kk_block_t *)result);
149   return result;
150 }
151
152 void defer_drop(kk_block_t *block) { drop_cell((ctrc_cell_t
      *)block); }
153
154 kk_block_t *get_block() {
155   kk_block_t *block = (kk_block_t *)pop_free();
156   block->header._field_idx = MAGIC_BITS;
157   return block;
158 }
```