On the Effects of Integrating Region-based Memory Management and Generational Garbage Collection in ML

Martin Elsman[1] ✉
[0000−0002−6061−5993]
mael@di.ku.dk

Niels Hallenberg[2]
[0000−0002−4180−860X]
niels.hallenberg@simcorp.com

[1] University of Copenhagen, Denmark
[2] SimCorp A/S, Denmark

Abstract. We present a region-based memory management scheme with support for generational garbage collection. The scheme is implemented in the MLKit Standard ML compiler, which features a compile-time region inference algorithm. The compiler generates native x64 machine code and deploys region types at runtime to avoid write barrier problems and to support partly tag-free garbage collection. We measure the characteristics of the scheme, for a number of benchmarks, and compare it to the Mlton state-of-the-art Standard ML compiler and configurations of the MLKit with and without region inference and generational garbage collection enabled. Although region inference often serves the purpose of generations, we demonstrate that, in some cases, generational garbage collection combined with region inference is beneficial.

Keywords: Region inference · Generational garbage collection

1 Introduction

Region-based memory management allows for programmers to associate the lifetimes of objects with so-called regions and to reason about how and when such regions are allocated and deallocated. Region-based memory management, as it is implemented for instance in Rust [2], can be a valuable tool for constructing critical systems, such as real-time embedded systems [25]. Region inference differs from explicit region-based memory management by taking a non-annotated program as input and producing as output a region-annotated program, including directives for allocating and deallocating regions [27]. The result is a programming paradigm where programmers can learn to write region-friendly code (by following certain patterns [28]) for essential parts of a program and perhaps retain a combination of region inference and garbage collection [17] for programs (or the parts of a program) that are not time critical.

Both region-inference and generational garbage collection have been shown to manage short-lived values well. In this paper we present a framework that combines these techniques, and discuss the effects of the integration.

The region-based memory management scheme that we consider is based on the stack discipline. Whenever e is some expression, region inference may decide to replace e with the term `letregion` ρ `in` e' `end`, where e' is the result of transforming the expression e, which includes annotating allocating expressions with particular region variables (e.g., ρ) specifying the region each value should be stored in. The semantics of the `letregion` term is first to allocate a region (initially an empty list of pages) on the region stack, bind the region to the region variable ρ, evaluate e', and, finally, deallocate the region bound to ρ (and its pages). The region type system allows regions to be passed to functions at run time (i.e., functions can be region-polymorphic) and to be captured in closures. The soundness of region inference ensures that a region is not deallocated as long as a value within it may be used by the remainder of the computation. When combining region inference and reference-tracing garbage collection, to remedy for the sometimes overly static approximation of liveness, we must be careful to rule out the possibility of deallocating regions with incoming pointers from live objects. Luckily, it turns out that such pointers can be ruled out by the region type system [8], which means that we can be sure that a tracing garbage collector will not be chasing dangling pointers at run time.

Our generational collector associates two generations with each region. It has the feature that an object is promoted to the old generation of its region (during a collection) only if it has survived a previous collection. Compared to the earlier non-generational collection technique [17], we may run a minor collection by only traversing (and copying) objects in the young generations.

The contributions of this paper are the following:

1. We present a technique for combining region-based memory management with a generational (stop the world) garbage collector, using a notion of typed regions, which allows us to deal with mutable data in minor collections and for tag-free representations of certain kinds of values such as tuples.
2. To demonstrate the absolute feasibility of the technique, we show empirically that the MLKit generates code that, in many cases, is comparable in performance to executables generated with the Mlton compiler (v20180207).
3. We demonstrate empirically that the combination of generational garbage collection and region-based memory management can lead to improved performance over using non-generational garbage collection but also that the increased memory waste (unused memory in region pages), caused by having multiple generations associated with each region, sometimes leads to an overhead compared to when a non-generational collection strategy is used.
4. We demonstrate empirically that when combined with generational garbage collection, region inference will take care of reclaiming most of the data in young generations with the effect that minor collections occur less often.

The study is performed in the context of the MLKit [28]. It generates native x64 machine code for Linux and macOS [9] and implements a number of techniques for refining the representations of regions [4, 27], including dividing regions into stack allocated (bounded) regions and heap allocated regions.

The paper is organised as follows. In Sect. 2, we present the generational garbage collection algorithm and how the algorithm is extended to work with mutable and large objects. In Sect. 3, we present a number of experimental results. In Sect. 4, we describe related work, and in Sect. 5, we conclude.

2 Generational Garbage Collection

A *region descriptor* represents an unbounded region and consists of a pointer to the previous region descriptor on the stack (p), a generation descriptor for the young generation (g_{y}), a generation descriptor for the old generation (g_{o}), and a list (L) for large objects, which are objects that do not fit in a region page; see Fig. 1. Each *generation descriptor* (g) consists of a pointer to a list of fixed-sized region pages (fp) and an allocation pointer (a).

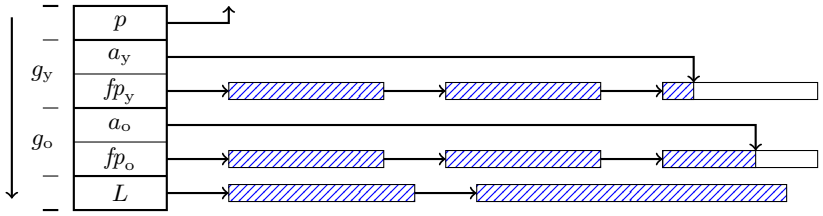

Fig. 1. A region descriptor on the down-growing stack. Region descriptors are linked, through "previous pointers" (p), hold generation descriptors (g_{y} and g_{o}), and hold a linked list of large objects (L).

The garbage collector we describe is a *generational* collector, which supports both minor and major collections. In a *minor* collection, only reachable objects allocated in young generations are traversed and *evacuated* (i.e., copied); those allocated in old generations are left untouched. In a *major* collection, all reachable objects are traversed and evacuated. In a minor collection, only reachable objects allocated in young generations are traversed, but a minor collection does not differentiate between in which region an object is stored, as there can be pointers from objects in newer regions to objects in older regions.

Consider a region r_2 above a region r_1 on the stack, with two generations each. This scenario allows for *deep* pointers from r_2 pointing to objects in region r_1 as shown in Fig. 2 (labeled 1 to 4) and *shallow* pointers pointing from objects allocated in region r_1 into objects allocated in region r_2 (labeled 5 to 8). Shallow pointers only exist between regions allocated in the same `letregion` construct, which is a sufficient requirement to rule out the possibility of dangling pointers [8, 17]. The scheme that we first describe does not allow for pointers to point from an old generation to a young generation (i.e., the pointers labeled 3 and 7); mutable objects, which may violate this principle, are treated later in Sect. 2.3.

When an object in a young generation of a region is evacuated, the object may be promoted to the old generation of the region. The collector implements

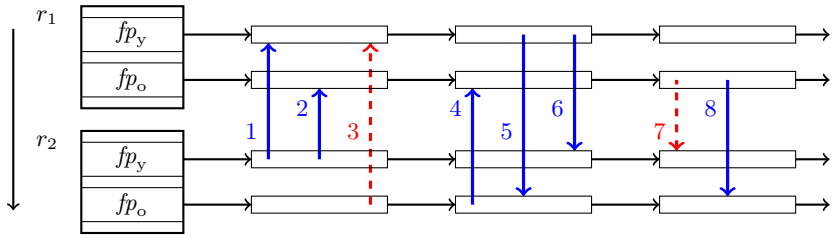

Fig. 2. Possible and impossible pointers. Impossible pointers are those that are dashed. The stack grows downwards. Shallow pointers (e.g., pointers from values in r_1 to values in r_2) are allowed only between regions that are allocated and deallocated simultaneously (e.g., a list's elements are stored independently from the spine of the list.)

the following promotion strategy, which guarantees that only long-living values are promoted to old generations:

Definition 1 (Promotion Strategy). *Promote objects when they have survived precisely one collection. The first time a value in a region r is evacuated, the value stays allocated in the young generation. During the following garbage collection, the value is promoted (moved) to the old generation of r.*

During a minor garbage collection, objects that have survived one collection must be promoted to the old generation, whereas objects that have not yet survived a collection should remain in the young generation. However, the implementation must preserve a *generation upward-closure* property, which states that, after a collection, whenever a value v has been promoted to an old generation, all values v' pointed to by v are also residing in old generations.

Fig. 3 shows two regions and their young generations. The *black areas* contain objects that have survived one collection. The *white areas* signify objects that have been allocated since the last collection. Objects allocated in the black areas will be promoted to an old generation and objects allocated in the white area will stay allocated in a young generation. Fig. 3 shows different combinations of pointers from white and black areas into white and black areas.

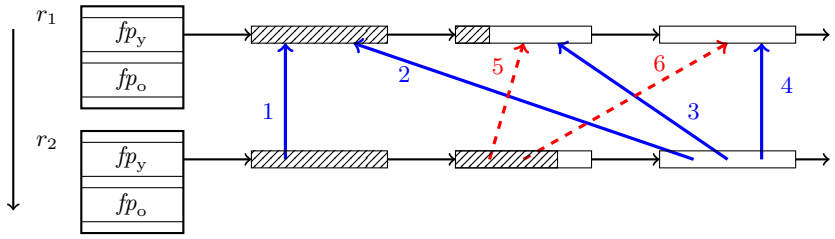

Fig. 3. The black areas contain objects that have survived one collection and white areas contain objects allocated since the last collection.

To implement the promotion strategy, the generation upward-closure invariant must disallow values in black areas to point at values in white areas (pointers 5 and 6 in Fig. 3):

Definition 2 (Generation Upward-Closure). *If a value resides in an old generation and points to a value v' then v' resides in an old generation. If a value resides in a black area in a young generation and points to a value v' then v' resides in an old generation or in a black area in a young generation.*

We now argue that the promotion strategy satisfies the Generation Upward-Closure invariant. The argument is a case-by-case analysis of the possible pointers shown in Fig. 3 (pointers 1, 2, 3, and 4), where each pointer takes the form $v_2 \rightarrow v_1$ and where v_2 is allocated in r_2 and v_1 is allocated in r_1:

Pointer 1. Both v_2 and v_1 reside in black areas, which means that, given v_2 is live, they will both be promoted to old generations according to the promotion strategy. The possibly promoted pointer will thus trivially satisfy Definition 2, part 1.

Pointer 2. If v_2 is live then it will be promoted to the black area of the young generation while v_1 is promoted to the old generation. The possibly promoted pointer will trivially satisfy Definition 2, part 2.

Pointer 3. Both v_2 and v_1 reside in white areas of young generations, which means that, given v_2 is live, they will both be promoted to black areas in young generations. Again, the possibly promoted pointer will trivially satisfy Definition 2, part 2.

Pointer 4. Similar to pointer 3.

Pointer 3 gives rise to some considerations because v_1 is allocated in a region page containing both a black and a white area. How do we mark v_1 as being allocated in a white area? One possibility is that we mark each object as being white or black, which will require that all objects are stored with a tag. A less costful solution, which we shall pursue, is to introduce the notion of a *region page color pointer* (*colorPtr*), which points at the first white value in the region page. Given a value v located at a position p in a region page and the color pointer *colorPtr* associated with the region page, if $p < colorPtr$ then v is allocated in the black area of the region page; otherwise, v is allocated in the white area.[3] Notice, that color pointers are updated and referenced only during a garbage collection; it does not change when allocating new values.

For the scheme to be sound, we need to make sure that pointers of the form of pointer 5 and pointer 6 never occur as the promotion strategy would otherwise lead to pointers from old generations to young generations, which would violate Definition 2. As we have shown, the garbage collector will never introduce such pointers and, luckily, neither will the mutator, except due to mutable data assignment, which we will treat in Sect. 2.3.

[3] In the implementation, the color pointer associated with a region page is located in the header of the page. If *colorPtr* points past the page, the entire page is black.

An alternative to the implemented promotion strategy is to add additional generations and let a minor collection traverse all objects except those in an oldest generation. Such a solution, however, could introduce a large amount of unused memory in region pages. Another promotion strategy would be to promote objects when they have survived a number ($N \geq 0$) of collections, which generalises the implemented promotion strategy, but is intractable as it requires tracking of the number of times each object in a young generation has survived a collection.

2.1 Evacuating Objects

The *evacuation* process copies live objects into fresh pages so that the copied-from pages can be reclaimed, including the parts of the pages that hold unreachable values. Definition 2 is implemented as follows. During a major collection, the collector will evacuate objects from old generations into old generations. During a minor collection, however, old generations will be left untouched and the collector will not attempt at traversing values stored in old generation pages. During a major or a minor collection, the collector will evacuate objects in young generation white areas into young generation black areas. Moreover, the collector will evacuate objects in young generation black areas into old generations. The evacuation strategy is implemented by marking all region pages in old generations black, which means that the same algorithm can be used to evacuate objects in minor and major collections. All objects in black areas are copied into black areas in old generations. All objects in white areas are copied into black areas in young generations. All objects allocated between two collections are allocated in white areas in young generations.

Before a major collection, all region pages are assembled to form the from-space as shown in Fig. 4. For a minor collection, from-space is formed from all young generation pages. After a collection (minor or major), the from-space pages are added to the free-list of pages.

Fig. 4. From-space contains black region pages from old generations, black region pages from young generations, white region pages from young generations, and partly-white region pages from young generations. No white region pages from old generations exist.

To distinguish pointers from non-pointers, integers and other unboxed values (e.g., booleans and enumeration datatypes) are represented as tagged values with the least significant bit set. Records are represented as a vector of values with a prefix tag word, which is used by the collector to identify the number of record components. Pairs and triples, however, are represented without a prefix tag word. Given a pointer to a value in a region page, the collector can determine

that the value is a pair or a triple by inspecting the region type associated with the region in which the object resides. In practice, the implementation works with the region types RTY_BOT, RTY_PAIR, RTY_TRIPLE, RTY_DOUBLE, RTY_REF, RTY_ARRAY, and RTY_TOP. Here the region type RTY_TOP is used for specifying regions that can contain values of arbitrary type, except those associated with the other region types. The region type RTY_BOT never occurs at run time, but is used for specifying type and region polymorphic functions. The region unification algorithm will fail to unify two regions with different types (except if one of the region types is RTY_BOT), which provides the guarantee that values stored in a region at run time are classified according to the region type of the region. For efficiency, the region type for a region is stored both in the generation descriptor for the old generation and in the generation descriptor for the young generation.

Values stored in finite regions on the stack are traversed by the garbage collector, but never copied or collected.

2.2 The GC Algorithm

The GC algorithm makes use of a series of auxiliary utility functions:

- `in_oldgen_and_minor(p)`: Returns TRUE if the collection is a minor collection and `p` points to an object in a region page for which the old-generation bit is set. Returns FALSE otherwise.
- `is_int(p)`: Returns TRUE if the least-significant bit in `p` is set. Returns FALSE otherwise.
- `tag_is_fwd_ptr(w)`: Returns TRUE if the tag word `w` is the reserved forward pointer tag, which is different from other tags used for tagged objects. Returns FALSE otherwise.
- `is_pairregion(r)`: Returns TRUE if the runtime type associated with the region descriptor `r` is REGION_PAIR. Returns FALSE otherwise.
- `in_tospace(p)`: Returns TRUE if `p` points to an object in a region page for which the to-space bit is set. Returns FALSE otherwise.
- `acopy_pair(r,p)`: Allocates a pair in the region associated with the region descriptor `r` and copies into the newly allocated memory the two pointers (or integers) contained in the pair pointed to by `p`.
- `obj_sz(w)`: Returns the size of the object in words, given its tag word.
- `gendesc(p)`: Returns the generation descriptor for the generation in which the object pointed to by `p` resides. Each region page in the generation has associated with it a generation pointer, pointing at the generation descriptor for the generation. Generation pointers are installed when a new region page is associated with a generation.
- `push_scanstack(a)`: Pushes the allocation pointer `a` onto the scan stack.
- `pop_scanstack()`: Pops and returns the top scan pointer from the scan stack. Returns NULL if the scan stack is empty.
- `target_gen(g,p)`: Returns the old generation associated with `g`'s region unless `g` is a young generation and `p` appears in a white area in `g`, in which case it returns `g`.

A central part of the GC algorithm is the function `evacuate`, shown in Fig. 5, which copies live values under consideration from from-space into to-space. It takes a pointer `p` and copies the value pointed to into to-space provided it is not already copied and that it is a prospect (i.e., under a minor collection, values in old generations are not copied.) For brevity, only pairs are treated specially; the implementation also treats regions of type RTY_TRIPLE and RTY_REF specially, as also triples and references are represented unboxed.

```
void* evacuate(void* p) {
  if ( is_int(p) ||
       in_oldgen_and_minor(p) )
    return p;
  g = gendesc(p);
  g_t = target_gen(g,p);
  if ( is_pairregion(g) ) {
    if ( in_tospace(*(p+1)) )
      return *(p+1); // fwd-ptr
    a = acopy_pair(g_t,p);
    *(p+1) = a;       // set fwd-ptr
  } else {
    if ( tag_is_fwd_ptr(*p) )
      return *p;
    a = acopy(g_t,p);
    *p = a;           // set fwd-ptr
  }
  if ( g_t->status == NONE ) {
    g_t->status = SOME;
    push_scanstack(a);
  }
  return a;
}
```

Fig. 5. The function `evacuate` assumes that the argument `p` points to an object and that it perhaps resides in from-space and needs to be copied to to-space. After copying, a forward-pointer is installed.

```
void cheney(void* s) {
  g = gendesc(s);
  if ( is_pairregion(g) ) {
    while ( s+1 != g->a ) {
      *(s+1) = evacuate(*(s+1));
      *(s+2) = evacuate(*(s+2));
      s = next_pair(s,g);
    }
  } else {
    while ( s != g->a ) {
      for ( i=1; i<obj_sz(*s); i++ )
        *(s+i) = evacuate(*(s+i));
      s = next_value(s,g);
    }
  }
  g->status = NONE;
}
```

Fig. 6. The function `cheney` assumes that the argument scan pointer `s` points to a value that has already been copied to to-space but for which the components have not yet been evacuated. The function is named `cheney` because it degenerates to Cheney's algorithm if multi-generations are disabled.

Another central function is the `cheney` function, which takes care of scanning the values that have been copied into to-space. During scanning, the `cheney` function may call `evacuate` on values that have themselves not yet been copied, which may cause an update to the generation allocation pointer. Once, for all regions, the scan-pointer reaches the allocation pointer, the collection terminates. The `cheney` function is shown in Fig. 6. Notice, again, that special treatment is required for dealing with untagged values (only the case for pairs is shown.)

The main GC function, called `gc` is shown in Fig. 7. It evacuates all values in the root set and continues by calling the `cheney` function on all values on the scan stack. Notice that the `evacuate` function pushes values that have been copied to to-space onto the scan stack for further processing (the g_t`->status` field is used to ensure that the scan pointer is pushed at most once.)

```
void gc(void** rootset) {
  while ( p = next_root(rootset) ) *p = evacuate(*p);
  while ( p = pop_scanstack() ) cheney(p);
}
```

Fig. 7. The main GC function evacuates each of the values in the root set after which the `cheney` function is called with scan pointers from the scan stack as long as there are scan pointers on the stack.

To determines whether a minor or a major collection is run, a so-called *heap-to-live ratio* is maintained, which by default is set to 3.0. Whenever the size of the free-list of pages becomes less than 1/3 of the total region heap, garbage collection is initiated upon the next function entry (i.e., safe point). After each collection, it is ensured that the number of allocated region pages is at least 3.0 times the size of to-space (given the heap-to-live ratio is 3.0). The following rules are deployed for switching between major and minor collections, allowing an arbitrary number of minor collections between two major collections:

1. If the current collection is a major collection, the next collection will be a minor collection. The region heap is enlarged to satisfy the heap-to-live ratio.
2. If the current collection is a minor collection and the heap-to-live ratio is not satisfied after the collection, the next collection will be a major collection.

2.3 Mutable Objects and Large Objects

In the presence of mutable objects, the generation upward closure invariant may be violated during program evaluation. In particular, a reference cell (which are rare in a functional language) residing in an old generation, may be assigned to point at a value residing in a young generation. We refine the generation upward-closure condition as follows:

Definition 3 (Refined Generation Upward-Closure). *For all values v, if v is non-mutable and resides in an old generation then for all values v' pointed to from v, v' resides in an old generation.*

The refined generation upward-closure invariant is safe, if each minor collection traverses all reachable mutable values (even those that reside in old generations). For minor collections we extend the root set to contain, not only live values on the stack, but also all references and tables allocated. How does the collector locate all references and tables? Simply by arranging that such values are stored in regions with distinguished region types. During a minor collection, the region stack is traversed and objects in regions of type RTY_REF and RTY_ARRAY are traversed. Thus, we avoid the implementation of the usual "remembered set" of mutable values that have been updated since the previous collection. This strategy can potentially be more costly than if a proper "remembered set" is maintained, which we leave to future work.

Concerning the treatment of large objects, there are several options. In the implementation, we are currently treating large objects without associating with being either young or old. Large objects are kept in one list associated with a region descriptor. Following this strategy, large objects are not associated with a particular generation (nor need they be associated with a color) and may therefore only be deleted during major collections. However, large objects should be traversed (not copied), when reached, both during major and minor collections.

3 Experimental Results

In this section, we describe a series of experiments that serve to demonstrate the relationship between region inference, non-generational garbage collection, and the generational garbage collection algorithm presented in Sect. 2.

The experiments are performed with MLKit version 4.4.1 and Mlton v20180207. MLKit version 4.4.1 generates native x64 machine code, which is also the case for Mlton v20180207. The two compilers are very different. Whereas Mlton is a whole-program highly-optimising compiler, MLKit features a smart-recompilation system that allows for quick rebuilds upon modification of source code.

All benchmark programs are executed on a MacBook Pro (15-inch, 2016) with a 2.7GHz Intel Core i7 processor and 16GB of memory running macOS. Times reported are wall clock times and memory usage is measured using the macOS `/usr/bin/time` program. Measurements are averages over 10 runs. We use m to specify memory usage (resident set size) and t to specify wall clock execution time (in seconds). Subscripts describe the mode of the compiler, with $*_{\mathrm{r}}$ signifying region inference enabled, $*_{\mathrm{g}}$ signifying garbage collection enabled, and $*_{\mathrm{G}}$ signifying generational garbage collection enabled. Thus, t_{rG} specifies wall clock execution time with region inference and generational garbage collection enabled. We use m_{mlton} and t_{mlton} to signify memory usage and wall clock execution time for executables running code generated by Mlton. The benchmark programs span from micro-benchmarks such as `fib37` and `tak` (7 and 12 lines), which only use the runtime stack for allocation, to larger programs, such as `vliw` and `mlyacc` (3676 and 7353 lines), that solve real-world problems. The program `msort-rf` has been made region-friendly by the programmer.

By *disabling* region inference, we mean instructing region inference to allocate all values that would be allocated in infinite regions in global regions (collapsed according to their region type). Then not a single infinite region is deallocated at run time and the non-generational garbage collection algorithm essentially reduces to Cheney's algorithm. Disabling region inference does not change the property that many values are allocated in finite regions on the stack.

3.1 Comparison with Mlton

In this section, we present base numbers for running the benchmark programs using the MLKit compiler with region inference and non-generational garbage collection enabled. Fig. 8 shows wall clock time for MLKit generated executables

relative to wall clock time for Mlton (version v20180207) generated executables. We see that for some of the programs, Mlton outperforms the MLKit (with and without garbage collection enabled). Mlton's whole-program compilation strategy, efficient IO-operations, and optimised instruction selection for the x64 architecture, are good candidates for an explanation. Raw numbers for the configurations are shown in Fig. 9, which also shows memory usage for the different configurations. Even though the performance of all but one benchmark is better with region inference alone, for some of the benchmark programs (i.e., those with numbers marked in bold in Fig. 9), region inference alone does not suffice to obtain good memory performance.

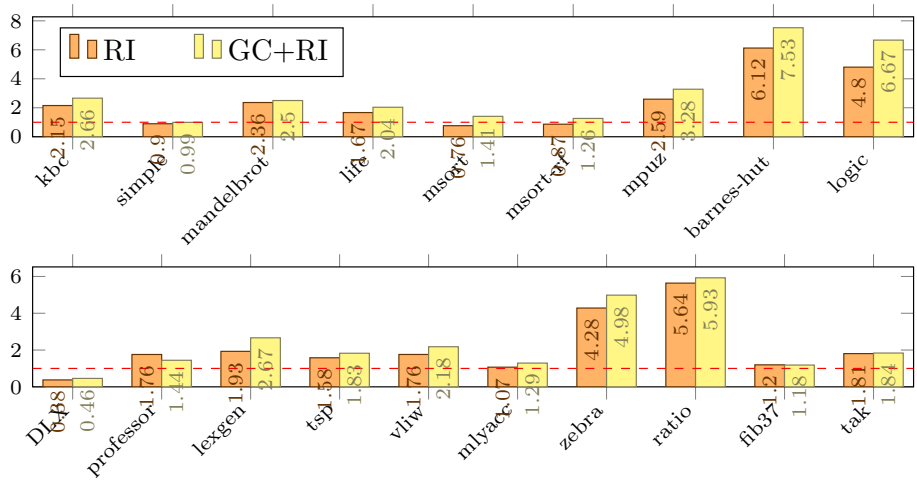

Fig. 8. Wall clock execution times for MLKit generated executables relative to execution times for Mlton generated executables (the dashed red base line). The orange (left) bars show measurements for MLKit with only region inference enabled. The yellow (right) bars show measurements for when both region inference and GC is enabled.

3.2 Generational Garbage Collection

Measurements showing the effect of non-generational and generational garbage collection in concert with region inference is shown in Fig. 10. First, notice that region inference has a positive influence or no effect on performance in all but one of the benchmarks, namely the Knuth-Bendix completion program, for which region inference adds an excessive number of region parameters to the main mutually recursive functions (explaining the slowdown). Second, generational garbage collection alone (without region inference) performs better than or equivalent to (in all but one case) non-generational garbage collection (the red line). Finally, for six or seven of the benchmarks, the combination of region inference and generational garbage collection performs better than the combination

Program	t_{mlton}	t_{r}	t_{rg}	m_{mlton}	m_{r}	m_{rg}
`kbc`	0.10	0.22	0.28	2.5M	**6.9M**	3.4M
`simple`	0.26	0.24	0.26	6.4M	2.6M	3.4M
`mandelbrot`	0.09	0.22	0.24	978K	1.4M	1.6M
`life`	0.54	0.91	1.11	2.6M	**14M**	1.6M
`msort`	1.09	0.83	1.53	427M	**410M**	137M
`msort-rf`	0.81	0.70	1.03	652M	102M	124M
`mpuz`	0.34	0.88	1.11	950K	1.2M	1.3M
`barnes-hut`	0.14	0.85	1.05	2.2M	**284M**	2.4M
`logic`	0.11	0.54	0.75	2.4M	**276M**	2.4M
`DLX`	0.51	0.19	0.23	33M	6.7M	6.9M
`professor`	0.37	0.66	0.54	1.6M	**10M**	1.4M
`lexgen`	0.21	0.41	0.57	18M	**50M**	8.1M
`tsp`	0.14	0.22	0.25	11M	8.3M	13M
`vliw`	0.05	0.09	0.11	8.4M	**9.7M**	4.6M
`mlyacc`	0.19	0.20	0.24	7.0M	**66M**	6.6M
`zebra`	0.51	2.18	2.54	1.6M	**132M**	1.3M
`ratio`	0.35	1.98	2.08	50M	**38M**	10M
`fib37`	0.32	0.38	0.38	937K	1.1M	1.1M
`tak`	0.68	1.23	1.26	938K	1.1M	1.1M

Fig. 9. Wall clock execution times and maximum resident memory usage for Mlton generated executables and for MLKit generated executables with only region inference enabled and with both region inference and non-generational GC enabled (averages of 10 runs). Numbers in bold highlight benchmarks for which region inference alone does not suffice to obtain good memory behavior.

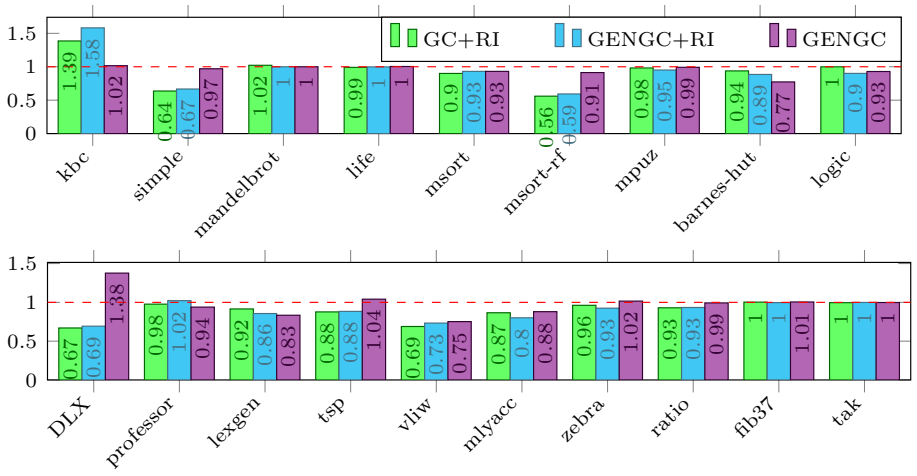

Fig. 10. Wall clock execution times for different configurations of MLKit generated executables relative to execution times for executables with only non-generational GC enabled (the red dashed base line). The green (left) bars show measurements for a configuration with region inference and non-generational GC. The blue (middle) bars show measurements for when both region inference and generational GC is enabled. The violet (right) bars show measurements for when only generational GC is enabled.

of non-generational garbage collection and region inference. The results are arguably quite sensitive to the heap-to-live ratio (a fair comparison should perhaps allow the combination of generational garbage collection and region inference to work with a higher heap-to-live ratio).

Fig. 11 shows the garbage collection counts (c_{rg}, c_{rG}, c_{g}, and c_{G}) for the different configurations. Notice that the garbage collection counts (and times) are smaller when region inference is enabled. Notice also that the percentage of memory reclaimed by the garbage collector is (close to) invariant to whether the garbage collector is generational or not.

The MLKit features a *region profiling tool* [16], which allows for showing a program's use of regions over time. Fig. 12 shows region profiles of MLYacc computations for four different MLKit runtime configurations. The profiles show that generational garbage collection combined with region inference often requires more memory than when region inference is combined with non-generational garbage collection, but also, that the profile obtained alone with generational garbage collection is similar to the profile obtained with region inference and non-generational garbage collection enabled. The figure also demonstrates a crucial point, namely that the global regions are often those that needs to be collected by the reference tracing collector, which means that schemes that attempt at collecting only the top-most regions will probably fail to be effective.

3.3 Memory Waste

Region inference combined with generational garbage collection results in more memory waste (unused memory in region pages) than when combined with non-generational garbage collection (up to 17 percentage points more). The reason is that, with generational garbage collection, each infinite region contains two lists of region pages (one list for each generation), each of which may not be fully utilised. Fig. 13 gives memory waste percentages for the configurations w_{rg} (region inference and non-generational garbage collection), w_{rG} (region inference and generational garbage collection), w_{g} (non-generational garbage collection), and w_{G} (generational garbage collection). As expected, the waste is high for the region inference configurations. We also see that generational garbage collection combined with region inference gives rise to the highest degree of waste.

4 Related Work

Most related to this work is the previous work on combining region inference and garbage collection in the MLKit [17]. Compared to the earlier work, the present work investigates how generational garbage collection can be combined with region inference and how the concept of typed regions can be used to implement a generation write barrier. There is a large body of related work concerning general garbage collection techniques [19] and garbage collection techniques for functional languages, including [7, 18, 23, 29].

Program	c_{rg}	g_{rg}(ms)	p_{rg}	c_{rG}		g_{rG}(ms)		p_{rG}	c_{g}	g_{g}(ms)	c_{G}		g_{G}(ms)	
`kbc`	40	5.9	43%	32	(11)	4.9	(1.8)	42%	204	12.6	245	(35)	16.3	(2.2)
`simple`	7	2.8	7%	10	(5)	3.3	(1.8)	4%	17	8.7	23	(11)	8.7	(5.4)
`mandelbrot`	1	0.1	0%	1	(0)	0.2	(0.0)	0%	1	0.1	1	(0)	0.1	(0.0)
`life`	142	6.4	17%	121	(8)	5.4	(0.5)	16%	677	25.4	880	(139)	28.3	(6.8)
`msort`	33	718.7	53%	47	(23)	762.7	(491.3)	59%	41	1001.1	55	(26)	894.3	(584.0
`msort-rf`	23	179.3	6%	35	(17)	270.5	(161.6)	8%	42	1050.2	56	(27)	936.9	(640.9
`mpuz`	2	0.3	2%	2	(1)	0.1	(0.0)	2%	2	0.2	2	(1)	0.2	(0.0)
`barnes-hut`	1948	315.3	63%	1545	(414)	214.0	(79.4)	63%	4272	638.3	4243	(870)	400.4	(143.8)
`logic`	2276	350.2	100%	2571	(348)	289.7	(55.1)	100%	2306	367.2	2478	(316)	299.3	(54.0)
`DLX`	5	2.3	0%	6	(3)	3.9	(1.6)	0%	104	119.5	204	(102)	245.8	(109.5)
`professor`	1218	20.7	27%	1065	(14)	16.0	(0.3)	27%	9821	148.9	8503	(103)	94.3	(2.6)
`lexgen`	254	148.1	80%	241	(41)	96.4	(25.3)	79%	451	251.6	479	(80)	155.7	(47.1)
`tsp`	12	16.5	5%	17	(8)	21.3	(7.7)	6%	19	60.5	29	(14)	79.0	(40.9)
`vliw`	23	8.3	13%	24	(10)	7.5	(3.3)	13%	214	77.6	221	(40)	42.0	(14.4)
`mlyacc`	115	61.8	69%	88	(18)	36.3	(9.6)	61%	211	104.5	239	(67)	75.0	(35.3)
`zebra`	5008	90.6	57%	3010	(337)	65.4	(8.4)	56%	17357	274.4	23023	(1001)	310.2	(19.1)
`ratio`	34	49.2	23%	37	(8)	62.5	(13.8)	22%	110	194.4	99	(13)	180.1	(17.7)
`fib37`	1	0.1	0%	1	(0)	0.1	(0.0)	0%	1	0.1	1	(0)	0.1	(0.0)
`tak`	1	0.1	0%	1	(0)	0.1	(0.0)	0%	1	0.1	1	(0)	0.1	(0.0)

Fig. 11. GC counts (c_*) and GC times (g_*) for the different configurations. Reported counts are the total number of collections with the number of major collections and the accumulated major collection time in parentheses. The p_* columns show the percentage of bytes reclaimed by GC (in contrast to region inference).

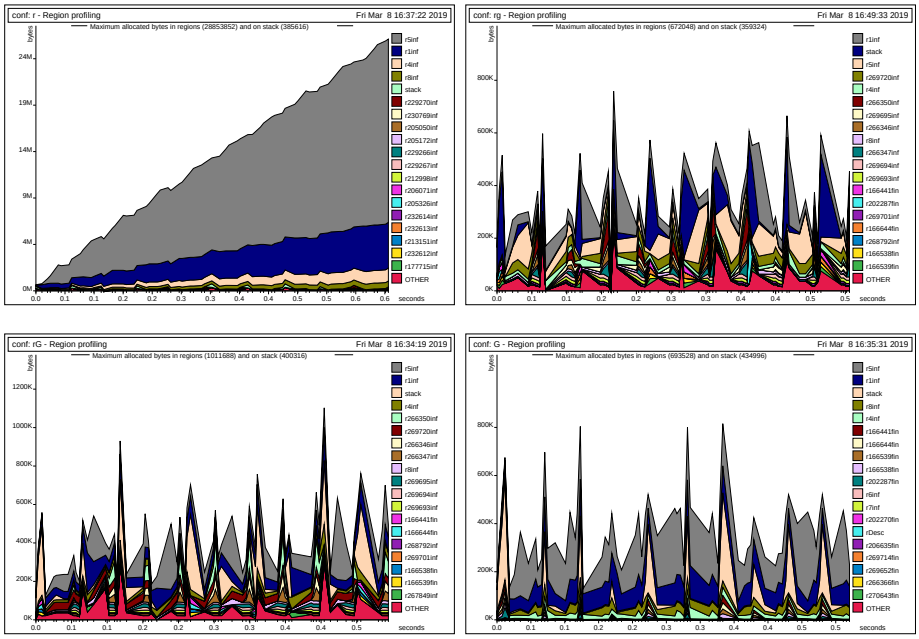

Fig. 12. The top-left MLYacc region profile shows the memory usage over time for a runtime configuration with region inference enabled and reference tracing GC disabled (denoted `r`). The top-right region profile shows the memory usage for a configuration where non-generational GC is combined with region inference (denoted `rg`). The configuration for the bottom-left region profile combines generational GC with region inference (denoted `rG`) whereas the configuration for the bottom-right region profile is using generational GC only (denoted `G`).

Program	w_{rg} (%)	w_{rG} (%)	w_{g} (%)	w_{G} (%)
`kbc`	42	57	4	8
`simple`	13	30	2	6
`mandelbrot`	0	0	0	0
`life`	8	17	4	9
`msort`	2	5	2	4
`msort-rf`	3	6	2	4
`mpuz`	69	82	47	65
`barnes-hut`	10	18	2	5
`logic`	3	6	3	6
`DLX`	23	32	1	2

Program	w_{rg} (%)	w_{rG} (%)	w_{g} (%)	w_{G} (%)
`professor`	25	38	10	17
`lexgen`	10	18	1	2
`tsp`	7	12	5	7
`vliw`	13	28	1	3
`mlyacc`	8	21	1	2
`zebra`	31	38	10	22
`ratio`	5	7	1	2
`fib37`	0	0	0	0
`tak`	0	0	0	0

Fig. 13. Memory waste. The numbers show the average percentage of region waste (unused memory in region pages) measured at each collection.

Incremental, concurrent, and real-time garbage collection techniques for functional languages have recently obtained much attention. In particular, the presence of generations has been shown useful for collecting parts of the heap incrementally and in a concurrent and parallel fashion [22, 21, 3]. We leave it to future work to investigate the use of regions and generations in the MLKit for supporting concurrency and parallelism in the language.

A particular body of related work investigates the notion of escape analysis for improving stack allocation in garbage collected systems [5, 24]. Region inference and MLKit's polymorphic multiplicity analysis [4] allow more objects to be stack allocated than traditional escape analyses, which allows only local, non-escaping values to be stack allocated. Other work investigates the use of static prediction techniques and linear typing for inferring heap space usage [20].

Cyclone [26] is a region-based type-safe C dialect, for which, the programmer can decide if an object should reside in the GC heap or in a region. Another region-based language is Gay and Aiken's RC system, which features limited explicit regions for C, combined with reference counting of regions [15]. A modern language for system programming is Rust, which is based on ownership types for controlling the use of resources, including memory [2]. Ownership types are also used for real-time implementations of Java [6]. None of the above systems are combined with techniques for automatic generational garbage collection.

Also related to the present work is the work by Aiken et al. [1], who show how region inference may be improved for some programs by removing the constraints of the stack discipline, which may cause a garbage collector to run less often. Region inference has also been used in practical settings without combining it with reference-tracing garbage collection. In particular, it has been used as the primary memory management scheme for a web server [10, 11, 13, 14].

5 Conclusion and Future Work

We have presented a technique for combining region inference and generational garbage collection in a functional language. Whereas generational collection by itself is shown (in most cases) to be beneficial compared to a simple Cheney-style non-generational collector, when generational collection is combined with region inference, it turns out that region inference will take care of reclaiming much of the memory that generational garbage collection would otherwise reclaim. There are, however, potential benefits of a generational collector, which, in a few cases, also leads to improved performance. For a more detailed description of the implementation, consult the companion technical report [12].

As a first obvious candidate for future work, the x64 code generator can be improved to generate more efficient code. Second, for making the technique useful for applications that make heavy use of mutable objects, a proper implementation of a "remembered set" would be an appropriate next step. Finally, an obvious candidate for future work is to investigate the possibility of combining region inference and, perhaps, generations, with features for concurrency and parallelism.

References

1. Aiken, A., Fähndrich, M., Levien, R.: Better static memory management: Improving region-based analysis of higher-order languages. In: ACM Conference on Programming Languages and Implementation. PLDI '95 (June 1995)
2. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: ACM Conference on Object-oriented Programming, Systems, Languages, and Applications. OOPSLA '02 (2002)
3. Anderson, T.A.: Optimizations in a private nursery-based garbage collector. In: ACM International Symposium on Memory Management. ISMM '10 (2010)
4. Birkedal, L., Tofte, M., Vejlstrup, M.: From region inference to von Neumann machines via region representation inference. In: ACM Symposium on Principles of Programming Languages. POPL '96 (January 1996)
5. Blanchet, B.: Escape analysis : Correctness proof, implementation and experimental results. In: ACM Symposium on Principles of Programming Languages (POPL'98). pp. 25–37. ACM Press (January 1998)
6. Boyapati, C., Salcianu, A., Beebee, Jr., W., Rinard, M.: Ownership types for safe region-based memory management in real-time java. In: ACM Conference on Programming Language Design and Implementation. PLDI '03 (2003)
7. Doligez, D., Leroy, X.: A concurrent, generational garbage collector for a multithreaded implementation of ml. In: ACM Symposium on Principles of Programming Languages. POPL '93 (1993)
8. Elsman, M.: Garbage collection safety for region-based memory management. In: ACM Workshop on Types in Language Design and Implementation. TLDI '03 (January 2003)
9. Elsman, M., Hallenberg, N.: An optimizing backend for the ML Kit using a stack of regions. Student Project 95-7-8, University of Copenhagen (DIKU) (July 5 1995)
10. Elsman, M., Hallenberg, N.: SMLserver—A Functional Approach to Web Publishing. The IT University of Copenhagen (2002), (154 pages). Available via `http://www.smlserver.org`
11. Elsman, M., Hallenberg, N.: Web programming with SMLserver. In: International Symposium on Practical Aspects of Declarative Languages (PADL'03). Springer-Verlag (January 2003)
12. Elsman, M., Hallenberg, N.: Combining region inference and generational garbage collection. Tech. Rep. 2019/01, Department of Computer Science, University of Copenhagen (DIKU) (November 2019), ISSN 0107-8283
13. Elsman, M., Larsen, K.F.: Typing XHTML Web applications in ML. In: International Symposium on Practical Aspects of Declarative Languages (PADL'04). Springer-Verlag (June 2004)
14. Elsman, M., Munksgaard, P., Larsen, K.F.: Experience report: Type-safe multitier programming with Standard ML modules. In: Proceedings of the ML Family Workshop. ML '18 (September 2018)
15. Gay, D., Aiken, A.: Language support for regions. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01). ACM Press, Snowbird, Utah (June 2001)
16. Hallenberg, N.: A region profiler for a Standard ML compiler based on region inference. Student Project 96-5-7, Department of Computer Science, University of Copenhagen (DIKU) (June 1996)
17. Hallenberg, N., Elsman, M., Tofte, M.: Combining region inference and garbage collection. In: ACM Conference on Programming Language Design and Implementation (PLDI'02). ACM Press (June 2002), berlin, Germany

18. Huelsbergen, L., Winterbottom, P.: Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. In: ACM International Symposium on Memory Management. ISMM '98 (1998)
19. Jones, R., Hosking, A., Moss, E.: The Garbage Collection Handbook: The Art of Automatic Memory Management. Chapman & Hall/CRC (2011)
20. Jost, S., Hammond, K., Loidl, H.W., Hofmann, M.: Static determination of quantitative resource usage for higher-order programs. In: ACM Symposium on Principles of Programming Languages. POPL '10 (2010)
21. Marlow, S., Peyton Jones, S.: Multicore garbage collection with local heaps. In: ACM International Symposium on Memory Management. ISMM '11 (2011)
22. Marlow, S., Peyton Jones, S., Singh, S.: Runtime support for multicore haskell. In: ACM International Conference on Functional Programming. ICFP '09 (2009)
23. Reppy, J.H.: A high-performance garbage collector for Standard ML. Tech. rep., AT&T Bell Laboratories (January 1994)
24. Salagnac, G., Yovine, S., Garbervetsky, D.: Fast escape analysis for region-based memory management. Electron. Notes Th. C. S. **131**, 99–110 (May 2005)
25. Salagnac, G., Nakhli, C., Rippert, C., Yovine, S.: Efficient Region-Based Memory Management for Resource-limited Real-Time Embedded Systems. In: Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (Jul 2006)
26. Swamy, N., Hicks, M., Morrisett, G., Grossman, D., Jim, T.: Safe manual memory management in cyclone. Sci. Comput. Program. **62**(2), 122–144 (Oct 2006)
27. Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N.: A retrospective on region-based memory management. Higher-Order and Symbolic Computation **17**(3), 245–265 (Sep 2004)
28. Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N., Olesen, T.H., Sestoft, P.: Programming with regions in the MLKit (revised for version 4.3.0). Tech. rep., IT University of Copenhagen, Denmark (January 2006)
29. Ueno, K., Ohori, A.: A fully concurrent garbage collector for functional programs on multicore processors. In: ACM International Conference on Functional Programming. ICFP '16 (2016)