Combining Region Inference and Garbage Collection

Niels Hallenberg
nh@itu.dk

Martin Elsman*
mael@dina.kvl.dk

Mads Tofte
tofte@itu.dk

IT University of Copenhagen
Glentevej 67, DK-2400 Copenhagen NV, Denmark

ABSTRACT

This paper describes a memory discipline that combines region-based memory management and copying garbage collection by extending Cheney's copying garbage collection algorithm to work with regions. The paper presents empirical evidence that region inference very significantly reduces the number of garbage collections; and evidence that the fastest execution is obtained by using regions alone, without garbage collection.

The memory discipline is implemented for Standard ML in the ML Kit compiler and measurements show that for a variety of benchmark programs, code generated by the compiler is as efficient, both with respect to execution time and memory usage, as programs compiled with Standard ML of New Jersey, another state-of-the-art Standard ML compiler.

Categories and Subject Descriptors

D.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.3 [**Programming Languages**]: Language Constructs and Features—*Dynamic storage management*; F.3 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Algorithms, Languages

Keywords

Garbage collection, region inference, Standard ML

1. INTRODUCTION

This paper presents a memory discipline that integrates region-based memory management and automatic heap management ("garbage collection").

*Part time at Royal Veterinary and Agricultural University of Denmark.

PLDI'02, June 17-19, 2002, Berlin, Germany.

In the rest of this section we describe the two strategies and summarise the differences between the two. The main contributions are summarised in Section 1.2.

1.1 Background

A popular memory management discipline for block structured languages is *stack allocation*. Every allocation point is matched by a deallocation point and these points are easily identified in the program. Allocation and deallocation take place at procedure entry and exit, respectively. This strategy often leads to fast and compact use of memory.

The main limitation of the stack discipline is that for some algorithms, lifetimes of data simply are not nested. Some other form of recycling is needed in such cases. In *heap allocation*, values that do not fit in the stack discipline are allocted in the heap, which is a part of the store separate from the stack.

The most basic form of heap allocation is *manual heap allocation*, in which the programmer is in charge of allocating and deallocating values. The technique is notoriously difficult to use in practice: it is easy to allocate memory, but hard to know when to free it. Freeing memory too soon may lead the program to crash ("dangling pointers") while freeing memory too late may lead to wasteful use of memory ("memory leaks").

Automatic heap management addresses these problems by leaving deallocation of memory to a part of the runtime system, the *garbage collector*. From time to time, the garbage collector interrupts the user's program and recycles the parts of memory that are not needed for the remainder of the computation (i.e., the "garbage"). Most implementations of functional languages and some implementations of object-oriented languages use automatic heap management. Some even use automatic heap management and no stack at all [2].

However, automatic heap management is not perfect either. The separation of allocation and deallocation makes it hard for the programmer to know how long values will live and therefore how much memory the program will use. Garbage collection can account for a high percentage of the running time, whereas deallocation in the stack discipline is very inexpensive.

There is a large body of work concerning garbage collection techniques, see for example [23, 14]. These techniques share the following features:

- Automation. Garbage is reclaimed automatically at runtime, by the garbage collector.
- Lifetimes are determined at runtime. The garbage collector traverses values in order to locate garbage.

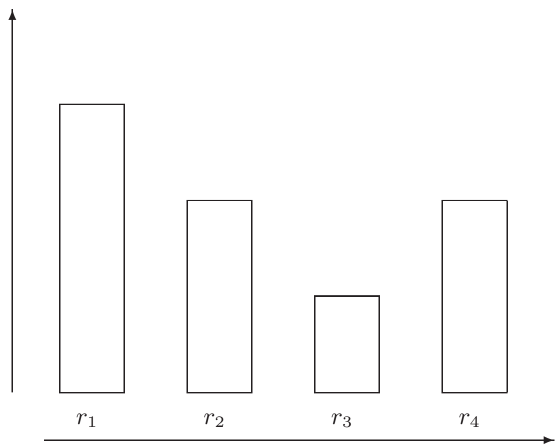

Figure 1: A stack of regions. A region is a rectangle in the picture. The stack grows to the right, i.e., r_4 is the topmost region and is the first region to be deallocated. Each region may grow upwards.

- Uniformity. The garbage collector uses a fixed strategy for all user programs, although some garbage collection algorithms rely on heuristics about common memory behaviour of programs (e.g., [15]).

Region-based memory management [20, 3, 21, 17] attempts to achieve both the predictability and efficiency of the stack discipline and the flexibility and safety of automatic heap management. Conceptually, the store is organised as a *stack of regions*, see Figure 1. Allocation and deallocation directives of regions are inserted into the program at compile time, based on a program analysis called *region inference* [20, 3]. Value-creating expressions are annotated with information that directs in what region values go at runtime. Moreover, if e is some region-annotated expression, then so is

`letregion r in` e `end`

Expressions of this form are evaluated as follows. First a region is allocated on top of the stack and bound to the region variable `r`. Then e is evaluated, possibly using the region bound to `r` for holding values. Finally, upon reaching `end`, the region is reclaimed. A particularly important aspect of region inference is the notion of *region polymorphism*, which allows regions to be passed to functions at runtime.

Region-based memory management provides the following properties:

- Automation. Garbage is reclaimed automatically at runtime, by popping the region stack.

- Lifetimes are decided at compile time. Region inference and other static analyses decide lifetimes at compile time [20, 3]. There is *no* tagging (for pointer traversal) and *no* runtime traversal of values in order to free memory.

- Specialization. Region inference specializes memory management to the user's program.

Region-based memory management is implemented for Standard ML in the ML Kit [18], a region-based compiler for all of SML'97, including Modules and the Standard ML Basis Library. To date, the largest Standard ML programs compiled under the region scheme are AnnoDomini, a 60,000 lines program and the ML Kit itself, a 90,000 lines program.

1.2 Contributions of the Paper

So far, there has been no direct way of comparing region-based memory management with garbage collection. Part of the reason is theoretical. It is known that region-based memory management and garbage collection are incomparable, in that there are programs that use arbitrary more space in one scheme than in the other. But even experimental comparisons are difficult: a Standard ML compiler is a complex piece of software and differences in performance in code produced by different compilers may stem from many other factors than differences between regions and garbage collection.

At the same time, practical experimentation with region-based memory management has suggested that although some rewriting of source programs is often necessary in order to get good memory behaviour, often very little rewriting is required, even for large programs. (In the case of AnnoDomini, a reasonably good execution was obtained after modifying 10 lines out of 60,000 [7].) This suggests that "most of the time," region inference estimates lifetimes correctly.

The purpose of this paper is to investigate the relationship between region inference and garbage collection more closely. In particular, can a combination of garbage collection and region inference reduce the need for tuning programs? Can such a combination give practical results that are as good as the ones one obtain with tuned programs using regions alone?

Conversely, from the point of view of garbage collection: is region inference an efficient way of reducing the amount of garbage collection required?

In order to answer these questions, we have designed a new back-end and runtime system for the ML Kit that allows one to compile and run programs in different modes, including:

1. Using regions alone, with values untagged and support for dangling pointers (in a pure region based system where values are not traversed by a garbage collector, no tags are needed to distinguish pointers from non-pointers)

2. Using regions alone, but with values tagged; this mode makes it possible to isolate the effect of tagging on performance

3. Using a copying garbage collector within a degenerate region stack consisting of one region only

4. Using a combination of regions and the copying garbage collector

The remainder of the paper is organised as follows. Section 2 describes the new runtime system for integrating region inference and garbage collection based on Cheney's copying garbage collection algorithm. An overview of the ML Kit implementation is given in Section 3. Section 4 presents evidence that region inference very significantly reduces the number of garbage collections; evidence that the fastest execution is obtained by using regions alone, without garbage collection; and empirical evidence that the combination of region inference and garbage collection is comparable to Standard ML of New Jersey (another state-of-the-art Standard ML compiler) both regarding time and memory usage. Finally, in Sections 5 and 7 we describe related work and conclude.

2. GARBAGE COLLECTING REGIONS

Region inference imposes a restriction on how garbage collection can work, namely, if two values belong to the same region before the collection and both survive the collection then they must belong to the same region after the collection. (Otherwise, the popping of the region stack could become unsound.) The first design issue for a garbage collector then becomes whether one should try to garbage collect just one region at a time, or all regions. The former would allow, for example, that "global" regions (i.e., regions that are pushed onto the empty region stack when the program starts and not deallocated until the program terminates) are garbage collected separately. However, determining the set of pointers that point into a given region appears to be expensive, for, in principle, there can be pointers from any region into the region in question. Thus the algorithm that we propose collects all regions in every garbage collection.

To discuss the algorithm in more detail, we need to describe the physical representation of regions, initially without considering garbage collection.

2.1 Physical Representation of Regions

The store consists of a *stack* and, separate from the stack, a *region heap*. The stack consists of *activation records*. The region heap consists of a set of fixed-size *region pages*, some of which are linked together in a *free-list*. At runtime we distinguish between two kinds of regions [3]:

1. Regions inferred to hold only one value at a time. The size of the region is the maximal size of the values that may be allocated in the region. Those regions are called *finite regions* and are allocated in activation records on the stack. Finite regions usually contain tuples and closures.

2. Regions inferred to hold an unbounded number of values are called *infinite regions*. An infinite region is represented by a linked list of region pages, pointed to by a *region descriptor*, which resides in an activation record on the stack. Infinite regions usually contain lists and other recursive data structures.

A region descriptor is a triple (e, fp, a) of pointers, where e, the *end pointer*, points to the end of the most recently allocated page in the region; a, the *allocation pointer*, points to the first available free location in that page; and fp, the *first-page pointer*, points to the first page of the region [8]. Figure 2 shows an example runtime stack containing three region descriptors and one finite region.

Allocating a value is done at a if there is enough space in the region page; otherwise, the region is extended with a region page taken from the free-list.

An infinite region is allocated by requesting a region page from the free-list and updating a region descriptor. When an infinite region is popped, its region pages are appended to the free-list; this operation can be done in constant time

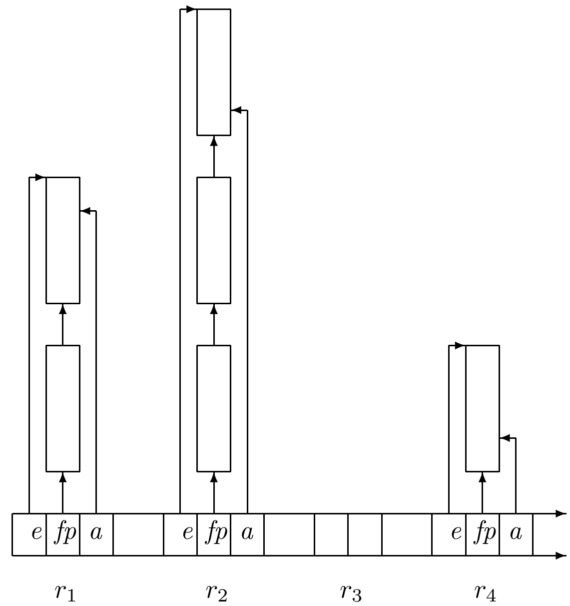

Figure 2: The runtime stack (bottom) contains three region descriptors and a finite region (r_3). Each region descriptor is a triple of pointers.

with the use of the first-page pointer *fp*, the end-pointer e, and knowledge about the size of a region page.

Values that fit in one word, such as integers and booleans, are implemented unboxed and therefore do not reside in distinguished regions. The size of an activation record is determined at compile time. An activation record is allocated on the runtime stack at entry to a function and popped on exit. Addresses of unboxed values, finite regions and region descriptors inside an activation record are statically determined offsets from the stack pointer. Thus, allocating a value in a finite region amounts to a store operation at a known offset from the stack pointer. No code is needed for allocating or deallocating a finite region at runtime, beyond the code for pushing and popping the enclosing activation record.

2.2 Cheney's Stop and Copy Algorithm

Our garbage collection algorithm is based on Cheney's stop and copy algorithm [5]. Cheney's algorithm has the advantage of being relatively straightforward. An argument against Cheney's algorithm might be that, unlike generational garbage collectors, it does not employ heuristics concerning lifetimes. But region inference already separates data according to lifetimes, so it is not clear that the garbage collector has to do so as well.

In this section we review Cheney's algorithm. In Section 2.3 we describe how we have modified it to deal with regions.

Cheney's algorithm uses two address spaces (*semi-spaces*), called the *from-space* and the *to-space*, respectively. The program allocates values into the from-space. Garbage collection is initiated when the from-space is full. The *root set* is the set of variables that are live in the activation records on the stack. The algorithm copies all the values reachable from the root set from the from-space into the to-space.

Thereafter the from-space and the to-space swap roles.

The Cheney algorithm is outlined below:

```
fun cheney(fromspace,tospace,s,a)=
let
  fun evacuate(p)=
    case !p of
      FORWARDED p' => p'
    | VALUE v =>
         let a0 = a
         in a:=a+size(v);
            tospace[a0..a-1]:= v;
            fromspace[p]:=
              FORWARDED a0;
            a0
         end

  fun scan(v)=
    (for each pointer p in v do
       p:= evacuate(p);
     s:= s + size(v))
in
   while  s < a do scan(!s)
end
```

The algorithm performs a breadth-first search and evacuates values from the from-space. It uses a *scan* pointer, s, and an *allocation* pointer, a, both of which point into the to-space; a points at the first free location in to-space and s points at the next value to be scanned. The invariant $s \leq a$ holds throughout. The part of the to-space between s and a serves as a queue for the breadth-first search. The collection is complete when we obtain $s = a$.

Every boxed representation of a value has a *tag-field* for implementing pointer traversal. After copying a value, the algorithm replaces the tag-field in the original copy of the value with a *forward pointer*, which points at the new copy of the value in to-space. Forward pointers can be distinguished from all other tags.

The function `scan` scans the value v pointed at by s and calls the function `evacuate` on all pointer fields inside v. We think of a value as a record of words, some of which are pointers. We assume a mechanism that can distinguish the pointers of a value from the non-pointers. Each pointer field is updated to contain the result of evacuating it. Finally, `scan` increases s, in preparation for scanning of the next unscanned value.

A call `evacuate`(p), where p is a pointer, examines the tag-field of the contents of p (i.e., `!`p). If the tag-field is a forward pointer then the forward pointer is returned. Otherwise `!`p is a value v; in this case we copy v into to-space (increasing a in the process) and replace the original copy of v by a forward pointer.

The algorithm assumes that there are no pointers from the heap to the stack. Initially, s and a point at the first address in to-space. We then apply `evacuate` on all pointer values in the root set and update the pointers in the root set with the new locations. At this point, the memory between s and a contains values evacuated from the root set. Then, `cheney`(*fromspace*,*tospace*, s, a) completes the collection.

2.3 Adapting Cheney's Algorithm for Regions

Cheney's algorithm can be extended to work with regions, as follows. Intuitively, each region is associated with a from-space and a to-space. A region descriptor is now a quadruple (e, fp, a, b) where a plays the dual role of being the allocation pointer for region inference and for the garbage collector and b is a so-called region status, which we have more to say about below. Scan pointers are kept in a scan stack; there is no scan pointer in the region descriptor. In the following, when r is some region descriptor, we use the notation $r \rightarrow a$ to refer to the allocation pointer in r; we use similar notation to access the other components of a region descriptor.

We now apply Cheney's algorithm locally on each region and use the stop criteria: $\forall r \in Reg : (r \rightarrow a) = s_r$, where Reg is the set of region descriptors on the stack and s_r is the scan pointer of r. The stop criteria is implemented using the *scan stack*, which consists of those scan pointers s_r for which $s_r \neq (r \rightarrow a)$.

The garbage collector never allocates into from-spaces. At the start of a garbage collection, the region stack is traversed and the region pages in the from-space areas (pointed at by $r \rightarrow fp$) are linked together to form a single global from-space area. Next, for every region descriptor r on the stack, $r \rightarrow fp$ is initialised to point at a fresh region page taken from the free-list. Moreover, $r \rightarrow a$ is initialised to point at the beginning of the page pointed to by $r \rightarrow fp$ and $r \rightarrow e$ at the end of the page pointed to by $r \rightarrow fp$. While collection is in progress, region pages are allocated from the free-list, which is disjoint from the global from-space area. After garbage collection, the global from-space area is appended to the free-list in a constant-time operation.

The Cheney algorithm extended to regions is outlined below:

```
fun evacuate(p)=
  case !p of
    FORWARDED p' => p'
  | VALUE v =>
      let val r = regiondesc(p)
          val a = alloc(r, v)
      in
        if r->b = NONE then
          (r->b:= SOME;
           push_onto_scanstack(a))
        else ();
        fromspace[p]:= FORWARDED a;
        a
      end

fun cheney(fromspace,r,s,a)=
  (while s <> a do
     let val v = !s
     in
       for each pointer p in v do
         p:= evacuate(p);
       s:= next_value(v,r)
     end;
   r->b:= NONE)

fun collect_regions()=
  while scanstack_not_empty() do
    let val s = pop_scanstack()
        val r = regiondesc(s)
    in cheney(fromspace, r, s, r->a)
    end
```

In a call `cheney`(*fromspace*, r, s, a), r is the address of a region descriptor of a region; it plays the role of the to-space. The `for`-loop scans the value v pointed at by s and calls the function `evacuate` on all pointer fields inside v. The call `next_value`(v, r) proceeds to the value after v in r (which may entail proceeding to the next region page in the region).

Next consider function `evacuate`. Given a pointer p to a value v, `regiondesc`(p) returns the address of the region descriptor of the region containing v (see Section 2.4). The function `alloc`(r, v) allocates v in the region described by r, returning the address of the new copy. (`alloc` extends the region with a new page, if necessary.) The field b in the region descriptor is a two-valued mark, called the *region status*; the field is NONE if $s_r = (r \rightarrow a)$ and SOME if $s_r \neq (r \rightarrow a)$. The field b is set to NONE in function `cheney` when all values has been scanned.

Because a region is composed of region pages, it is not always the case that $s_r \leq (r \rightarrow a)$ but $s_r = (r \rightarrow a)$ still signifies that the queue of unscanned values in the region is empty. When this happens, the region status of the region is changed to NONE.

The algorithm maintains the invariant that the region status $r \rightarrow b$ of some region descriptor r is SOME if and only if either s_r is on the scan stack or r denotes a region that is currently being scanned.[1]

Finally, `collect_regions` repeatedly calls `cheney` on one region at a time, till the scan stack is empty.

The maximal depth of the scan stack is limited by the number of region descriptors on the region stack; at any time, at most one pointer for each region is on the scan stack.

2.4 Region Page Descriptors

Every region page starts with a *region page descriptor*. It contains a pointer to the next region page in the region. It also contains an *origin pointer*, which points back to the region descriptor of the region.

As mentioned earlier, all region pages used by regions have the same fixed size. By instrumenting the compiler and the runtime system, it is possible to change the size of region pages to 2^n, where $1 \leq n \leq w$ and w is the number of bits per word (typically 32). Furthermore, the runtime system ensures that every region page starts on an address divisible by 2^n. Given the address p of value v, the region page descriptor of the page that contains v is found by computing the bitwise and of p and $\underbrace{1\cdots1}_{w-n}\underbrace{0\cdots0}_{n}$.

Thus `regiondesc`(p) can be computed by accessing the region page descriptor as described above and then extracting the origin pointer from it.

2.5 Finite Regions

So far, we have considered infinite regions only. Unfortunately, finite regions complicate matters, for there can be pointers from region pages into finite regions on the stack and indeed from a value in a finite region into another finite region. We therefore distinguish between three kinds of values:

1. Values allocated in infinite regions. These values are traversed and evacuated as described above.

2. Values allocated in finite regions residing in activation records on the stack. Such values are traversed and updated but are not moved.

3. Constants in the data area of the program binary. Because such values do not point at values in finite or infinite regions, these values are not traversed, not updated, and not copied.

To revise the algorithm to work with all three kinds of values, a separate *scan buffer* is used for finite regions. A value in a finite region is not moved and hence no forward pointer is stored after the value is traversed. To avoid that the algorithm traverses finite regions more than once, traversed values in finite regions are marked as constants by updating the value tags. After garbage collection, the constant-marks are removed and values obtain their original tags. The scan buffer is used both for holding values that remain to be scanned (similary to the scan stack for values in infinite regions) and for keeping track of traversed values in finite regions.

All what is needed is to revise the two functions `evacuate` and `collect_regions` shown below:

```
fun evacuate(p)=
  case !p of
    FORWARDED p' => p'
  | CONSTANT c => p
  | VALUE v =>
      if points_into_stack(p) then
        (p:= set_tag_const(!p);
         add_scan_buffer(p);
         p)
     else
       let val r = regiondesc(p)
           val a = alloc(r, v)
       in
         if r->b = NONE then
           (r->b:= SOME;
            push_onto_scanstack(a))
         else ();
         fromspace[p]:= FORWARDED a;
         a
       end

fun collect_regions()=
  (while scanstack_not_empty() ||
         scanbuffer_not_done() do
     (while scanbuffer_not_done() do
        let val s = get_scanbuffer()
        in
          for each pointer p in !s do
            p:= evacuate(p)
        end;
      while scanstack_not_empty() do
        let val s = pop_scanstack()
            val r = regiondesc(s)
        in cheney(fromspace, r, s, r->a)
        end);
   for each pointer p in scanbuffer do
     p:= remove_tag_const(!p))
```

[1] In the implementation, the region status occupies only one bit and is encoded in one of the pointer fields in the region descriptor.

Consider the function `evacuate`. Constant values are recognized by inspecting the tag-field of the value (see Section 2.2). The result of evacuating a pointer to a constant is the pointer itself. A pointer p pointing at a value v in a finite region is recognized by a range check on the stack boundaries (function `points_into_stack`). The value v has not yet been traversed; otherwise it would have been marked as a constant. The algorithm marks v as a constant and adds it to the scan buffer (i.e., we postpone the traversal of v).

The stop criteria in function `collect_regions` is now implemented using both the scan stack and the scan buffer. The function `get_scanbuffer` obtains the next un-scanned value v in scan buffer; the value v is not removed from the buffer. At the end of a collection, we remove all constant-marks (function `remove_tag_const`) on traversed values in finite regions. The maximal size of the scan buffer is limited by the number of finite regions on the stack.

2.6 Dangling Pointers

Region inference allows for both *shallow* and *deep* pointers, that is, pointers from older regions to newer regions and from newer regions to older regions. A shallow pointer may turn into a dangling pointer if the newer region is deallocated before the older region [19]. When memory is not traversed by a garbage collector, such dangling pointers are safe because region inference has discovered that these pointers are not dereferenced by the program at runtime. Our pointer-tracing garbage collection algorithm, however, does not work when there are dangling pointers. Therefore, when garbage collection is enabled in the compiler, region inference is weakened to prevent dangling pointers by forcing values stored in a closure to live at least as long as the closure [20]. Only in special cases does this weakening of region inference alter region-annotations. Consider the following Standard ML program:

```
fun f a = ()
fun g v = fn () => f v
val h = g (2,3)
```

In this program, the function `f` makes no use of its argument. When applied to an argument `v`, the function `g` returns a closure containing `v`, which is a pointer if `v` is boxed. Applying the non-weakened version of region inference to the program yields the following region-annotated program:

```
fun f at r1 [] (a)= ()
fun g at r1 [r7] (v)= (fn () => f[] v)at r7
val h = letregion r8
        in g[r1] (2,3)at r8
        end
```

Due to the region-annotated types that are inferred for `f` and `g`, region inference concludes that the argument passed to `g` can be deallocated after the application of `g`. The result is that, after deallocation of region `r8`, `h` is bound to a closure containing a dangling pointer.

When garbage collection is enabled, on the other hand, the weakening of region inference ensures that regions holding values captured in a closure live at least as long as the closure itself. Thus when garbage collection is enabled the result of applying region inference to the binding of `h` yields the following region-annotated version of the binding:

```
val h = g[r1] (2,3)at r1
```

In this region-annotated version of the binding, the pair `(2,3)` is allocated in the global region `r1`, which happens to be the same region in which the closure returned by `g` is allocated.

It turns out that for all the benchmark programs mentioned in Section 4, the weakening of region inference as described here has no visible effect on memory usage or execution time.

3. THE ML KIT

The ML Kit is a Standard ML compiler that uses region inference as the basis for memory management [18]. The ML Kit is extended to support garbage collection of regions as described in the previous sections. The compiler is composed of a series of translations that gradually compiles programs into x86 machine code:[2]

- Elaboration. Programs that are invalid according to the language specification are rejected [16].
- Modules Compilation. In this phase, Modules are eliminated and program fragments are compiled into an explicitly typed language called `LambdaExp` [7].
- Optimization. An optimizer rewrites `LambdaExp` fragments as long as it can guarantee that the resulting fragments run in less space than the original fragments. Optimizations include function inlining, specialization of recursive functions, unboxing of function arguments, and elimination of polymorphic equality [6].
- Region inference. In this phase, `LambdaExp` fragments are translated into a language `RegionExp` in which memory directives are explicit [17].
- Region representation inference. Regions are divided into finite and infinite regions based on a static approximation to the number of values that are stored in the particular region [3].
- Register allocation and instruction selection. This translation compiles `RegionExp` fragments into x86 machine instructions [8, 13]. When garbage collection is enabled in the compiler, values are tagged so as to allow pointer tracing and pointer forwarding.

To execute a program compiled with the ML Kit, the generated x86 machine code is linked with a runtime system, written in C. The runtime system includes *region primitives* for manipulating the region-stack, such as primitives for allocating and deallocating regions, and primitives for allocating in regions. When garbage collection is enabled in the compiler, the generated code is linked with a version of the runtime system that integrates the region primitives with the garbage collector described in the previous sections.

The runtime system may also be compiled with support for *region profiling*, which makes it possible to inspect memory usage in regions over time [12].

3.1 Large Objects

There is one important aspect of the runtime system that the previous description of the garbage collection algorithm does not mention, namely how the runtime system manages

[2] A bytecode backend to the ML Kit is available as well.

large objects (i.e., objects that do not fit in a single region page). To manage large objects efficiently and to allow efficient natural representations of certain datatypes, such as strings and arrays, memory for large objects is allocated using `malloc` and associated with a particular region in a linked list, pointed to from a field in the region descriptor. Upon resetting or deallocation of a region, large objects in the associated linked list are deallocated using `free`.

Although certain types of large objects (such as large arrays and vectors) need be traversed by the garbage collector, large objects are never copied by the collector.

4. EXPERIMENTAL RESULTS

In this section, we describe a series of experiments that serve to describe the relationship between region inference and the garbage collection algorithm shown in Section 2.

We first investigate the effect of enabling tagging (Section 4.1). Section 4.2 looks at execution times and the number of garbage collections performed when garbage collection is combined with region inference. Section 4.3 seeks an answer to the question: of the memory reclaimed, what proportion is reclaimed by region management and what proportion is reclaimed by garbage collection? Finally, Section 4.4 compares memory usage and execution times with Standard ML of New Jersey (SML/NJ), another state-of-the-art Standard ML compiler. Section 4.5 compares the time and memory usage for bootstrapping the ML Kit with SML/NJ and the ML Kit itself, respectively.

All benchmark programs are run on a 750Mhz Pentium III Linux box with 512Mb RAM. Times reported are user CPU times and memory usage is measured in kilobytes using the `/proc` special file-system under the Linux operating system. We use m to specify memory usage (resident set size) and t to specify execution time (in seconds). Subscripts describe the mode of the compiler: $*_{\mathtt{r}}$ signifies region inference enabled, $*_{\mathtt{t}}$ signifies tagging enabled and $*_{\mathtt{g}}$ signifies garbage collection enabled (e.g., $t_{\mathtt{rt}}$ means time with regions and tagging enabled). We use $t_{\mathtt{smlnj}}$ and $m_{\mathtt{smlnj}}$ to denote execution time and memory usage for SML/NJ.

The experiments are performed with the ML Kit version 4.1.0 [18] and Standard ML of New Jersey 110.0.7. The benchmark programs are listed in Figure 3.

By *disabling* region inference, we understand instructing the region inference algorithm to allocate all values that would be allocated in infinite regions in one global region. Then not a single infinite region is deallocated at runtime and the garbage collection algorithm essentially reduces to Cheney's algorithm. Notice that disabling region inference in this sense does not change the property that many values are allocated in finite regions on the stack.

Whenever the size of the free-list becomes less than 1/3 of the total region heap, garbage collection is initiated upon the next function entry (i.e., safe point). After garbage collection, we make sure that the number of region pages in the region heap is at least three times the size of to-space (the *heap-to-live* ratio).

4.1 Effect of Tagging

When region inference is used without garbage collection, values need not be tagged so as to implement pointer tracing. Table 1 isolates the effect of tagging by showing the execution time and memory usage for the benchmark programs with tagging enabled and tagging disabled. In both cases, region inference is enabled and garbage collection disabled.

Program	Lines	Description
`vliw`	3676	VLIW instruction scheduler
`logic`	346	SML/NJ benchmark program
`zebra`	302	Solves the Zebra puzzle
`tyan`	1018	Grobner Basis calculation
`tsp`	493	Traveling salesman problem
`mpuz`	142	Emacs M-x mpuz puzzle
`DLX`	2836	DLX RISC instruction simulation
`ratio`	619	Image analysis
`lexgen`	1318	Lexer generation
`mlyacc`	7353	Parser generation
`simple`	1052	Spherical fluid-dynamics program
`professor`	276	Solves puzzle by exhaustive search
`fib35`	9	The Fibbonachi micro-benchmark
`tak`	17	The Tak micro-benchmark
`msort`	81	Sorting 100,000 integers
`kitlife`	230	The game of life
`kitkb`	725	Knuth-Bendix completion

Figure 3: The benchmark programs span from small micro-benchmarks (`fib35`, `tak`, and `msort`) to larger programs, such as `vliw` and `mlyacc`, that solve real-world problems. The `Lines` column shows the size of each benchmark. None of the benchmark programs, except `msort`, `kitlife`, and `kitkb`, has been optimised for region inference. The benchmark programs `fib35` and `tak` use only the runtime stack for allocation.

The table shows that tagging adds a substantial cost to execution time ($t_{\mathtt{r}} < t_{\mathtt{rt}}$) and to memory usage ($m_{\mathtt{r}} < m_{\mathtt{rt}}$).

For programs where lists and reals account for the majority of the memory usage, the memory overhead of tagging is close to 50 percent, due to the value tags in allocated pair and real values [6].

4.2 Effect of Region Inference on Garbage Collection

Table 2 demonstrates the effect of region inference on garbage collection. First, the table shows dramatic savings in number of garbage collections when enabling region inference (i.e., $\#GC_{\mathtt{rgt}} < \#GC_{\mathtt{gt}}$). Second, for most of the benchmark programs, enabling region inference decreases execution time (i.e., $t_{\mathtt{rgt}} < t_{\mathtt{gt}}$).

Third, comparing Table 2 and Table 1, we see that $t_{\mathtt{r}} < t_{\mathtt{rgt}}$ for all benchmark programs. The fastest execution is obtained by relying solely on region-based memory management.

Finally, Table 2 shows that, when combined with garbage collection, region inference often has a negative effect on memory usage (i.e., $m_{\mathtt{gt}} < m_{\mathtt{rgt}}$). This negative effect is mostly due to waste in regions, which we have more to say about in the next section. The programs `DLX` and `msort` behave very well without garbage collection (compare Table 2 and Table 1) and in these cases, when garbage collection is combined with region inference, garbage collection is initiated only at the start of execution; thus it does not become necessary to allocate three times the amount of live memory and therefore $m_{\mathtt{rgt}} < m_{\mathtt{gt}}$.

Effect of Region Inference on Garbage Collection

Program	Time (seconds)			Memory (bytes)			Collections		
	$t_{\mathtt{gt}}$	$t_{\mathtt{rgt}}$	%	$m_{\mathtt{gt}}$	$m_{\mathtt{rgt}}$	%	$\#GC_{\mathtt{gt}}$	$\#GC_{\mathtt{rgt}}$	%
`vliw`	1.99	1.16	42	1640K	2376K	–45	265	21	92
`logic`	6.94	7.02	–1	892K	892K	0	2582	2574	0
`zebra`	4.60	4.46	3	548K	644K	–18	2071	408	80
`tyan`	10.2	7.16	30	1352K	2800K	–107	1098	343	69
`tsp`	4.15	3.56	14	8532K	8536K	0	16	10	38
`mpuz`	10.2	10.2	0	568K	568K	0	2	2	0
`DLX`	9.33	7.40	21	5560K	4428K	20	102	3	97
`ratio`	1.86	1.76	5	1628K	1640K	–1	36	13	64
`lexgen`	7.87	6.77	14	3076K	3912K	–27	293	155	47
`mlyacc`	0.51	0.35	31	2676K	3680K	–38	60	29	52
`simple`	2.39	2.10	12	2372K	2452K	–3	16	6	62
`professor`	1.04	0.73	30	576K	640K	–11	2816	122	96
`fib35`	1.91	1.91	0	500K	500K	0	1	1	0
`tak`	14.0	14.0	0	500K	500K	0	1	1	0
`msort`	1.14	0.71	38	9912K	8328K	16	17	7	59
`kitlife`	1.89	1.83	3	612K	592K	3	818	2	100
`kitkb`	1.57	1.77	–13	1076K	1352K	–26	193	4	98

Table 2: There are significant savings in number of garbage collections performed when region inference is enabled. For most programs, enabling region inference also significantly reduces execution time. With respect to memory usage, the effect of enabling region inference strongly depends on the program. Improvements (e.g., $(t_{\mathtt{gt}} - t_{\mathtt{rgt}})/t_{\mathtt{gt}}$) are written in percentages.

Effect of Tagging on Time and Memory Usage

Program	Time (seconds)			Memory (bytes)		
	$t_{\mathtt{r}}$	$t_{\mathtt{rt}}$	%	$m_{\mathtt{r}}$	$m_{\mathtt{rt}}$	%
`vliw`	0.89	0.98	10	4376K	5880K	34
`logic`	3.15	3.82	21	128M	171M	34
`zebra`	3.68	3.91	6	6836K	10M	46
`tyan`	4.67	5.32	14	199M	283M	42
`tsp`	3.37	4.39	30	3624K	5820K	61
`mpuz`	8.01	9.17	14	524K	536K	2
`DLX`	6.01	6.99	16	2972K	3548K	19
`ratio`	1.44	1.54	7	2784K	3856K	39
`lexgen`	4.77	5.17	8	19M	27M	42
`mlyacc`	0.17	0.19	12	7796K	10M	28
`simple`	1.65	1.79	8	1276K	1708K	34
`professor`	0.66	0.68	3	4820K	6840K	42
`fib35`	1.38	1.69	22	480K	476K	0
`tak`	12.4	13.0	5	480K	476K	0
`msort`	0.50	0.57	14	4976K	6372K	28
`kitlife`	1.55	1.57	1	524K	564K	8
`kitkb`	1.60	1.64	2	1104K	1136K	3

Table 1: The effect of enabling tagging in the ML Kit. The time overhead $(t_{\mathtt{rt}} - t_{\mathtt{r}})/t_{\mathtt{r}}$ (written in percentages) varies from 1 to 30 percent with an average of 11 percent. The memory usage overhead varies between 0 and 60 percent with an average of 27 percent.

4.3 Memory Recycled by Region Inference

We now more deeply explore the relationship between garbage collection and region inference. We first ask: of the memory that is reclaimed, what percentage is reclaimed by region inference? (The rest must be collected by the garbage collector.) This fraction depends on the garbage collection strategy used; eventually region inference reclaims all garbage, when the program ends. The fewer times we garbage collect, the higher the fraction of garbage collected by region inference becomes.

Table 3 shows, in percentages, how much memory is recycled by region inference ($RI_{\mathtt{rgt}}$) and garbage collection ($GC_{\mathtt{rgt}}$), respectively. The table also shows the amount of *region waste* (i.e., non-used memory in region pages) as a percentage of the total amount of memory allocated for region pages. The region waste column is calculated as an average of region waste computed at each garbage collection invocation.

We see that region inference recycles the vast amount of memory for many of the programs. However, for the benchmark programs `logic`, `zebra`, `tyan`, `lexgen`, and `mlyacc`, a high percentage of memory is deallocated by the garbage collector. It is important to notice that finite regions are not accounted for here; finite regions are allocated in activation records on the stack. Previous measurements demonstrate that a high percentage of all values may be stored in finite regions [3].

To compute, for each garbage collection, the fractions of memory reclaimed by garbage collection and region inference, respectively, we proceed as follows. Let g_i be garbage collection phase i. Let L_i be the amount of live data after g_i (i.e., number of region pages in the to-space) and let $A_{\mathtt{p}}$ be the total number of region pages requested in the period between g_i and g_{i+1}. Moreover, let A_{i+1} be the number of region pages in from-space just before g_{i+1}.

Memory Recycling and Region Waste

Program	Recycling (%)		Waste (%)
	$RI_{\mathtt{rgt}}$	$GC_{\mathtt{rgt}}$	$W_{\mathtt{rgt}}$
`vliw`	85.2	14.8	22.9
`logic`	0.1	99.9	3.2
`zebra`	33.1	66.9	27.2
`tyan`	7.7	92.3	16.2
`tsp`	91.7	8.3	4.4
`mpuz`	100	0.0	–
`DLX`	100	0.0	–
`ratio`	75.5	24.5	16.0
`lexgen`	24.2	75.8	18.9
`mlyacc`	27.8	72.2	19.4
`simple`	92.5	7.5	17.8
`professor`	85.7	14.3	19.5
`fib35`	–	–	–
`tak`	–	–	–
`msort`	100	0.0	5.0
`kitlife`	100	0.0	–
`kitkb`	99.9	0.1	–

Table 3: The programs `fib35` and `tak` do not use regions, hence no values appear in the columns for these programs. No value appears in the third column for benchmark programs for which the garbage collector runs only a few times. The same heap-to-live ratio of 3.0 was used for all benchmarks.

Then, the amount of data reclaimed by garbage collection is $A_{i+1} - L_{i+1}$ and the amount of data reclaimed by region inference is $L_i + A_{\mathtt{p}} - A_{i+1}$. The total amount of data reclaimed is $L_i + A_{\mathtt{p}} - L_{i+1}$, thus, we get the fractions: $RI = \frac{L_i + A_{\mathtt{p}} - A_{i+1}}{L_i + A_{\mathtt{p}} - L_{i+1}}$ and $GC = \frac{A_{i+1} - L_{i+1}}{L_i + A_{\mathtt{p}} - L_{i+1}}$.

Figure 4 shows the fraction GC (i.e., $1-RI$) for the benchmark `professor` as a function of time. Throughout, region inference takes care of most of the deallocation.

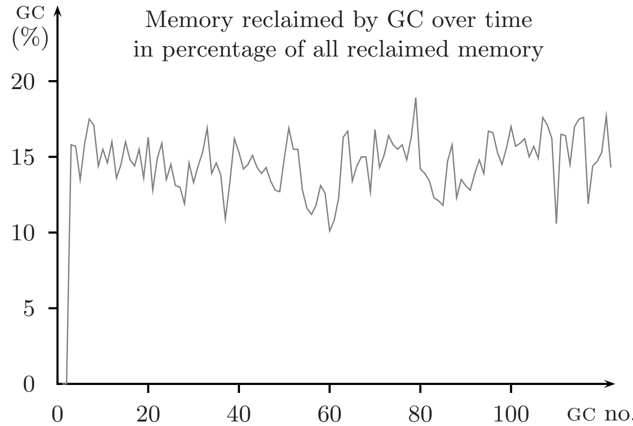

Figure 4: The figure shows the amount of memory (in percentages) reclaimed by garbage collection as a function of time. At garbage collection number 60, the garbage collector reclaims approximately 10 percent of the memory reclaimed since garbage collection number 59. Hence region inference reclaims 90 percent of the garbage in that period. Time is the garbage collection cycle number.

4.4 Comparison with SML/NJ

In this section we compare memory usage and execution time for executables generated by the ML Kit with executables generated by Standard ML of New Jersey version 110.0.7. The purpose here is not to suggest which compiler is better, but merely to demonstrate that combining garbage collection with region inference may produce results that are comparable in performance to state-of-the-art compilers. The measurements should be taken with a grain of salt; as mentioned earlier, differences in performance in code produced by different compilers may stem from many other factors than differences between regions and garbage collection.

Table 4 shows execution times and memory usage for all benchmark programs compiled with Standard ML of New Jersey.

The benchmark programs can be divided into three groups of programs; those that run in less time than with SML/NJ and uses less memory (programs `zebra`, `DLX`, `professor`, `fib35`, `tak`, and `kitlife`), those that run in more time and uses more memory (programs `tyan`, `lexgen`, `mlyacc`, and `simple`), and the remaining programs (`logic`, `tsp`, `mpuz`, `ratio`, `msort`, and `kitkb`).

The benchmark programs in the second group (in particu-

Comparison with Standard ML of New Jersey

Program	Time (seconds)			Memory (bytes)		
	$t_{\mathtt{smlnj}}$	$t_{\mathtt{rgt}}$	$\frac{t_{\mathtt{smlnj}}}{t_{\mathtt{rgt}}}$	$m_{\mathtt{smlnj}}$	$m_{\mathtt{rgt}}$	$\frac{m_{\mathtt{smlnj}}}{m_{\mathtt{rgt}}}$
`vliw`	1.44	1.16	**1.2**	556K	2668K	0.2
`logic`	3.75	7.02	0.5	1936K	1012K	**1.9**
`zebra`	10.4	4.46	**2.3**	864K	852K	**1.0**
`tyan`	3.78	7.16	0.5	1820K	2972K	0.6
`tsp`	20.8	3.56	**5.8**	7408K	8640K	0.9
`mpuz`	20.7	10.2	**2.0**	840K	768K	**1.1**
`DLX`	10.1	7.40	**1.4**	4892K	4564K	**1.1**
`ratio`	3.15	1.76	**1.8**	2500K	3312K	0.8
`lexgen`	5.95	6.77	0.9	304K	3976K	0.1
`mlyacc`	0.33	0.35	0.9	1692K	3504K	0.5
`simple`	1.64	2.10	0.8	1880K	2520K	0.7
`professor`	2.18	0.73	**3.0**	924K	832K	**1.1**
`fib35`	2.72	1.91	**1.4**	1072K	652K	**1.6**
`tak`	22.4	14.0	**1.6**	860K	648K	**1.3**
`msort`	0.87	0.71	**1.2**	3516K	8516K	0.4
`kitlife`	2.54	1.83	**1.4**	1064K	780K	**1.4**
`kitkb`	2.51	1.77	**1.4**	752K	1572K	0.5

Table 4: Comparison of Standard ML of New Jersey with the version of the compiler that combines region inference and garbage collection. A heap-to-live ratio of 3.0 was used for all benchmarks.

lar `tyan` and `lexgen`) suggest that the garbage collector does not always do a good job. There may be several reasons for this:

- All top-level variables are included in the root set. This inefficiency may be overcome by applying an analysis for nullifying top-level variables after their last use.
- Region waste. As shown in Table 3, some benchmark programs cause a significant amount of unused memory in region pages.
- Insufficient safe points. In the ML Kit, garbage collection may be initiated only at function entry—not at arbitrary allocation points.
- Lack of tail-calls. It is possible for region inference to introduce `letregion` constructs around expressions that occur in tail-call contexts and that would otherwise cause applications within the expression to be implemented as tail-calls. By widening the scope of such `letregion` constructs tail-calls may be implemented properly. This feature is not yet implemented.

For programs in the third group, it is possible to alter the trade-off between memory usage and execution time by altering the heap-to-live ratio (default is 3.0), because the heap-to-live ratio controls how often garbage collection is invoked.

4.5 Bootstrapping

In this section we compare bootstrapping the ML Kit using either SML/NJ or the ML Kit itself [18]. In the first setting, the SML/NJ compiler is used to compile the ML Kit sources into a version of the ML Kit that, when running, uses the SML/NJ runtime system. This version of the ML Kit is called kit_1. Using kit_1 to compile the ML Kit sources into kit_2 uses 809Mb and takes 40:41min.[3]

The kit_2 executable runs on the runtime system of the ML Kit using the combination of region inference and garbage collection. Using kit_2 to compile the ML Kit sources into kit_3 uses 904Mb and takes 17:33min. The combination of region inference and garbage collection works very well on this large program.

Figure 5 shows the memory behavior of the ML Kit (using region inference and garbage collection) when compiling the benchmark program `kitkb`. The global region `r1` is by far the largest. The ML Kit is not optimized for regions and without the garbage collector, region `r1` would grow without ever decreasing.

5. RELATED WORK

Related work fall into several categories. First, there is a large body of work concerning general garbage collection techniques [23, 14] and escape analysis for improving stack allocation in garbage collected systems [4]. The extra complexity of region inference and the polymorphic multiplicity analysis implemented in the ML Kit [3] allow more objects to be stack allocated than does traditional escape analyses, which allows only local, non-escaping values to be stack allocated.

[3] A 1 GHz Pentium III (Coppermine) machine equipped with 1Gb RAM is used for the bootstrapping experiments.

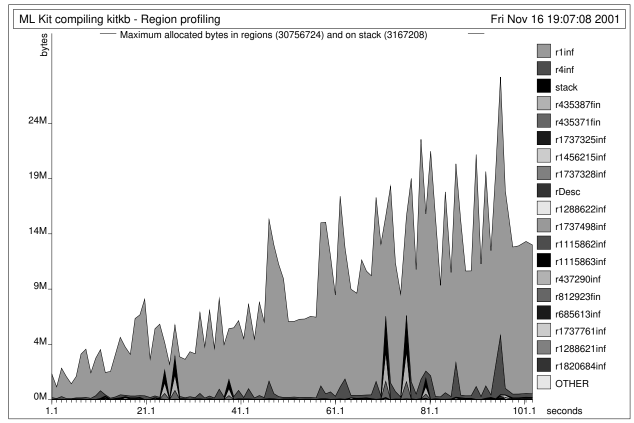

Figure 5: A region profile of the ML Kit using region inference and garbage collection when compiling the benchmark program `kitkb`. Without garbage collection, region `r1` would grow without ever decreasing.

Gay and Aiken demonstrate how explicit region based memory management may work succesfully for C when combined with reference counting of regions [9, 10]. In their RC system, however, individual objects in a region are not garbage collected until the entire region is freed, which is important in our case where, for instance, a programmer cannot be assumed to have arranged that the result of a function is stored in a region different from intermediate results computed by the function. Aiken et al. [1] show how region inference may be improved for some programs by removing the constraints of the stack discipline; such improvements to region inference could cause a fall-back garbage collector to run less frequently.

Also related to our work is the work on Cyclone [11], a safe version of the C programming language that uses regions for memory management. In Cyclone, a particular region—the heap—is garbage collected, whereas other regions, except regions corresponding to individual activation records, must be allocated and deallocated explicitly by the programmer.

Slightly related to our work is the work by Appel and Wang on integrating with a program a type-safe specialized garbage collector for the program [22]. The focus of their work, however, is on type safety and it is yet to be shown how well the approach works in a real system.

6. FUTURE DIRECTIONS

There are several directions for future work. First, the results in this paper suggest that the combination of region inference and garbage collection in a general purpose ML compiler is a viable and efficient strategy for memory management. As outlined in Section 4.4, however, there are several ways the interaction between region inference and garbage collection can be improved. In particular, the widening of some `letregion` constructs to ensure proper implementation of tail calls is called for. Also, arranging that garbage collection can be initiated at arbitrary allocation points—instead of only at function entry points—may improve memory usage for some programs.

A second direction for future work is to investigate the use of region inference as an advanced, polymorphic escape analysis by collapsing all infinite regions into one single heap. In such a setting, which corresponds to the disabling of region inference in our benchmark tests, an efficient generational collector could be used for the single heap.

Finally, a third direction for future work concerning the combination of garbage collection and region inference is to investigate the possibility of combining region inference with a tag-free (or nearly tag-free) garbage collection scheme. In the basic region typing rules, two values are forced into the same region, only if their types are identical. This property suggests that tags for pointer traversal can be moved from individual values to the level of regions, which could improve memory usage significantly for many programs.

7. CONCLUSION

Based on Cheney's copying garbage collection algorithm, we have developed a runtime system that integrates garbage collection with region based memory management.

The runtime system is implemented for all of Standard ML in the ML Kit compiler. Concerning execution time, measurements show that the fastest execution is obtained by using regions alone. Concerning memory consumption, the experiments confirm that most region-optimised programs use less space and time when using regions alone than when using regions combined with garbage collection or garbage collection alone. (Region inference does not need tags for pointer traversal, nor space for copying.) For programs that are not optimised for regions, adding garbage collection reduces memory usage but increases running times.

From the point-of-view of garbage collection, the measurements demonstrate that the pressure on garbage collection is reduced significantly by integrating garbage collection with region inference.

Finally, measurements show that the combination of region inference and garbage collection, as implemented in the ML Kit, is as efficient with respect to memory usage and execution time as a state-of-the-art generational garbage collection system.

8. ACKNOWLEDGMENTS

We would like to thank Ken Friis Larsen and Lars Birkedal for their careful reading of a draft of this paper. Also thanks to the anonymous referees for detailed comments and suggestions.

9. REFERENCES

[1] Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, pages 174–185, La Jolla, CA, June 1995. ACM Press.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, January 1996.

[4] Bruno Blanchet. Escape analysis : Correctness proof, implementation and experimental results. In *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 25–37. ACM Press, January 1998.

[5] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[6] Martin Elsman. Polymorphic equality—no tags required. In *Second International Workshop on Types in Compilation*, March 1998.

[7] Martin Elsman. Static interpretation of modules. In *Fourth International Conference on Functional Programming (ICFP'99)*, pages 208–219. ACM Press, September 1999.

[8] Martin Elsman and Niels Hallenberg. An optimizing backend for the ML Kit using a stack of regions. Student Project 95-7-8, Department of Computer Science, University of Copenhagen (DIKU), July 5 1995.

[9] David Gay and Alexander Aiken. Memory management with explicit regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, Montreal, Canada, June 1998. ACM Press.

[10] David Gay and Alexander Aiken. Language support for regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, Snowbird, Utah, June 2001. ACM Press.

[11] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, Berlin, Germany, 2002. ACM Press.

[12] Niels Hallenberg. 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.

[13] Niels Hallenberg. Combining garbage collection and region inference in the ML Kit. Master's thesis, Department of Computer Science, University of Copenhagen, 1999.

[14] Richard Jones and Rafael Lins. *Garbage Collection*. Wiley, 1996.

[15] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[16] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[17] Mads Tofte and Lars Birkedal. A region inference algorithm. *Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):734–767, July 1998.

[18] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with regions in the ML Kit (for version 4). Technical report, IT University of Copenhagen, October 2001.

[19] Mads Tofte and Jean-Pierre Talpin. A theory of stack allocation in polymorphically typed languages. Technical Report DIKU-report 93/15, Department of Computer Science, University of Copenhagen, 1993.

[20] Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201. ACM Press, January 1994.

[21] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[22] Daniel Wang and Andrew Appel. Type-preserving garbage collection. In *Conference Record of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 166–178. ACM Press, January 2001.

[23] Paul R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *Memory Management, Proceedings, International Workshop IWMM92*, pages 1–42. Springer-Verlag, September 1992.