A region-based memory manager for Prolog[*]

Henning Makholm[†‡]
henning@makholm.net

Abstract

We extend Tofte and Talpin's region-based model for memory management to support backtracking and cuts, which makes it suitable for use with Prolog and other logic programming languages. We describe how the extended model can be implemented and report on the performance of a prototype implementation. The prototype implementation performs well when compared to a garbage-collecting Prolog implementation using comparable technology for non-memory-management issues.

1 Introduction

The most important reason why real programmers like declarative programming is probably that a declarative language relieves the programmer of thinking about memory management. This is not in conflict with the commonly-seen slogan that declarative programming is about "what" rather than "how"; memory management is simply one of the major "how" problems that declarative languages obliterate.

Most contemporary implementations of declarative languages handle this by garbage collection. With garbage collection, the decisions about when and how to reuse some piece of heap memory are made solely at run time. Thus the time and effort spent by an imperative programmer to figure out when what can be safely deallocated has been traded for CPU cycles spent at *run time*. Present-day garbage collectors are so efficient that their cost in terms of run-time CPU cycles is quite small for most realistic programs, but it is still an interesting question whether one can improve the implementation by instead trading programmer effort for CPU cycles spent by the *compiler*.

[*]First Workshop on Memory Management in Logic Programming Implementations, London, 24th July 2000. A revised and extended version of this paper will appear at the 2000 ACM SIGPLAN International Symposium on Memory Management.

[†]DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 København Ø, Denmark

[‡]Supported in part by the Danish National Science Research Council via project PLT (Programming Language Technology)

Region-based memory management is one technique for moving memory-management decisions into the compiler. It was proposed by Tofte and Talpin [1994, 1997] for call-by-value functional languages and implemented in the ML Kit compiler [Tofte et al. 1997] for Standard ML. One advantage of region-based memory management is that it makes it possible to reason about the space *and* time requirements of programs and program fragments: With region-based memory management, a program is never interrupted by a garbage collection of indeterminate length; nevertheless, experience with the ML Kit shows that the performance of region-based memory management can be comparable to stop-and-copy garbage collection.

This paper reports on an attempt to apply region-based memory management to logic programming languages in general and Prolog with cut in particular. The part of this experiment that required new innovations was adapting the region-based run-time model to Prolog's destructive backtracking; thus the paper has a strong implementation slant, and most of the space is spent on describing the run-time architecture.

The organisation of this paper is as follows. Section 2 introduces the basic region-based model. Readers who are familiar with the ML Kit's representation of regions will not find much new here, but should nevertheless skim the sections to learn about the abstraction and notation we use afterwards. Section 3 describes how we adapted this model to work well with Prolog. Section 4 reports on some preliminary performance experiments, and Section 5 concludes.

2 Basic region-based memory management

We introduce region-based memory management by comparing the interface between the (run-time) *memory manager* and the *client program* (which we take to be the part of the running program that is not the memory manager).

In imperative languages such as C(++) or Pascal this interface consists of two operations:

$$\boxed{\begin{array}{ll} \mathit{alloc}: & n\text{: integer} \rightarrow \alpha\text{: pointer} \\ \mathit{free}: & \alpha\text{: pointer} \rightarrow \textit{(nothing)} \end{array}}$$

which the client program can use for allocating a block consisting of a specified amount of memory, and to deallocate the block when it is not necessary anymore.

For garbage collection the interface is simpler:

$$\boxed{\mathit{alloc}: \quad n\text{: integer} \rightarrow \alpha\text{: pointer}}$$

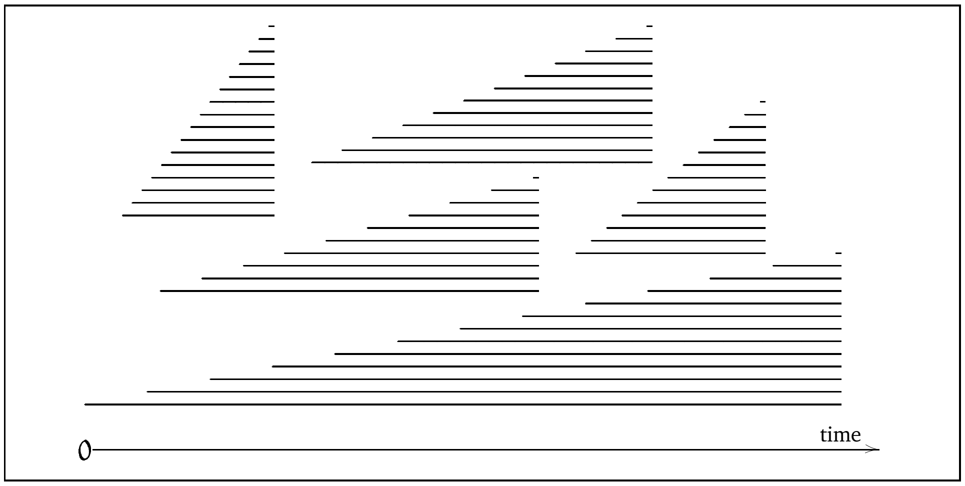

Figure 1: Memory-block lifetimes in the region model. Each horizontal line corresponds to the lifetime of one memory block. The triangular collections of blocks correspond to regions. The figure shows that several regions can exist at the same time, all growing; but a region can only shrink by being completely deallocated.

Here there is no *free* operation. Instead the garbage collector observes which use the client program makes of the allocated memory and may decide to deallocate a block once it can be seen to be unused. (Because the client program needs to adhere to certain conventions to make this observation possible, the interface is really not as abstract and clear-cut as it looks here; the point now is primarily that the client program does not directly control deallocations.)

The overall goal of region-based memory management is to let the compiler augment the original (declarative) program such that it does control the deallocation times of the memory it allocates. Thus the compiled client program that eventually runs behaves more like an imperative one with respect to memory management. However, it would be very difficult to arrange for every block of memory to be individually deallocated at the right time. The **region model** is the fundamental trick that keeps the complexity of the task under control: Instead of the simple two-operation interface shown above, a region-enabled client program uses a richer interface:

makeregion: *(nothing)* $\rightarrow$ ρ: REGION

alloc: ρ: REGION, n: integer $\rightarrow$ α: pointer

killregion: ρ: REGION $\rightarrow$ *(nothing)*

This interface introduces an abstract type called REGION. In the *alloc* operation, a region functions as a "licence to allocate memory". The client program can create and destroy regions at will, but destroying a region (with the *killregion* operation) has the important side effect that *every memory block that has been allocated using that particular region is implicitly deallocated*. Thus another way to look at a region is as a way to collect allocations into groups for simultaneous deallocation. Figure 1 shows a graphical representation of how the lifetimes of memory blocks in the region model relate to each other.

It ought to be mentioned here that this interface does *not* require region creation and deletion to follow a last-in-first-out discipline. Such a discipline has been implicit in the presentation of much of the earlier work on regions, but has never actually been followed in implementations. Indeed, the techniques of Aiken et al. [1995] and Birkedal et al. [1996] for avoiding memory leaks in tail-recursive programs depend on creating client programs that use regions in a non-LIFO pattern.

It is outside the scope of this article to describe the **region inference** process that augments the client program with *makeregion* and *killregion* operations. It turns out that the techniques and algorithms that have been developed for functional languages [Tofte and Talpin 1997; Tofte and Birkedal 1998] can be adapted[1] to Prolog with no fundamental changes except for the backtracking support discussed below.

On the other hand, the implementation of the region-based memory manager needs to be substantially enhanced to support Prolog. Therefore, let us start by describing the implementation of the basic region model.

The three-operation interface shown above is an abstract interface and could in principle be implemented in any of several ways. In practise, however, region-based memory management is nearly always associated with the following particular implementation in which each of the three operations can always be completed in constant time—even *killregion* which may deallocate an arbitrary number of memory blocks.

The heap is divided into fixed-size **pages**, and each region consists of a linked list of these pages. In each page a single word is used to point to the next page in the list, and the entire rest of the page is available for the client program's allocations. Each *alloc* operation tries to allocate memory for the newest page in the region; if there is not enough unallocated memory there a fresh page is added to the region, and whatever small amount of memory might have been free in the previously newest page is lost.

The *killregion* operation simply concatenates the region's page list onto a global list of free pages. That is a constant-time operation if we can find both ends of the page list quickly. Therefore, we implement the abstract type REGION as a pointer to a **management record** which contains pointers to either end of the page list as well as information about how many of the

[1]For details on this adaptation, see the technical report [Makholm 2000].

newest page's payload words have been allocated yet. The management record itself is allocated by *makeregion* in the first of the region's pages.

While this scheme implements each of the three operations in constant time, it rests on the assumption that each individual allocation is small enough to fit in a single page. The Prolog subset we support in our prototype implementation does not include the "`=..`" predicate or its cousins, so a constant bound on every allocation in a program can be determined statically. For less constrained languages, we conjecture that allocations which may be arbitrarily big occur seldom enough to permit special-case workarounds.

3 Extensions to the region model for Prolog

In this section we describe extensions to the region model which are necessary to make it work well in a Prolog implementation, and sketch the implementation we have implemented.

The extensions have to do with backtracking (Section 3.1), cuts (Section 3.2), and logical variables (Section 3.3). In Section 3.4 we briefly describe the running-time properties of our implementation.

3.1 Backtracking

When a typical Prolog implementation backtracks, it simultaneously undoes every heap allocation that has been made since the choice point was created. That is sound because Prolog offers no way of communicating intermediate results across backtracking (save for imperative features such as assert/retract which are normally handled in special ways) so the code that follows the backtracking cannot know any references to these allocations. Some implementations use this as their *only* kind of memory management, but even implementations with garbage collectors usually make sure to deallocate on backtracking, at least in the easy case where no collection has occured since the allocation.

If region-based memory management is to compete seriously with traditional memory-management strategies for Prolog, this deallocate-on-backtracking effect must be duplicated. Thus, we add operations to the memory-manager interface to tell the memory manager about backtracking:

makeregion: *(nothing)* $\rightarrow$ ρ: REGION
alloc: ρ: REGION, n: integer $\rightarrow$ α: pointer
killregion: ρ: REGION $\rightarrow$ *(nothing)*
pushchoice: *(nothing)* $\rightarrow$ *(nothing)*
backtrack: *(nothing)* $\rightarrow$ *(nothing)*

The intended semantics of *backtrack* is to undo not just *alloc*s but *every* memory-management operation since (and including) the latest *pushchoice* operation. In particular, *backtrack* may make a region reappear even if *kill-region* has been used on it while the choice point existed. This convention means that region inference basically does not have to care about backtracking, so that we can use region inference techniques developed for functional languages.

For the moment, we assume that every choice point created with the *pushchoice* operation is eventually used for backtracking. In Section 3.2 we shall consider the situation when cuts (either explicit cuts or "green" cuts inferred by the compiler) are allowed.

To implement this 5-operation interface, we must find ways for each memory-management operation to save enough information to allow itself to be undone at the right point in time.

3.1.1 How to undo *alloc* operations

When a *backtrack* operation is executed, some regions may have grown since the *pushchoice*; we must then **shrink** these regions accordingly. This means that we need to remember which size the regions had at the time of *pushchoice*. A choice point[2] must contain a (pointer to a) list of little structures that we call **snapshots**. Each snapshot contains enough information to restore one region to the state it had when it was created. That means that a snapshot contains a pointer to the region and a copy (a "snapshot") of the entire management record at the time the snapshot was created.

If would be inefficient to create snapshots for all existing regions each time *pushchoice* is executed, because the number of existing regions may be arbitrarily big. Instead we create shapshots on an as-needed basis: *pushchoice* creates none, but each time an *alloc* is executed it checks whether the newest choice point has a snapshot for the region and creates one if it hasn't.

Now, *backtrack* can undo the relevant *alloc* operations by traversing the choice point's snapshot list and **shrink** each mentioned region to the size saved in the snapshot.

Shrinking a region entails restoring the management record with the data stored in the snapshot, and perhaps cutting one or more pages out of the page list at the newer end of the region, contributing them to the free-pages list. With carefully chosen structure of the page lists, the latter task can be done in constant time.

[2]Ideally, we can imagine that the memory manager maintain its own stack of memory-management data related to the choice points. In a practical implementation, of course, it is more efficient to allow the memory manager to store its private data in the same choice point structures that the client program uses to keep track of its backtracking issues.

3.1.2 How to undo *makeregion* operations

Now let us consider what should happen if a *makeregion* primitive is executed between *pushchoice* and *backtrack*.

In this case, the *backtrack* will be backtracking to a point in time where the region did not exist at all. This means that *backtrack* must deallocate the region exactly as the *killregion* operation does it in the basic region model. So *backtrack* must be able to find the region. We store in the choice point a list of regions that should be deallocated at backtracking to that choicepoint. We call this list the **termination list**.

Intuitively, the fact that a region is mentioned in a termination list serves the same purpose as a snapshot: it tells something about the region's state at the time the choice point was created. Only in the case of a termination list entry the saved state is not "so-and-so big" but "not there at all".

It is straightforward to extend *makeregion* to insert the new region in the termination list, and *backtrack* to traverse the termination list and deallocate regions. We should also make sure that *alloc* operations immediately after the *makeregion* do not create snapshots, because it would be a waste of time for *backtrack* to meticulously shrink the region, only to deallocate it totally just thereafter.

3.1.3 How to undo *killregion* operations

If a *killregion* occurs between *pushchoice* and *backtrack*, there are two different cases to consider. The first is

pushchoice ... *makeregion* ... *killregion* ... *backtrack*

which is *not* difficult. Backtracking does not interfere with the region's life at all, and *killregion* can do just what it did in Section 2—except that the region should be removed from the choice point's termination list lest it be deallocated twice.

The difficult case is

makeregion ... *pushchoice* ... (*alloc*?) ... *killregion* ... *backtrack*.

Here the semantics of our backtracking region model is that *killregion* should make the region disappear, and *backtrack* should make it reappear in the same place. The only practical way to implement this is of course to make sure that the region is never deallocated at all. A first approximation to an implementation would be to have *killregion* do nothing at all in this case. Then it would be right there for *backtrack* to restore to its saved condition.

A unsatisfactory side of this approach shows if we imagine that *alloc* operations added a lot of pages to the region between *pushchoice* and *killregion*. The only thing *backtrack* does to these pages is to shrink them away immediately. It would be better to add them to the free-list as soon as the *killregion* call happens. Thus when *killregion* can't entirely deallocate the region, it instead shrinks it to the size it would assume anyway at the next *backtrack*.

3.2 Cuts

Now we consider how the model must be further extended to support cuts. We add a *cut* operation to the memory manager interface:

makeregion: *(nothing)* $\rightarrow \rho$: REGION
alloc: ρ: REGION, n: integer $\rightarrow \alpha$: pointer
killregion: ρ: REGION $\rightarrow$ *(nothing)*
pushchoice: *(nothing)* $\rightarrow \delta$: MARK
backtrack: *(nothing)* $\rightarrow$ *(nothing)*
cut: δ: MARK $\rightarrow$ *(nothing)*

Intuitively, a *cut* should undo one or more *pushchoice* operations *without* undoing the other memory-management operations that have occured since the *pushchoice*. The MARKs that appear in the signatures of *pushchoice* and *cut* serves to define how many choice points should be cut away. In the following discussion we assume, without loss of generality, that a single choice point is to be cut away; this is always the newest existing choice point. Call the choice point to be cut away C_0 and the second-to-newest one C_1.

The task is to reset the memory manager's administative data structures to the state they would have had if C_0 had never been created.

First C_0's termination list is concatenated onto C_1's termination list. Then each of the snapshots in C_0's snapshot list is inspected; one of the following three cases apply for each of them:

a) The region has neither a snapshot nor a termination list entry for C_1. The snapshot for C_0 should be *moved* to C_1's snapshot list.

b) The region has a snapshot or a termination list entry for C_1, and has not been killed. The snapshot for C_0 is obsolete and should simply be removed.

c) The region has a snapshot or a termination list entry for C_1, and it has been killed. After removing the C_0 snapshot, *cut* should try to restart the suspended *killregion* operation (which means that the region is either shrunk or deallocated).

3.3 Logical variables

Backtracking in Prolog must undo variable instantiations that happened after the choice point was created. This is usually implemented by recording the addresses of variable instantiations in a global **trail** that is consulted at backtracking time.

This scheme does not work well with region-based memory management, because a global trail might grow without bounds if the program never actually backtracks. It would be very costly to tidy the trail each time region pages were deallocated, so it appears that a global trail is incompatible with region-based memory management.

Instead we give each region its own trail. Rather than an array of variable addresses, we maintain the region-local trail as a (chronologically ordered) linked list of the instantiated variables. This means that it is unnecessary to set aside a separate area of memory for trail data; but on the other hand it is necessary to unwind the trail everytime the region is shrunk, lest the region ends up with some of its trail list in freed pages.

3.4 Time complexity considerations

As described in the technical report [Makholm 2000] it is possible to maintain enough redundant summary information in the snapshots and management records that every memory management operation can locate any snapshot it needs to inspect in constant time. Therefore, most memory-management operations take constant time. The exceptions are:

- *backtrack* takes time proportional to the number of regions to be deallocated or shrunk. This extra time can be amortized over the *makeregion* and *alloc* operations done since the choice point was created. Additionally, of course, it uses time proportional to the necessary amount of untrailing.

  This is still more predictable than garbage collection, but is not completely satisfactory, as it breaks our ability to reason about how long it might take for a program to get from any point A to any point B. It now holds only when the execution path that contains point A has not yet failed when point B is reached.

  We argue, however, that these situations still account for most of the cases where one would *want* to reason about running times. The time to get from A to B is really interesting only when A and B represent interaction with the program's environment. And Prolog programs rarely interact with their environment in execution paths that might fail and backtrack, at least not readable and maintainable Prolog programs.

- *cut* takes time proportional to the number of snapshots in the cutaway choice point's snapshot list. This extra time cannot in general be amortized, because a snapshot may, in general need to be moved an unbounded number of times, but we conjecture (based on our inability to construct a counterexample) that it is amortizable if we restrict our attention to client programs produced by our region inference algorithm.

- *cut* and *killregion* may shrink regions and thus use time proportional to the amount of untrailing this shrinking necessiates.

  This extra cost is especially unsettling for *killregion* because we feel that the pure region-based operations ought to be constant-time operations. It would be possible to avoid this by letting *killregion* refrain from shrinking the region if there is untrailing to be done, but we are not sure how that would affect the predictability of the program's *space* needs.

4 Experimental results

As implied in the previous sections, the memory-management operations must do significant (if mostly bounded) work in order to maintain the administrative data that allows backtracking at the region level. It might therefore be feared that backtracking-aware region-based memory management is inherently so slow that it is unusable in practise. We have done a short series of practical experiments to investigate whether such fear is justified.

The experiments compare the running times for a prototype region-based compiler for a subset of Prolog with the running times for a garbage-collecting reference compiler which has been derived from our region-based prototype such that the only difference between the two implementations is the memory-management strategy. The garbage collector is a simple non-generational copying garbage collector.

These implementations are available electronically at ⟨`http://www.diku.dk/students/makholm/rpsys.tar.gz`⟩. In order to be able to stress the memory manager more, both implementations treat numbers as boxed values which must be allocated on the heap.

It should be noted that our experiments do not aim at being the definitive comparison between region-based memory management and garbage collection. For example, our region-based implementation does not yet implement the "multiplicity analysis" technique of Birkedal et al. [1996] (which in some cases can increase the efficiency of region-enabled programs dramatically by allocating provably small regions on the call stack instead of the heap), and the garbage collector used in the experiments is handicapped somewhat by not being generational. Still, the experiments ought to tell something about whether the costs of region-based memory management versus garbage collection are of the same order of magnitude at all.

	`10queens.pro`			`puzzle.pro`		
	GC	WAM	regions	GC	WAM	regions
Running time (s)	19.0	18.3	20.0	7.1	6.1	7.3
Max net heap size	347	347	162	20103	20103	3903
Max gross heap size	347	347	368	20103	20103	4512
Max regions alive			12			13

Figure 2: Running times from the the first set of experiments. The "net heap size" is the number of machine words actually allocated by the client program, whereas the "gross heap size" includes management data and unused areas of region pages.

	`ack.pro`		`quick.pro`		`filerev.pro`	
	GC	reg	GC	reg	GC	reg
Running time (s)	19.2	18.3	5.7	5.7	3.0	1.3
Max net heap size		2054		287686		94120
Max gross heap size	65472	32736	615168	307584	217856	108928
Max live data	7		80007		89871	
Max regions alive		2046		56		726

Figure 3: Running times from the second set of experiments. The unit of heap size are machine words. The size of the garbage-collected heap is the sum of the sizes of the semispaces.

Figure 2 shows results for two example programs that backtrack so often that the garbage-collecting implementation does all of its memory management by backtracking and never gets to start a collection at all.

`10queens.pro` finds all solutions to the 8-queens problem (extended to 10 queens and a 10×10 chessboard to increase the running time). `puzzle.pro` solves a series of cryptoarithmetic puzzles of the "`SEND+MORE=MONEY`" variety using brute force and logical variables.

For these programs we also added a second reference implementation (labeled "WAM" in the table—which does not mean that it is a strictly WAM-based implementation, only that the memory management is WAM-like) which relies purely on backtracking for memory management. That implementation is a little faster than the garbage-collecting one; primarily because it does not check for heap overflow (it wouldn't be able to do anything about it).

We can see that the region-based implementation is 5–10% slower than the two reference implementations. This is not unexpected—the region-based memory manager does a lot of work that eventually turns out to be unused—but the difference is also not so big that we think it ought to disqualify region-based memory management in general.

Figure 3 shows results for three example programs that run for long enough without backtracking to need real memory management. We have generously allowed the garbage-collecting implementation to use twice as much heap as the region-based one needs, because the garbage collector divides the available memory into two semispaces and only uses one at the time. The figure also shows the maximal amount of live data in any collection; this number can be used to judge how our heap allotment would compare to a garbage collector which sizes its heap adaptively.

`ack.pro` computes a value of Ackermann's function with input $(3, 8)$. It was selected for being extremely friendly to the copying garbage collector due to the low amount of live data at any point in the computation. Furthermore, the program uses the region model rather inefficiently (it only allocates 2 words in each region on average), yet the region-based implementation happens to outperform the garbage-collecting one. We suppose this is because the region-based memory manager uses a LIFO memory reuse strategy and thus has better locality of reference than the inherently FIFO garbage collector. It does not better the running time with the garbage collector to decrease the heap size (the more frequent collections begin to dominate) but a generational collector may score better.

`quick.pro` sorts a list of 20000 pseudorandom number using a list-processing Quicksort. Here—although that is not shown in the table—the garbage collector *can* be made to outperform the region model slightly by increasing heap size.

`filerev.pro` was selected to be very hostile to the garbage collector—it keeps a lot of data live while only working on a small subset of them. As expected, the region-based implementation wins this race easily.

5 Conclusion

We have described how a region-based memory manager can be extended to support backtracking.

The predictive timing properties that make region-based memory management attractive for functional languages do not completely carry over to our variant for Prolog, but we still think that our extended region model provides better control over the time used for memory management than garbage collection.

Experiments with a prototype implementation show that it is likely that region-based memory management for Prolog can compete with and in some cases outperform memory management by garbage collection.

These results suggest that it would be worthwhile to investigate how region-based memory management performs in larger and more realistic contexts than our small example programs.

References

Aiken, A., Fähndrich, M., and Levien, R. [1995]. Better static memory management: Improving region-based analysis of higher-order languages (extended abstract). In *Programming Language Design and Implementation (ACM SIGPLAN Conference, PLDI '95, La Jolla, CA, USA)*, special issue of *ACM SIGPLAN Notices*, 30(6):174–185. ⟨`http://http.cs.berkeley.edu/~aiken/ftp/region.ps`⟩.

Birkedal, L., Tofte, M., and Vejlstrup, M. [1996]. From region inference to von Neumann machines via region representation inference. In *Principles of Programming Languages (23rd ACM SIGPLAN-SIGACT Symposium, POPL '96, St. Petersburg Beach, FL, USA)*, pages 171–183. ACM Press, New York, NY, USA, ISBN 0-89791-769-3. ⟨`http://www.diku.dk/users/tofte/publ/popl96.ps.gz`⟩.

Makholm, H. [2000]. Region-based memory management in Prolog. Master's thesis, Department of Computer Science, University of Copenhagen. ⟨`http://www.diku.dk/~makholm/speciale.ps.gz`⟩.

Tofte, M. and Birkedal, L. [1998]. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):724–767. ⟨`http://www.itu.dk/research/mlkit/kit_general/toplas98.ps.gz`⟩.

Tofte, M. and Talpin, J.-P. [1994]. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Principles of Programming Languages (21st ACM SIGPLAN-SIGACT Symposium, POPL '94, Portland, OR, USA)*, pages 188–201. ACM Press, New York, NY, USA, ISBN 0-89791-636-0. ⟨`ftp://ftp.diku.dk/diku/semantics/papers/D-235.dvi.gz`⟩.

Tofte, M. and Talpin, J.-P. [1997]. Region-based memory management. *Information and Computation*, 132(2):109–176. ⟨`http://www.itu.dk/research/mlkit/kit2/infocomp97.ps`⟩.

Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N., Olesen, T. H., Sestoft, P., and Bertelsen, P. [1997]. Programming with regions in the ML Kit. Technical Report DIKU-TR-97/12, Department of Computer Science, University of Copenhagen. ⟨`http://www.diku.dk/research-groups/topps/activities/kit2/diku97-12.a4.ps.gz`⟩.