A region-based memory manager for Prolog[*]

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

Categories and Subject Descriptors

D.1.6 [**Programming Techniques**]: Logic Programming; D.3.2 [**Programming Languages**]: Language Classifications—*constraint and logic languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*dynamic storage management*; D.3.4 [**Programming Languages**]: Processors—*memory management, run-time environments*

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 allows the programmer to forget.

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*.

Region-based memory management is one technique for moving memory-management decisions into the compiler. It was proposed by Tofte and Talpin [9, 10] for call-by-value functional languages and implemented in the ML Kit compiler [8] for Standard ML. One advantage of region-based memory management is that it makes it easy to reason about the space *and* time requirements of programs and program fragments: With region-based memory management all memory-management operations take constant time to complete, and a program is never interrupted by a garbage collection of indeterminate (or at least hard-to-predict) 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. Though there exist incremental garbage collection techniques that also allow each memory-management operation to complete in constant time, the overhead of these methods tend to be much higher than for region-based memory management.

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 introduces our prototype Prolog compiler which implements region-based memory management. Section 5 briefly discusses the **region in-**

[*]This is an extended and revised version of a paper that appeared at the CL2000 Workshop on Memory Management in Logic Programming Implementations, July 24th, London

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

ference process that prepares Prolog programs to use the region-based memory manager. Section 6 reports on some preliminary performance experiments, and Section 7 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:

alloc:	n: integer $\to$ α: pointer
free:	α: pointer $\to$ *(nothing)*

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:

alloc:	n: integer $\to$ α: pointer

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)* $\to$ ρ: REGION
alloc:	ρ: REGION, n: integer $\to$ α: pointer
killregion:	ρ: REGION $\to$ *(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

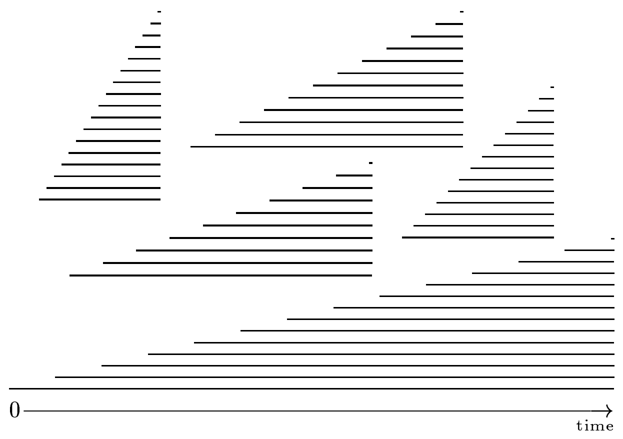

Figure 1: Memory-block lifetimes in the region model. Each horizontal line corresponds to the lifetime of one memory block. The triangular collections of blocks correspond to regions. The figure shows that several regions can exist at the same time, all growing; but a region can shrink only by being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

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 for avoiding memory leaks in tail-recursive programs we describe in Section 5.4 all depend on creating client programs that use regions in a non-LIFO pattern.

2.1 Implementing 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 RE-

GION 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 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. In the Prolog subset we support in our prototype implementation, we do not support built-in predicates like `=../2` which allows construction of structures with arbitrary many members, or built-in predicates to create atoms that do not occur in the source program. Therefore, a constant bound on every atomic 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.

2.2 Space efficiency

If we assume that regions grow big and that the page size is large compared to the size of individual allocations, the region model has an excellent ratio of words used for memory management per word used for the client program's payload data. On average, only a few words per page full of payload data will be used for the link to the next page or wasted as slack at the end of a page.

However, if a region does *not* grow big—that is, if only a couple of words is ever allocated in it—the figure is not as favourable because an entire page will be used to hold that couple of words. This suggests that pages should not be too big, and definitely should be smaller than the "pages" of several hundred words each that are used for implementing virtual memory. In the experiments we report on in Section 6 we use a total page size of 16 words. This might seem excessively small, but remember that the administrative overhead per page is only one word.

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 occurred 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)* $\rightarrow \rho$: REGION
alloc: ρ: REGION, n: integer $\rightarrow \alpha$: pointer
killregion: ρ: REGION $\rightarrow$ *(nothing)*
pushchoice: *(nothing)* $\rightarrow$ *(nothing)*
backtrack: *(nothing)* $\rightarrow$ *(nothing)*

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.

The intended semantics of *backtrack* is to undo not just *allocs* but *every* memory-management operation since (and including) the latest *pushchoice* operation. In particular, *backtrack* may make a region reappear even if *killregion* 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.

It might be argued that it would be a cleaner design to require the client program to keep track of backtracking itself and use *killregion* only when it knows that the region will not be needed by pending choice points. However, consider a program fragment such as

```
makeX(X) :- ... .          % deterministic
makeY(why).
makeY(wye).
computeZ(X,Y,Z) :- ... . % deterministic
check(Z) :- ... .          % may fail
foo :- makeX(X),
       makeY(Y),
       computeZ(X,Y,Z),
       check(Z).
```

What should happen with the region that holds `X`?[1] It is not needed after the call to `computeZ/3`, so when we let the memory manager handle backtracking, we can safely insert a *killregion* operation at a point between `computeZ/3` and `check/1`. The first time around, there is still a choice point left over from `makeY/1`, so the memory manager should not really deallocate the region. The second time the same *killregion* is executed there are no choice points left, so the region can be deallocated.

If it were the client program's responsibility to only execute the second of the *killregion* operations, it would either have

[1]Of course, if `X` has a complex value, its components may be distributed over several regions—but for the sake of example we assume there is only one region involved here.

to contain two different copies of the code for the two last calls, or to maintain and test flags that keep track of which subgoals have choice points left. The former of these options is impractical because it might well cause an explosion in code size, and the latter means that the decision of whether to deallocate the region between `computeZ/3` and `check/1` is made at run time anyway. And if this decision must be made at run time, we think it is cleaner to let it be made by the memory manager, which already needs to track the relation between choice points and regions in order to reclaim partial regions at backtracking.

Now, how can we implement the 5-operation memory-manager interface above? The difference from traditional region-based memory management is that 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. In general only a few of the (arbitrarily many) existing regions will be used for allocations before the next *pushchoice* operation; creating (and eventually removing) snapshots for the other regions would be a profound waste of space and time. 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.

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*.

[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.

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 choice-point. 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)* $\rightarrow$ *(nothing)*
backtrack: *(nothing)* $\rightarrow$ *(nothing)*
cut: *(nothing)* $\rightarrow$ *(nothing)*

The *cut* operation pops away a single choice point[3] from the choice point stack, but without restoring the heap state to the one that existed at *pushchoice* time.

Call the choice point to be cut away C_0 and the second-newest one C_1. The task is then 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.

There is a problem here, however, because a global trail might grow without bounds if the program never actually backtracks. Traditionally, this is solved by **tidying** the trail when some of its entries might become obsolete—for example during *cut* operations. In the region-based implementation this strategy means that each time region pages are freed, the trail should be tidied of entries to variables in those region pages. That is not easy, however. The problem is how to find out whether a particular trail entry should be tidied away. In traditional memory models this can be accomplished by a few pointer comparisons, but a region-page list that is deallocated can contain pages with wildly different addresses in the heap.

[3]For expositional reasons we ignore the fact that Prolog's cut operation can cut away multiple choice points. Though the effects of a multi-level cut should be identical to performing *cut* operation described here multiple times, our implementation uses an integrated multi-level cut that considers the fate of each snapshot in light of the knowledge that an entire range of choice points are about to disappear.

The solution we select is to 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.

During the revision of this paper we bacame aware that a global trail *could* still be used with region-based memory management, if each trailed variable had attached a reference to the region it is allocated in, and an allocation-sequence number that can be compared to sequence numbers stored within the snapshots. This solution would consume more memory (each instantiated variable would need 4 words: value, trail entry, region reference, sequence number—as opposed to 2 in our current solution) but it is possible that its time efficiency would be better than the per-region trail strategy. Regrettably, we have not yet had time to test out this idea in practise.

3.4 Time complexity

As described in the technical report [5] 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 cut-away 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. In fact, it is possible to construct sequences of memory-management operations such that the total number of times a snapshot needs to be moved grows quadratically in the number of operations. We conjecture, however (based on our inability to construct a counterexample) that if we restrict our attention to client programs produced by our region inference algorithm, the time spent by *cut is* amortizable.

- *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. A PROTOTYPE IMPLEMENTATION

We have implemented the region-based memory manager described in the previous section, and used it as the run-time module of a prototype region-based Prolog compiler that compiles Prolog to machine code using C as "portable assembly languaget".

The subset of Prolog supported by this prototype consists of pure Prolog with cuts and a very limited set of built-in predicates for arithmetic and primitive character-based I/O.

Except for memory management, the implementation uses very simple and straight-forward techniques. For example, predicate calls follow a uniform scheme and *always* involce creating a choice point. In order to be able to stress the memory manager more, the implementation treats numbers and atoms as boxed values which must be allocated on the heap.

The prototype implementation is available electronically at ⟨`http://www.diku.dk/students/makholm/rpsys.tar.gz`⟩. This also contains the reference implementations we used for the comparisons in Section 6.

5. REGION INFERENCE

In this section we briefly sketch the **region inference** process that transforms a Prolog source program to a client program which uses the region-based memory manager. There is not much novel and innovative here, but we include it nevertheless because many readers of earlier drafts asked for information about this. Due to space constraints we can only give a rough sketch here; for details we refer to the technical report [5].

The run-time properties we require of the client program are, in order of importance:

Soundness: No region is ever killed before the last use of any memory block that has been allocated in it.

Precision: A memory block should be allocated in a region that is killed reasonably soon after the last use of the block.

Economy: The client program should use as little time as possible on creating regions, passing around REGION handles, etc.

Soundness is of course an essential property that must not be compromised; but with precision we necessarily have to settle for approximations. The precision of region inference depends on the source program as well as the strength of inference methods used. Because region inference is a static analysis whereas garbage collection occurs at run-time, it may end up with better or worse precision than garbage collection, depending on the program. For example, region inference can sometimes discover that it is safe to create *dangling pointers* by killing a region while there are still live pointers to it (something a garbage collector would never do)—but conversely it is also possible for region inference to be forced to let a short-lived memory block be allocated in a very long-lived region.

5.1 Type system

One of the keys to succesful region inference is to allow the different parts of a compound values (for example the elements of a list versus the cons cells of the list structure) to be allocated in different regions. This means that the region inference has to be able to reason about which parts a value might consist of. Like the existing region inference techniques, ours use *types* to structure this reasoning.

Now, Prolog is an untyped language, and no matter which implementation techniques we use, it ought to remain untyped as far as the programmer is concerned—or it would not be Prolog. We therefore use a "soft" type system that allows any syntactically correct program to be typed and automatically adjusts itself to be as precise as the program allows.

Our type system is similar in spirit to that of Mycroft and O'Keefe [6], but has the special feature that the type checker automatically derives a set of appropriate recursive datatype definitions for the program. Our prototype implementation does not support polymorphism (because polymorphic type checking with unlimited mutual recursion between predicates is undecidable without programmer-supplied type annotations). However, as evidenced by the large body of work on region inference for Standard ML, polymorphism does not conflict at all with region inference, and any practical heuristic for adding polymorphism to the type checker would blend nicely into the region inference and help to increase its precision.

5.2 Taming unification

A unique property of logic programming is that every computation step the program executes is expressed as a variant of a universal construct: unification. Thus, a unification such as "$X = f(Y)$" can be used for quite diverse purposes, depending on the instantiation states of X and Y. If X is free, the unification *constructs* an f structure. If Y is free and X is bound to some non-veriable, the unification *inspects* an existing structure. If X and Y are both bound, the unification is a *proper unification* which compares structures to an arbitrary depth and possibly instantiates variables found inside either value.

Finding out which purposes each unification serves is important for the region inference. The most prominent difference is that between a proper unification and a mere construction or inspection. Because a proper unification may need to compare arbitrarily deep pieces of the terms, the region inference must make sure that neither of the values contain dangling pointers. However, in the case of a mere construction or inspection, it does not hurt if there is a dangling pointer somewhere inside the operand of the f functor. (In fact, the operand itself could be a dangling pointer, if no other piece of the program needed to follow it). Thus it will improve the precision of the region inference to know which unifications can never be proper.

It is also of some interest to be able to distinguish constructions from inspections. Though this does not improve the precision of the region inference (in both cases the region that contains—or will contain—the f functor must be alive), it can improve the economy of the client program, because the REGION handle is necessary for constructions, which allocate memory, but not for mere inspections. For example, a procedure that traverses a (known and closed) list does not need to know *which* region the cons cells lie in, as long as the region inference guarantees that whatever region it is will always be alive when the procedure is called. On the other hand, a procedure that *creates* a list may need to be passed a REGION handle as parameter, so it can allocate the cons cells in the region expected by the caller.

To approximate the instantiation states of the values manipulated by a logic program—and, by extension, to classify the rôles of its unification—is the job of a *mode analysis*. Several mode analyses are described in the literature; the one we use in our prototype implementation is a rather naïve type-based one which has been designed purely to be easy to implement while not producing too conservative results for our example programs. A more advanced mode analysis could readily be substituted, and would enhance the precision and economy of the region inference in addition to the other well-known but not memory-related optimizations made possible by mode information.

Our prototype program uses the mode information to translate the Prolog program into an intermediate form where different instructions are used for structure creation, inspection, and proper unification.

5.3 Identifying regions

After these preliminary exercises, it is time to actually assign regions the program's data. At this point, the mode-based translation to intermediate code has broken most unifications down to simpler steps. Also, our decision to make the memory-manager's *backtrack* operation able to undo *any* memory-management operation means that the region inference needs not be concerned about backtracking at all—we can simply pretend that at each predicate call a clause to execute is selected at random, and that failure stops the program. With unification and backtracking thus taken care of, the remaining problem can be handled with the same methods as used for functional languages. For the purpose of region inference, logical variables can be handled as special cases of Standard ML's reference types.

In brief, the standard methods employed here consists of:

1. Type check the program and annotate each expression and each procedure parameter with its type tree (or type graph, when recursive types are involved).
2. Decorate each node of the type trees (graphs) with a fresh **region name**. Some of these region names eventually become client-program variables that contain REGION handles; the annotation on a type node tells where to find the handle of the region where a structure described by the type node should live.
3. Unify region names with each other, as required by the **region type system** used. The region type system makes sure that the region-annotated types' claims about where the data can be found will actually be true at run time.
4. Based on the region-annotated types, decide when each region should be created and killed. The rules for this decision is also part of the region type system.

The region type system is responsible for the soundness of the results, and the fact that all region names start out different and are unified only as required by the region type system helps extract the maximal amount of precision from the region type system. The property we call economy has not received much attention in the literature. The "removal of get-regions" described by Birkedal et al. [3] is one optimization that aims at economy; our prototype implementation does this, together with a "region merge" optimizations that merges regions if they are scheduled to be killed at the same time (in which case there is no point in keeping them separate).

5.4 Region type systems

The original region type system by Tofte and Talpin is found in [9, 10], which also give a proof of its soundness. A detailed inference algorithm is constructed by Tofte and Birkedal [7]. Most of the complications there have to do with supporting higher-order types which do not occur in our setting, so region inference for Prolog is actually simpler than for Standard ML.

The main problem with the Tofte–Talpin region type system is that it is very imprecise when applied to iterative (i.e., tail-recursive) computations. This is because it insists that the lifetimes of different regions follow a stack discipline in unison with the expression structure of the source program. Therefore, anything that goes *into* a tail call must live in regions that can only be killed after the tail call returns.

Different strategies for alleviating this problem have been proposed. The one used by the ML Kit adds a "storage mode analysis" [3] which runs after the region inference and identifies places where it should be safe to **reset** a region, which means to deallocate all data *within* the region but not kill the region itself. With this strategy tail-recursive programs *can* run in constant space, but the tail recursion is programmed in special ways which are not always intuitive [8]. No proof of the soundness of resetting has appeared in print.

Another strategy was proposed by Aiken, Fähndrich, and Levien [1] who extend the Tofte–Taplin system to allow a region to be killed before control leaves its lexical scope. For example, a region that holds a parameter of a tail call (and whose lexical scope therefore must include the entire call) is allowed to be killed by the called procedure. This allows good region annotations of iterative programs with less rewriting than the ML Kit solution requires. Additionally, this system allows dangling pointers in more places than the Tofte–Talpin system (which let pointers dangle only within function closures). A soundness proof is sketched in [1].

Crary, Walker, and Morrisett [4] recently described a yet stronger region type system and prove its correctness, but did not investigate how region inference could be automated.

Developing region type systems that gives good results for a wide range of programming styles with a minimum of rewriting is still an open research area. Future improvements are likely to be applicable to Prolog, too.

In our prototype implementation we use a very simple region type system most akin to the original Tofte–Talpin one. Therefore our prototype does not perform well on iterative programs.

5.5 Multiplicity analysis

Recall from Section 2.2 that region-based memory management can waste a lot of space in regions where only a few words are ever allocated. The ML Kit addresses this problem with a *multiplicity analysis* [3] which runs after region inference and idenfifies regions that are provably only used for a single allocation operation each. These "finite" regions are then *not* handled by the normal region-based memory manager; instead the appropriate amount of memory is reserved on the call stack. That avoids rounding the gross memory usage for the allocation up to the region-page size.

Experiments with the ML Kit has shown that this optimization not only improves the space efficiency of the program, but also results in a significant speed-up because allocating memory on the call stack is quicker than initializing a region.

Our prototype implementation does not include a multiplicity analysis. We expect that it would be possible to allocate finite regions directly on Prolog's local stack, but the details of the interaction with backtracking have yet to be worked out

6. EXPERIMENTAL RESULTS

As can be seen from Section 3, 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 nongenerational copying garbage collector and does not implement variable shunting or early reset (see [2]).

Our experiments do not aim at being the definitive comparison between region-based memory management and garbage collection. As noted in the previous section, most parts of our prototype's region inference are significantly cruder than the state of the art; we have deliberately selected example programs that are known to work well togeter with the prototype region inference. On the other hand, the garbage collector used in the experiments is not representative of the state of the art in garbage collection for logic programming languages. 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.

The timing experiments were made on a HP Apollo 9000/ 735 workstation with a 99 MHz PA-RISC1.1 processor, running HP-UX 10.20. To give an idea of the general performance of our prototype implementations, we measured the running time of our `10queens.pro` and `ack.pro` example programs with SICStus Prolog 3.8.3 in compiled mode on the same machine. Our implementations run both of these programs in between 18.3 and 20.0 seconds; SICStus uses 16.4 seconds for `10queens.pro` and 11.9 seconds for `ack.pro`. It should be noted that whereas (as far as we understand) SICStus "compiles" to bytecode which is subsequently interpreted by the runtime system, our implementations compile to machine code.

Table 1 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 partially instantiated data structures.

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 and is seen to be a little faster than the garbage-collecting one. This difference is caused by time spent by the garbage-collecting implementation on checking for heap overflow and excluding the random values in allocated-but-not-yet-used local-stack words from the root set. (For the `10queens.pro` example, all local-stack words happen to be filled with meaningful data as soon as they are allocated, so the only possible explanation for the time difference here is the heap-overflow check).

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.

Table 1: 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.

	`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	162		20103	3903
Max gross heap size		347	368		20103	4512
Max regions alive			12			13

Table 2: Running times from the second set of experiments. The size of the garbage-collected heap is the sum of the sizes of the semispaces.

	`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	65427	32736	615168	307584	217856	108928
Max live data	7		80007		89871	
Max regions alive		2046		56		726

Table 2 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` is a classic benchmark for region-based memory management. It 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.

7. 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.

7.1 Further work

These results are encouraging, but cannot guarantee that region-based memory management performs well in larger and more realistic contexts than our small example programs. Eventually, the only real test would be to add region-based memory management to an existing Prolog implementation and compare its performance with the same implementation using garbage collection. This is not something that can be done immediately, however; the following is a nonexhaustive list of problems that must be solved before region-based memory management can be used for full Prolog:

- The built-in predicates `functor/3`, `arg/3`, and `=../2` which allow constructing and analysing structures using a dynamically-determined atom as the functor and a dynamically-determined number of arguments. It should not be too difficult to mix these with regions at runtime. However, they are fundamentally hostile to a type-based region inference, and while a well thoughtout type system might be able to resolve some cases to more benign constructs, it would still need to have a conservative fall-back option, which in turn could have devastating effects on the region inference's precision.

- `call/1` is necessarily hard to reason about statically and hard to implement in a compiler. Advanced type and mode analysis techniques might be able to convert some cases of `call/1` to more benign primitive constructs, but a region-based compiler will probably always need a very conservative (and inefficient) fall-back option to duplicate the intended interpretative semantics.

- `assert/1` and `retract/1`. Allowing arbitrary run-time changes to the program can of course disrupt any kind of static reasoning. If database operations are restricted to simple facts, however, they could not only be tolerated but actually aided by region-based memory management: Implementations often need to copy asserted facts to a separate "persistent heap" lest they get deallocated by backtracking before a database query that retrieves them. In a region-based implementation the region inference may arrange for every value that might end up being used in an asserted fact to be allocated on the persistent heap from the beginning, modeled by a special pseudo-region. That way time-consuming copying of asserted facts can be reduced.

- `findall/3` and similar predicates that communicate data across backtracking in structured ways. Again, region-based memory management may optimize the implementation of these if the mode analysis discovers that the results do not contain uninstantiated or trailed variables. Conventional implementations have to copy each result out of the heap so that it will not be destroyed by the backtracking that follows. A region-based implementation may simply (with some cooperation from the memory manager) exempt the region(s) containing the result from shrinking in that backtracking operation, thus still deallocating intermediate results that the computation may have left in other regions.

- *Constraints* would present difficulties for the region inference. A powerful mode analysis might be able to find a point where a constraint is sure to have been executed, so that the regions needed by it can be killed, but if the mode analysis gives up, each constraint would become a space leak. It is possible that there is a way around this, but more work it necessary to investigate it.

8. REFERENCES

[1] Alexander Aiken, Manuel Fähndrich, and Raph Levien. 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, 18–21 June 1995, La Jolla, CA, USA)*, special issue of *ACM SIGPLAN Notices*, 30(6):174–185. ⟨`http://http.cs.berkeley.edu/~aiken/ftp/region.ps`⟩.

[2] Yves Bekkers, Olivier Ridoux, and L. Ungaro. Dynamic memory management for sequential logic programming languages. In Yves Bekkers and J. Cohen (eds), *Memory Management (International Workshop, IWMM '92, 16–18 September 1992, St. Malo, France)*, volume 637 of *Lecture Notes in Computer Science*, pages 82–102. Springer-Verlag, Heidelberg, Germany, ISBN 3-540-55940-X.

[3] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Principles of Programming Languages (23rd ACM SIGPLAN-SIGACT Symposium, POPL '96, 21–24 January 1996, 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`⟩.

[4] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Principles of Programming Languages (26th ACM SIGPLAN-SIGACT Symposium, POPL '99, 20-22 January 1999, San Antonio, Texas, US)*, pages 262–275. ACM Press, New York, NY, USA, ISBN 1-58113-095-3. ⟨`http://simon.cs.cornell.edu/home/jgm/papers/capabilities.ps`⟩.

[5] Henning Makholm. *Region-Based Memory Management in Prolog.* Master's thesis, Department of Computer Science, University of Copenhagen, 2000. ⟨`ftp://ftp.diku.dk/diku/semantics/papers/D-421.ps.gz`⟩.

[6] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for PROLOG. *Articificial Intelligence*, 23(3):295–307, August 1984.

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

[8] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Hjfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit. Technical Report DIKU-TR-97/12, Department of Computer Science, University of Copenhagen, April 1997. ⟨`http://www.diku.dk/research-groups/topps/activities/kit2/diku97-12.a4.ps.gz`⟩.

[9] Mads Tofte and Jean-Pierre Talpin. 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, 1994, ISBN 0-89791-636-0. ⟨`ftp://ftp.diku.dk/diku/semantics/papers/D-235.dvi.gz`⟩.

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