Runtime Support for Region-Based Memory Management in Mercury

Quan Phan
Department of Computer Science,
K.U.Leuven,
Celestijnenlaan, 200A, B-3001 Leuven,
Belgium
quan.phan@cs.kuleuven.be

Zoltan Somogyi
National ICT Australia and
Department of Computer Science and
Software Engineering,
The University of Melbourne, Australia
zs@csse.unimelb.edu.au

Gerda Janssens
Department of Computer Science,
K.U.Leuven,
Celestijnenlaan, 200A, B-3001 Leuven,
Belgium
gerda.janssens@cs.kuleuven.be

Abstract

Applying region-based memory management (RBMM) to logic programming languages poses a special challenge: backtracking can require regions removed during forward execution to be "resurrected", and any memory allocated during a computation that has been backtracked over must be recovered promptly, without waiting for the regions involved to come to the end of their life. In this paper, we describe how we implemented runtime support for RBMM in the logic programming language Mercury, whose specialized implementation of the language constructs involved in backtracking required equally specialized support. Our benchmark Mercury programs run about 25% faster on average with RBMM than with the usual Boehm garbage collector, and for some programs, RBMM achieves optimal memory consumption.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers, Memory management

General Terms Languages, Performance

Keywords Region-based memory management, logic programming, Mercury

1. Introduction

Runtime garbage collection has become the standard approach to memory management in the implementation of modern programming languages: it provides memory safety, good memory reuse, and reasonable performance. In this approach, decisions about which parts of memory can be reused are made completely at runtime, which can incur significant overheads.

Region-based memory management [13] is a recent technique for avoiding these overheads. It is based on the idea of grouping heap objects having the same lifetime into regions; reclaiming entire regions at the end of their lifetime makes collection very fast. All the decisions about which objects are allocated into which regions and when the regions should be created and removed are made at compile-time. Since the pioneering work about RBMM for functional programming (specifically SML) [13], there have been several improvements and new developments in that context [1; 2; 6], and the idea has also been adapted to object-oriented [4] and logic programming [8; 10] languages.

In [10], a static region analysis was developed for the pure logic programming language Mercury. This program analysis detects the regions which will be used by a program, and decides when they will be created and removed based on their liveness. It then transforms the original program by annotating it with the inferred information. The memory consumption of the region-annotated programs reported in [10] showed that RBMM can actually reclaim garbage early. However, this information was collected by using a region simulator because the runtime system of Mercury had not been extended to support the annotated programs.

In this paper we describe the design and implementation of the runtime support needed by such annotated programs in the runtime system of Mercury. The main challenge is to deal correctly and efficiently with backtracking, a feature unique to logic programming, without seriously affecting the performance of deterministic programs. Moreover, unlike Prolog systems, the Mercury implementation generates highly specialized code for several program constructs, which means that runtime support for RBMM has to be specialized too, both to fit into the Mercury implementation and to ensure that the efficiency of Mercury programs is maintained and even improved.

In Section 2 we briefly introduce Mercury, and show how the RBMM program analysis in [10] annotates Mercury programs with region information. Section 3 shows the basic extensions to the runtime system needed to support RBMM in deterministic programs, while Section 4 describes our handling of backtracking. Section 5 gives our experimental results. Section 6 concludes with comparisons to related work.

2. Background

2.1 Mercury

Mercury is a pure logic programming language intended for the creation of large, fast, reliable programs [11]. While the syntax of Mercury is based on the syntax of Prolog, semantically the two languages are very different due to Mercury's purity, its type, mode, determinism and module systems, and its support for evaluable functions. (Mercury treats functions as predicates with the return value as an extra argument, so in the rest of the paper we will talk only about predicates.)

Mercury has a strong Hindley-Milner type system very similar to Haskell's. Mercury programs are statically typed; the compiler knows the type of every argument of every predicate (from declarations or inference) and every local variable (from inference).

The mode system classifies each argument of each predicate as either input or output; there are exceptions, but they are not relevant to this paper. If input, the argument passed by the caller must be a ground term. If output, the argument passed by the caller must be a distinct free variable, which the predicate will instantiate to a ground term. It is possible for a predicate to have more than one mode; the usual example is `append`, which has two principal modes: `append(in,in,out)` and `append(out,out,in)`. We call each mode of a predicate a *procedure*. The Mercury compiler generates separate code for each procedure.

Each procedure has a determinism, which puts limits on the number of its possible solutions. Procedures with determinism *det* succeed exactly once; *semidet* procedures succeed at most once; *multi* procedures succeed at least once; while *nondet* procedures may succeed any number of times.

The subset of Mercury [9] we deal with in this paper does not support higher order programming (including typeclasses), or predicates and functions defined by foreign language code. The reason is that these language constructs present some challenges for region analysis, and the region analysis in [10] does not yet handle them. However, the distinction between the full language and the subset does not matter for this paper, since the language constructs we omit do not impose any new requirements on the runtime system or on the compiler's code generator.

2.2 Mercury Code inside the Compiler

The compiler converts all predicate definitions into an internal form. For our subset of Mercury, this internal form is given by this abstract syntax, in which a sequence of goals separated by commas is a conjunction, while a sequence of goals separated by semicolons is a disjunction:

$$\begin{array}{ll} \text{predicate } P: & p(x_1,\ldots,x_n) \ \leftarrow\ G \mid f(x_1,\ldots,x_n) = r \ \leftarrow\ G \\ \text{goal } G: & x = y \mid x = f(y_1,\ldots,y_n) \mid p(x_1,\ldots,x_n) \mid \\ & (G_1,\cdots,G_n) \mid (G_1;\ldots;G_n) \mid \mathit{not}\ G \mid \\ & (\mathit{if}\ G_c\ \mathit{then}\ G_t\ \mathit{else}\ G_e) \mid \mathit{some}\ [x_1,\ldots,x_n]\ G \end{array}$$

As this shows, the Mercury compiler internally converts any predicate definition with two or more clauses into a single clause with an explicit disjunction. The clause bodies themselves are transformed into *superhomogeneous form*, in which each atom (including clause heads) must be of one of the forms `p(X1,...,Xn)`, `Y = X`, or `Y = f(X1,...,Xn)`, where all of the `Xi` are distinct.

Inside the compiler, every goal (compound goals as well as calls and unifications) is annotated with mode and determinism information. For unifications, we show the mode information by writing `<=` for construction unifications, `=>` for deconstruction unifications, `==` for equality tests, and `:=` for assignments. The compiler reorders conjunctions as needed to ensure that goals that consume the value of a variable always follow the goal that produces its value. We show a Mercury program in this abstract syntax in Figure 1. This artificial program handles various lists of integers. It has no intuitive meaning, but it does illustrate the various kinds of interactions between regions and backtracking that we need to handle. We will use it as our running example. (The !IOs represent the state of the world being threaded through. For completeness, we include the definitions of `member` and `length` whose behaviour, as we will see, is of no importance to RBMM.)

In the rest of the paper, we will ignore negation, since `not` G can be implemented as `if` G `then fail else true`. Note that in Mercury (unlike in Prolog), the condition of an if-then-else may succeed several times. This will be clear from the determinism annotation on the goal representing the condition, and many parts of the compiler, including the implementation of RBMM, handle conditions of different determinisms differently.

Another situation in which determinism information is important is existential quantification. (Mercury also supports universal quantification, but the compiler internally converts $\mathit{all}\ [x_1,\ \ldots,\ x_n]\ G$ to $\mathit{not\ some}\ [x_1,\ \ldots,\ x_n]\ \mathit{not}\ G$, so we do not have to deal with it.) If $\mathit{some}\ [\ldots]\ G$ quantifies away all

```
main(!IO) :-                        % mode(in, in), semidet
(1) X <= [1, 3, -1, 3],             member(X, L) :-
(2) A <= [-2],                        L => [H | T],
(3) ( if p(X, A, B, Y) then           (H == X ; member(X, T)).
(4)     io.write(B, !IO),
(5)     io.write(Y, !IO)            % mode(in) = out, det.
      else                          length(L) = N :-
(6)     io.write(X, !IO),             ( L == [], N := 0
(7)     io.write(A, !IO)              ; L => [_ | T],
    ).                                  N := length(T) + 1
                                      ).

:- pred p(list(int), list(int), list(int), list(int)).
:- mode p(in, in, out, out) is semidet.
p(X, U, V, Y) :-
(1) X => [H | T],
(2) ( if H < 0 then
(3)     Y <= [H],
(4)     ( if   member(H, U)
(5)       then  V := U
(6)       else  V <= [H | U]
        )
      else
(7)     p(T, U, V, Y1),
(8)     ( if length(V) > length(Y1)
(9)       then  fail
(10)      else  Y <= [H | Y1]
        )
    ).
```

Figure 1. Our running example.

the output variables of G, then different solutions of G would be indistinguishable, so even if G can have more than one solution, $\mathit{some}\ [\ldots]\ G$ will not. We call such a quantification a *commit*, and we handle commits differently from other quantifications.

2.3 Memory Management for Mercury

Mercury allocates memory for structured terms from the heap. The heap has traditionally been managed by the Boehm-Demers-Weiser conservative garbage collector for C [3], since Mercury compiles to C. Using the Boehm collector is very convenient and it works well in practice, but it does have drawbacks.

The main drawback is that allocating memory requires a function call. Mercury programs, like those written in other declarative languages, allocate memory at a much higher rate than imperative language programs (due to the absence of destructive updating), so this is a significant source of overhead.

A second drawback is the absence of instant reclaiming. Prolog implementations can simply remember the heap pointer when entering a disjunction and restore it when backtracking to any of the later disjuncts, instantly reclaiming all the memory allocated since execution entered the previous disjunct. The Boehm collector cannot do this, since it does not allocate memory cells in chronological order. (One could link cells together in this order, but the resulting overhead would vastly outweigh the potential gain.)

The paper [5] proposed a solution to these drawbacks: a native garbage collector for Mercury. It allocated memory with a simple increment of the heap pointer, and allowed instant reclaiming by saving and restoring the heap pointer. Benchmarks proved that this approach could achieve significant speedups over the Boehm collector. Unfortunately it could also yield slowdowns, because the system had to preserve enough information to allow the collector to reconstruct the types of all values on the heap, since without this information, pointers cannot be correctly identified and traced, and because the engineering of the collector itself could never catch up with the amount of effort put into tuning the Boehm collector.

With region-based memory management, we do not need to preserve type information, yet allocating and collecting memory are still very simple and fast. As we will see later, we do incur some other kinds of overhead, but we expect that these will be significantly smaller than the benefits we get from fast allocation and collection (including instant reclaiming) will give us overall speedups. Our experimental data bears this out.

2.4 Region-Based Memory Management for Mercury

In logic programming languages, the existence of backtracking requires the notion of liveness to be divided into two parts. A variable, memory location, region and so on is *forward live* at a program point if it can be accessed during forward execution from that program point, and it is *backward live* at a program point if it can be accessed in backward execution (e.g., after backtracking to a choice point established before that program point). The two notions of liveness are independent: all four combinations of forward and backward liveness and deadness are possible.

The region analysis and transformation described in [10] has been implemented in the Mercury compiler. It analyzes programs (so far, only programs that fit into one module), decides which region to make each allocation in and when each region should be created and removed, and annotates the program with those decisions. This analysis considers only forward liveness. At the program point at which a region becomes forward dead, it will insert an instruction to reclaim that region's memory. Figure 2 shows a version of our running example from Figure 1 after this transformation.

```
main(!IO) :-
    create(R1),
(1) X <= [1, 3, -1, 3] in R1,
    create(R2),
(2) A <= [-2] in R2,
    ( if
        create(R3),
(3)     p(X@R1, A@R2, B@R2, Y@R3)
      then
(4)     io.write(B, !IO),
        remove(R2),
(5)     io.write(Y, !IO),
        remove(R3)
      else
(6)     io.write(X, !IO),
        remove(R1),
(7)     io.write(A, !IO),
        remove(R2)
    ).

p(X@R4, U@R5, V@R5, Y@R6) :-
(1) X => [H | T],
    ( if
        remove(R4),
(2)     H < 0
      then
(3)     Y  <= [H] in R6,
(4)     ( if    member(H, U@R5)
(5)       then  V := U
(6)       else  V <= [H | U] in R5
        )
      else
(7)     p(T@R4, U@R5, V@R5, Y1@R6),
(8)     ( if    length(V@R5) > length(Y1@R6)
(9)       then  fail
(10)      else  Y <= [H | Y1] in R6
        )
    ).

member(X, L@R7) :-
  L => [H | T],
  (H == X ; member(X, T@R7)).

length(L@R8) = N :-
  (
    L == [], N := 0
  ; L => [_ | T],
    N := length(T@R8) + 1
  ).
```

Figure 2. Region-annotated version of the program.

The instruction `create(R)` binds the region variable `R` to a newly created region, while the instruction `remove(R)` reclaims the region to which `R` is bound. Regions can (and usually do) live across procedure boundaries, and region variables are passed as extra parameters in procedure calls.

In our example code, we use the postfix `@Ri` to annotate both actual and formal parameters with their region variables. We also annotate each unification that constructs a new memory cell with the region from which the cell will be allocated. For example, in `main`, the skeleton of the list `X` is in region `R1`, that of the list `A` is in `R2`; the skeletons of the lists generated by the call to `p`, `B` and `Y`, are in `R2` and `R3`, respectively. Note that there are no regions for the elements of the lists because they are of type `int`, which is a primitive type in Mercury, and they are stored right in the first words of the *cons* cells in the skeletons. In the call to `p` in the condition of the if-then-else, `R1`, `R2`, and `R3` are passed as actual region parameters, corresponding to the formal parameters `R4`, `R5`, and `R6` in the definition of `p`. (`R2` and `R5` are duplicated only for exposition.)

Regarding the lifetime of the regions, in `main`, `R1` and `R2` are created before the construction of the two lists `X` and `A`. `main` creates `R3` before the call to `p` at (3), and `p` will use this region to store the skeleton of `Y`. All the `remove` instructions for regions are added after the last *forward* uses of the terms stored in them. `member` and `length` will only read from their input regions.

To execute the region-annotated programs, the runtime system of Mercury needs to be enhanced to work with the heap memory organized in terms of regions, and to provide support for backtracking in the context of RBMM, including instant reclaiming. We will discuss this necessity in detail in Section 4.

3. Runtime Support for Regions in Deterministic Programs

In this section we describe our implementation of regions and of the region management operations needed for deterministic programs. This aspect of our implementation is generally similar to the "standard" RBMM implementations for SML and Prolog, which are described in detail in [7; 8].

In our system, a region is a singly linked list of fixed-size region pages. Each region page has a *data area*, an array of words that can be used to store program data, and a pointer to the next region page to form the single-linked list. (In the newest region page of a region, this pointer is of course null.) The *handle* of the region (the way the rest of the system refers to it) is the address of the *region header*. Besides some other fields that we will introduce later, the header structure includes a *region size record*: a pointer to the newest region page, and a pointer to the next available word in the newest region page. Since region pages have a fixed size, these two values implicitly also specify the amount of free space in the newest region page. To avoid fragmentation, we store each region header at the start of the data area of its region's *first* region page. Figure 3 shows a region with two region pages; the shaded areas represent memory allocated to user data.

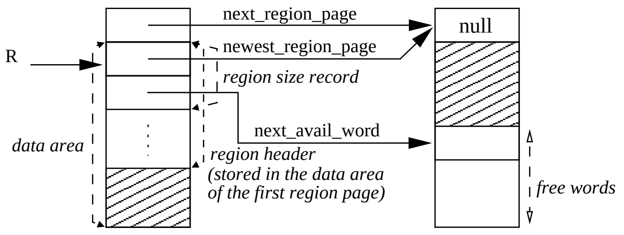


Figure 3. The data structure of a region `R`.

There is no bound on region size. When a *region is created* it will contain only one region page, but it can be extended by adding more region pages when necessary. The program maintains a global list of free region pages. If the free list runs out, the program requests a big chunk of memory from the operating system, divides it into region pages, and adds them to the free list. When a region needs to be extended, we take a region page from the free list and add it to the region as its new last region page, and then update the region's size record. When a *region is reclaimed*, we return all its region pages to the free list. An *allocation into a region* always happens in its newest region page simply by increasing the pointer to the next available word. When the free amount in this region page is not enough for an allocation, we extend the region before allocating.

The advantage of this implementation is that the basic region management actions are bounded in time. Disadvantages are that there is no natural size for the region pages [12], and that if the remaining space of a region page is not enough for an allocation, that space will be wasted when a new region page is added.

4. Runtime Support for Backtracking

Backtracking introduces two issues that need to be handled: reclaiming the memory allocated by the computations backtracked over, and ensuring that regions are reclaimed only when they are dead with respect to both forward and backward execution. The first issue obviously has to be handled at runtime. For our initial implementation, we have chosen to deal with the second issue, backward liveness, in the runtime system too. We expect this to give us the insights we will need later to redesign the program analysis to handle backward liveness both safely and precisely. Moreover, our current system can serve as a reference for that work.

In Mercury, disjunctions are the main source of backtracking because they provide alternatives. But an if-then-else is just a special kind of disjunction: $(if\ C\ then\ T\ else\ E)$ is semantically equivalent to $(C, T; not\ some\ [\cdots]\ C, E)$. Operationally, Mercury will try C. If C succeeds, Mercury executes T; if C fails, it executes E as if C had never been tried. The handling of commit (Section 2.2) is related to the handling of backtracking because committing to a solution may prune some alternatives of relevant disjunctions. Therefore, we need to provide runtime support for backtracking in the context of these three language constructs.

Our running example in Figure 2 illustrates our two tasks.

Task 1: Preventing the reclamation of backward live regions. The condition of the if-then-else in the `main` predicate is the call to the semidet procedure `p`. The RBMM transformation marks the region `R1` for removal in the call because it is forward dead (it is not used in the then part) even though it is backward live (it *is* used in the else part). We must make sure that `R1` is not actually removed while it is backward live. In this case, that means we need to delay the reclamation of `R1` until we reach the then part; it is not safe to actually destroy `R1` if the condition fails. We therefore distinguish *reclaiming* a region, which makes the memory of the region available for other uses and thus potentially destroys its contents, from the operation of *removing* a region, which causes the region to be reclaimed only when it is safe to do so. Basically, a region is removed when it is forward dead, and it is reclaimed when it is both forward and backward dead.

Task 2: Reclaiming the memory used by backtracked-over computations. The call to `p` has two output arguments, `B` and `Y`. `main` tells `p` to put any cells for `B` in `R2`. The condition is extended with a `create` instruction to create `R3` so that `p` can put `Y` into it. If the condition succeeds, we must leave both regions alone. If the condition fails, we should restore `R2` to its size before the condition, and we should reclaim `R3` in its entirety.

We now define several runtime concepts that we will use in the rest of the paper.

Old vs new regions. A region is *old* with respect to a point during the execution of a program if it was created before that point, otherwise it is *new* with respect to that point. We also refer to old regions as the *existing regions*. To allow efficient checks whether a region is old or new, we maintain a global *region sequence number* counter (starting at one) and include a `sequence_number` field in region headers. When we create a region, we timestamp it by setting its `sequence_number` from the global counter, and increment the counter. If at a point during the execution of the program (such as a resumption point where the program resumes when backtracking) we save the current sequence number, then all the regions which are created before that point, i.e., the old regions with respect to the point, will have their sequence numbers smaller than the saved value; ones which are created after that point, i.e., the new regions with respect to the point, will have their sequence numbers greater than or equal to the saved value. When the program backtracks to the save point, we can use the saved value to check whether a region has been created before or after the point. In the context of RBMM, the memory that we want to reclaim at a resumption point will be new allocations into existing regions, and new regions in their entirety (since they have been created by the computation we have just backtracked over).

Region list. To do instant reclaiming of new regions, knowing the sequence numbers of the new regions is not enough; we also need to *reach* them. We therefore link all the live regions into a doubly-linked *region list* (using two additional pointers in the region header). We maintain a global pointer to the head of the list, which will be the newest live region. When a region is created we add it to the head of the region list; when a region is reclaimed we remove it from the list. We maintain the invariant that the region list is ordered by regions' creation time, newest first. To reclaim new regions, we can traverse the region list from its head and reclaim each region until we meet an old one.

Region size snapshots. To do instant reclaiming of new allocations into an existing region, we need the old size of the region. When we need to remember the size of a region at a point, we can save its region size record at that point.

Protection. We will prevent the destruction of backward live regions by *protecting* them so that when a removal happens to the region during forward execution, the removal will be ignored.

Changes to live regions by a goal. When providing support for backtracking, sometimes we want to know about the changes which may be caused by a goal to the set of regions the goal may refer to. This means we need to know about any new regions the goal creates, any live regions the goal removes, and any live regions in which the goal performs allocations. We refer to these sets of regions as the goal's *created*, *removed*, and *allocated* sets, respectively. The region analysis of [10] computes, for each procedure, the set of regions which are input to it ($inputR$), the set of regions which it will create ($bornR$) and the set of regions it will removed ($deadR$). The *created*, *removed*, and *allocated* sets of goals can be computed from this in a fairly straightforward manner.

Changes to live regions by a goal: creation. Only `create` instructions and procedure calls may create regions. A `create` instruction always creates the region in its argument. A procedure call will create the regions that are the actual region parameters corresponding to the formal parameters in the $bornR$ set of the called procedure. For a compound goal, its created set is the set of all regions created inside it, *including* those that are also removed by it.

Changes to live regions by a goal: removal. We can similarly use `remove` instructions and the $deadR$ sets of procedures to compute the removed set of each goal. Since we only care about the old regions which are removed inside a goal, we exclude regions created inside the goal (i.e., the goal's created set) from its removed set.

Changes to live regions by a goal: allocation. A region is allocated into in construction unifications and procedure calls. A construction unification will allocate into the region with which it is annotated. A procedure call may allocate into any region in its input set ($inputR$). A program analysis could find out which subset is actually allocated into, but the implementation of such an analysis would be complicated, especially for multi-module programs. Instead, we use the conservative approximation and assume that a call may allocate into any region in its $inputR$ set.

Changes to live regions by a goal: an example. Take the condition of the if-then-else in the procedure `p` in Figure 2 as an example goal. We say that the region `R4` is removed in the condition because `R4` is live before the condition and `remove(R4)` has been added to the condition. Or take the condition of the if-then-else in `main`. We say region `R3` is created in the condition because `create(R3)` has been inserted into the condition, while region `R1` is removed in the condition because it is live before the condition and is removed in the call to `p`. We assume that both regions `R1` and `R2` can be allocated into during the condition (the call to `p`); since, in fact, `p` may allocate only into `R2`, this is a safe (if imprecise) assumption.

We provide the runtime support for backtracking for a program by generating extra supporting code at the right places to achieve our goals. In the next three subsections we will describe in detail the support for disjunctions, if-then-elses, and commits.

4.1 Support for disjunction

Although the Mercury language does not specify the language's search strategy, the Mercury compiler supports only one search strategy: depth-first search with chronological backtracking, so that the disjuncts of each disjunction are tried in order. Given a disjunction `(g1; ...; gi; ...; gn)`, we refer to `g1` as the first disjunct, to the `gi`s for all $1 < i < n$ as middle disjuncts, and to `gn` as the last disjunct of the disjunction. We will also use "later disjunct" to refer to any `gi` for $i > 1$.

A disjunction can have any determinism. The most general determinism is of course nondet, but if one of the disjuncts always has at least one solution, then the disjunction as a whole does too, so a disjunction can also be multi. And if the disjunction has no outputs (which happens frequently for disjunctions in the conditions of if-then-elses), then the disjunction as a whole cannot have more than one solution, which means that it will be either det or semidet, depending on whether it has an always-succeeding disjunct. (Typical programs do not contain det disjunctions, since they are equivalent to `true`.)

For our purposes, the important distinction is between nondet and multi disjunctions on the one hand, in which backtracking may reach a later disjunct from code executed outside the disjunction, *after* the success of a previous disjunct, and semidet and det disjunctions on the other hand, in which backtracking to a later disjunct is possible only from code *within* an earlier disjunct. Since we do not care about the minimum number of solutions of each disjunction, our support treats multi disjunctions the same as nondet ones and det disjunctions the same as semidet ones. In the following, we will therefore talk only about nondet and semidet disjunctions. We consider nondet disjunctions first, since they are more general.

Figure 4 shows in pseudo-code form the supporting code we added to a nondet disjunction. We insert code at the following points: (d1) which is the start of the first disjunct, (d2) which represents the start of every middle disjunct, and (d3) which is the start of the last disjunct. These code fragments communicate using shared data in what we call a *disj frame*. Each entry to a disjunction creates a new disj frame. Since multiple nested disjunctions can be active at the same time, we link these frames together to form the *disj stack* (this is possible due to chronological backtracking). The disj stack is not a separate stack; we reserve space for its frames in the usual stacks used by the Mercury language implementation. We maintain a global pointer to the top disj frame on the disj stack.

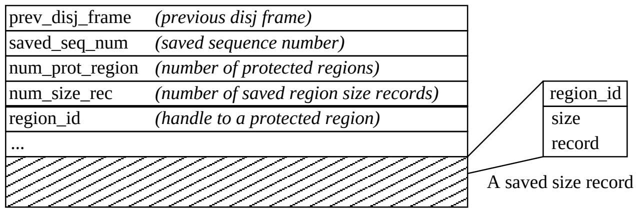
A disj frame has a fixed part and a nonfixed part. In Figure 5, the fixed part is the 4-slot box separated by a thick line from the nonfixed part. The four slots in the fixed part are:

- The `prev_disj_frame` slot holds the pointer to the previous disj frame, (or null if there is none).

```
...,
( (d1): start of the disjunction and also of the first disjunct
        (a) push a disj frame
        (b) save the global region sequence number
        (c) save region size records and their number
    g1
; ...
; (d2): start of a middle disjunct
        (a) do instant reclaiming of new regions
        (b) do instant reclaiming of allocations in old regions
    gi
; ...
; (d3): start of the last disjunct
        (a) do instant reclaiming of new regions
        (b) do instant reclaiming of allocations in old regions
        (c) pop the disj frame
    gn
), ...
```

Figure 4. RBMM runtime support for nondet disjunctions.

- The `saved_seq_num` slot holds the value of the global region sequence number at the time when the disjunction was entered.
- The `num_prot_region` field gives the number of regions which are protected by a semidet disjunction (which we will discuss later). For a nondet disjunction, this slot will contain zero.
- The `num_size_rec` field gives the number of region size records saved in the nonfixed part.



prev_disj_frame (previous disj frame)
saved_seq_num (saved sequence number)
num_prot_region (number of protected regions)
num_size_rec (number of saved region size records)
region_id (handle to a protected region)
...
region_id
size
record
A saved size record

Figure 5. The structure of a disj frame.

Disj-protecting backward live regions. Consider a region which was created before the execution of a disjunction. Assume that this region is removed during forward execution, either by the code of a disjunct, or after the success of that disjunct by code following and outside the disjunction, but that this region is backward live with respect to a later disjunct of the disjunction. In this case, we need to make sure that if the region is removed during forward execution, it will not be actually reclaimed. Of course, the instruction that removes the region may not be reached because forward execution may fail before it gets there. But in general, we have to assume that the `remove` instruction *will* be executed, and that if the region may be needed after backtracking, we will need to prevent it from being reclaimed during the forward execution. We achieve this by *disj-protecting* such regions as follows. At the start of the disjunction, i.e., at (d1), we push a disj frame on the disj stack and save the current global sequence number into the `saved_seq_num` slot of the disj frame. A region is disj-protected by a disj frame if its sequence number is smaller than the sequence number saved at the disj frame. The `remove` instruction will only reclaim a region if the region is not disj-protected. There is an invariant that if a region is protected by a disj frame, it is also protected by all the later frames on the disj stack. This means that to check if a region is disj-protected or not, we only need to check if it is protected by the top disj frame.

The program will no longer backtrack into a disjunction after starting the execution of its last disjunct. This means that no regions need to be protected any more by this disjunction. Therefore, at the start of the last disjunct, i.e., at (d3), we disj-unprotect them by popping the disj frame. The regions which had been previously been protected only by this disj frame will be reclaimed when execution reaches their `remove` instructions.

Instant reclaiming of new regions. When the program backtracks to a later disjunct, we want to reclaim all the regions that have been created during the computation that has just been backtracked over, i.e., all the regions that were created after entry to the disjunction. At (d1), we saved the global sequence number in the disj frame. Therefore at the start of a later disjunct of the disjunction, i.e., at (d2) and (d3), we just need to traverse the region list, and reclaim all the regions we see until we encounter a region whose sequence number indicates that it was created before the disj frame.

Instant reclaiming of new allocations in old regions. When arriving at a later disjunct, we want to restore all the regions that existed before the disjunction to the sizes they had when entering the disjunction, recovering any memory that has been allocated in them. To restore the size of a region, we need to save the region's size record in the nonfixed part of the disjunction's disj frame at (d1) so that we can restore the region's size at (d2) and (d3). We need three slots for each region: one for the region handle so that we know to which region the saved record belongs, and the other two for the record itself (see Figure 5). To be able to loop through the saved records and restore the regions at (d2) and (d3), we store the number of saved records in the fixed `num_size_rec` slot. The first saved record can be located by taking the address of the frame, and adding both the size of the fixed part and the number of slots for protected regions (which is zero in this case of nondet disjunctions).

The set of regions that existed before the disjunction and that may be allocated into by code following the disjunction is not available to the compiler. In theory, we could implement a global analysis to make it available, but such an analysis would be very complicated, especially for multi-module programs. Even if such an analysis existed, we would still have a big problem, which is that the number of regions in this set is not bounded, and in many cases the set would contain tens, hundreds or even thousands of regions. Saving and then restoring the sizes of that many regions can take a significant amount of both memory and time. We do not want this overhead to outweigh the benefits of instant reclaiming.

In our implementation, we have chosen to save and restore the sizes of only the regions that are *locally* forward live before the disjunction. (This information is readily available inside the Mercury compiler.) This may lead to some missed opportunities for recovering memory, but since nondet disjunctions are quite rare in most Mercury programs, we do not expect this to be too much of a problem. (We will see below that we do not miss memory recovery opportunities for semidet disjunctions.)

We save and restore the sizes of *all* regions that are locally forward live at the start of the disjunction (the number of these regions governs how much space we reserve for the nonfixed part of the disj frame). We save and restore the sizes even of regions that are never allocated into before backtracking, since (in the absence of the analysis mentioned above) we do not know which ones of those are. This may lead to some unnecessary saving and restoring, but since in typical programs the number of regions whose size we save and restore at a disjunction is usually relatively small, we do not expect the memory or runtime cost of these unnecessary saves and restores to be all that significant.

Specialized treatment of semidet disjunctions. Because at most one disjunct of a semidet disjunction may succeed, when one of its disjuncts is reached, it means that all the previous disjuncts have failed and more importantly that the execution has not passed outside the disjunction's scope. Therefore, we only need to provide runtime support for a semidet disjunction if in its scope there is some change with respect to the set of existing regions. This basically means that the runtime support for nondet disjunctions described above will only be applied to semidet disjunctions whose created, removed and allocated sets are not all empty. In our practical experience with Mercury, most semidet disjunctions contain only tests, and rarely make changes to the heap. Therefore the support we describe below is needed only by a relatively small fraction of semidet disjunctions.

For a semidet disjunction, the Mercury compiler generates code such that when one of its non-last disjunct succeeds, the execution will commit to it and not go back to try any later disjuncts. This means the code we add at (d3) may not be reached after the success of a non-last disjunct, causing two problems. First, the disj frame will not be popped. Second, the regions which are protected by this disjunction will not be reclaimed at the start of the execution of the last disjunct as in the case of nondet disjunctions. Our solution is to do these two tasks at the end of any non-last disjuncts, i.e., after their success at (e1) and (e2) as in Figure 6.

```
...,
( (d1): start of the disjunction and of the first disjunct
        (a) push a disj frame
        (b) save the global region sequence number
        (c) save region size records and their number
        (d) save protected regions and their number
    g1
  (e1): end of the first disjunct
        (a) reclaim protected regions
        (b) pop the disj frame
; ...
; (d2): start of a middle disjunct
        (a) do instant reclaiming of new regions
        (b) do instant reclaiming of allocations in old regions
    gi
  (e2): end of a middle disjunct
        (a) reclaim protected regions
        (b) pop the disj frame
; ...
; (d3): start of the last disjunct
        (a) do instant reclaiming of new regions
        (b) do instant reclaiming of allocations in old regions
        (c) pop the disj frame
    gn
), ...
```

Figure 6. RBMM runtime support for semidet disjunction.

To solve the first problem, we pop the frame at (e1.b) and (e2.b). To solve the second problem, at (d1) we loop through the regions in the disjunction's removed set. If a region is already protected, we do not want it to be reclaimed in the disjunction and its `remove` instructions inside the disjunction will be ineffective anyway, so we do not need to do anything. If a region is not already protected, we save its handle in the nonfixed part of the disj frame. At the end, we store the number of region handles we saved in the frame's `num_prot_region` slot. The code at (e1.a) and (e2.a) will loop through the saved handles, and reclaim all the saved regions (they were logically removed during the disjunct but their `remove` instructions were thwarted by the protection of this disjunction.)

At (d1.c), we save the sizes of only the regions in the disjunction's allocated set. Since execution cannot leave a semidet disjunction, we do not miss any memory recovery opportunities by restricting ourselves to these regions.

4.2 Support for if-then-else

The condition of an if-then-else (ite) can be either semidet or nondet. In most Mercury programs, the overwhelming majority are semidet, and this is the case we will look at first. Such if-then-elses share some properties with semidet disjunctions. If the condition succeeds, the execution will never enter the else part, and if the condition fails, the failure must have occurred in the scope of the condition.

Like disjunctions, if-then-elses need to protect regions from being reclaimed while backward live. But in the case of if-then-elses, we can restrict out attention to regions removed in the condition (i.e., in the condition's removed set), since this is the only part of the code in which the if-then-else itself can make a region backward live. When execution reaches the start of the then part, backtracking to the else part is no longer possible, which means that any regions that have been marked for removal in the condition have to be reclaimed for real, unless they are protected by a surrounding scope.

Also, if-then-elses, like disjunctions, should do instant reclaiming of memory allocated by backtracked-over computations. In the case of if-then-elses, this means that at the start of the else part, we should recover any memory allocated by the condition.

In general, we only need to provide support for changes to regions which occur inside the condition. This is good, because the conditions of if-then-elses are often very simple, containing only one or a few tests. Conditions whose created, removed and allocated sets are all empty are therefore fairly common. For such if-then-elses, the mechanisms we describe below are unnecessary, and so we optimize them away. If at least one these three sets is not empty, we add code at the starts of the condition, the then part, and the else part, i.e., at points (i1), (i2), and (i3) in Figure 7.

```
( if
    (i1): start of the condition
        (a) push an ite frame
        (b) save the protected regions and their number
        (c) save size records and their number
    ...
  then
    (i2): start of the then part
        (a) reclaim the ite-protected regions
        (b) pop the ite frame
    ...
  else
    (i3): start of the else part
        (a) unprotect the ite-protected regions
        (b) do instant reclaiming of new regions
        (c) do instant reclaiming of allocations in old regions
        (d) pop the ite frame
    ...
)
```

Figure 7. RBMM runtime support for if-then-else with semidet condition.

For each if-then-else, we use a data structure called an *ite frame* to store the information used for its runtime support. As with disj frames, we embed ite frames in the ordinary stacks used by the Mercury implementation, and link them together into the *ite stack*, with a global variable pointing to its top. The structure of an ite frame is exactly analogous to that of a disj frame, the only difference being that the first slot of the fixed part, `prev_ite_frame`, holds a pointer to the previous ite frame (or null if there is none).

Ite-protecting backward live regions. Since the compiler knows the regions in the removed set of the condition (in our example, `R1` is such a region), we will stop them from being reclaimed by *ite-protecting* them at the entry to the if-then-else. To allow us to ite-protect regions, we add to the region header a pointer field, `ite_protected`, which is set to null when a region is created. A region is ite-protected if its `ite_protected` field is not null. The `remove` instruction will now only reclaim a region if its `ite_protected` field is null and it is not disj-protected. (We do not use the same protection mechanism as in the case of disjunction. The reason for this will be explained when we describe how we handle if-then-elses with nondet conditions.) Before entering the condition, i.e., at (i1), we push an ite frame, and then iterate over the to-be-protected regions. If one of these regions is already protected for a surrounding disjunction or if-then-else, we ignore it. Otherwise, we protect it by setting its `ite_protected` field, which is currently null, to point to the ite frame. For such a protected region, we add its handle to a `region_id` slot in the nonfixed part of the ite frame. Then we also put the final number of regions we protect in this way into the frame's `num_prot_region` slot. We do this so that we can loop over all the regions protected by this ite frame in two places: at the start of the then part (i2.a), where we reclaim all these regions (giving delayed effect to the `remove` instructions in the condition), and at the start of the else part (i3.a), where we undo their protection by resetting their `ite_protected` fields to null.

Instant reclaiming. When the condition fails, we want to reclaim both the new regions created inside it and any new allocations into old regions. In our example in Figure 2 we want to reclaim all of `R3` and some of `R2`.

To reclaim new regions, at (i1.a) we save the current sequence number into the new frame's `saved_seq_num` slot, and at (i3.b), we add code that traverses the region list and reclaims all the regions until meeting an old region.

To reclaim new allocations into an old region, at (i1.c) we save its size record into the nonfixed part of the ite frame. Although it is reasonable to do this for the regions in the allocated set of the condition, it would be wasteful to reclaim new allocations into the regions which will be reclaimed right at the start of the else part. Unfortunately, while the compiler knows which old regions have `remove` instructions at the start of the else part, it does not know which of these will actually reclaim their regions, since it does not know which regions are protected by surrounding code. We handle this uncertainty as follows. We generate code at (i1.c) for every old region which is live at that point. For those that are not removed at the start of the else branch, this code always saves their size records unconditionally. For those that are removed at the start of the else branch, this code checks whether they are protected, and saves their size records only if they are. This is an optimization because the test to see if a region is protected takes less time than saving its size record, and restoring it if the condition fails. We record the number of size records we saved in the `num_size_record` slot, so that code at (i3.c) can restore them all.

The final action of an if-then-else with a semidet condition is to pop the ite frame at either (i2.b) or (i3.d).

if-then-else with nondet condition. Unlike Prolog, Mercury allows the condition of an if-then-else to have more than one solution. If the condition is nondet, then execution can backtrack into the condition from the then part or later code. This poses two problems we need to solve.

First, since the condition can succeed more than once, the code we add at the start of the then part (i2) can also be executed more than once. Because we need the ite frame every one of these times, we cannot let the code pop it at (i2.b); we must keep it until after the last time it may be used, i.e., after the last execution of the condition. We arrange for this to happen by modifying the way the code generator handles the failure of the condition.

Normally, the code generator arranges for failures of the condition *before* the condition succeeds for the first time to cause a branch to the start of the else part, while a failure of the condition *after* it has succeeded represents a failure of the if-then-else as a whole, and will be handled accordingly, in whatever way the surrounding context demands. For example, if the if-then-else is one disjunct of a disjunction, its failure will cause execution to resume at the start of the next disjunct. We call the place to branch to on failure of the whole if-then-else the *failure continuation*.

We modified the code generator so that if the nondet condition needs support for region operations, i.e., it has a nonempty created set, removed set or allocated set, we branch to the failure continuation only after we execute code to pop the ite frame, the same code that for semidet conditions we would execute at (i2.b).

Second, the condition being nondet means that it must include, directly or indirectly, a nondet disjunction (since this is the only Mercury construct that can introduce nondeterminism). Therefore we must ensure that the supporting code fragments we generate for the if-then-else and the disjunction inside it do not step on each other's toes.

Our support for if-then-elses with semidet conditions provides ite-protection for regions in the condition's removed set that are not yet protected before the if-then-else. For such a region in a nondet condition, there are two cases. The first case is when the region is removed before the first nondet disjunction inside the condition. That means that when the `remove` instruction is executed, the region is ite-protected but not disj-protected. The `remove` instruction will (correctly) not reclaim it. Later on, the region will be reclaimed when the condition succeeds for the first time by the supporting code added at (i2). Because the program may backtrack into the condition and may reach the then part again, when the region is reclaimed at (i2.a), we need to nullify its entry in the ite frame so that it will not be wrongly reclaimed again the next time execution reaches (i2.a). This explains our saving of the pointer to the ite frame in the `ite_protected` field in the region header of a protected region.

In the second case, the region is removed after the start of the first disjunction in the condition, either in the disjunction itself or at some point after it. In an execution containing a non-last disjunct, when the `remove` instruction is encountered the region is not reclaimed because it is both ite- and disj-protected. We need to ensure that if the condition succeeds and execution reaches the then part, the region should not be reclaimed at (i2) because it may be needed when the execution goes back into the condition. We therefore put different code at (i2.a) if the condition is nondet; this code will reclaim a region only if it is not currently disj-protected (Figure 8). The region will remain both ite- and disj-protected until the execution enters the last disjunct, at that time it will lose its disj-protection (Section 4.1). When the `remove` instruction in the condition is executed after this, it will not reclaim the region because it is still ite-protected, but the code at (i2.a) will reclaim it.

```
for each saved region_id
    if region_id != null && !is_disj_protected(region_id)
            reclaim the region;
            region_id  = null
```

Figure 8. Code at (i2.a) for if-then-else with nondet condition.

When the nondet condition fails, in both cases above, the region is only ite-protected, not disj-protected. It is because in the first case, the region is never disj-protected and in the second case, the failure happens only after all the disjuncts of the nondet code have been tried and failed, and the region has been disj-unprotected at the start of the last disjunct. This situation is exactly the same as when a semidet condition fails. Therefore the code at (i3) is exactly the same for nondet conditions as for semidet conditions.

4.3 Support for commit

When the goal inside a commit succeeds for the first time, we commit to that solution by discarding the inner goal's outstanding alternatives. We call the point in the code where this happens the *commit point*. If the inner goal is nondet (rather than multi), it may also fail. When it fails, the compiler's failure-handling mechanism causes execution to pass through a *failure point* before the program resumes forward execution at the resumption point of the next surrounding goal. The failure point is there to allow the execution of some cleanup code. We add code to support region operations at two or three points in Figure 9: the entry point of the commit (c1), the commit point (c2), and the failure point (c3); if the inside goal has determinism multi, we do not modify (c3) as execution will never reach there.

Consider a region that is in the removed set of a commit goal. If it is already protected by a disjunction or if-then-else when execution arrives at (c1), then the region should not be reclaimed by any code inside the commit, and the mechanisms we have described so far are sufficient to ensure this. If the region is not already protected at (c1), then the region should be reclaimed before execution reaches (c2). Ensuring this needs a new mechanism because the goal inside a commit will contain, directly or indirectly, at least one disjunction that can succeed more than once (if it did not, it would have at most one solution, and there would be no commit operation), and the runtime support for this disjunction will protect the region from being reclaimed during the execution of its non-last disjuncts. On the other hand, we cannot simply insert code at (c2) to reclaim the region, since it *can* already be reclaimed by its `remove` instruction in the execution of the last disjunct before reaching (c2). We do not need to worry about the case when regions are protected only by semidet disjunctions or by if-then-elses inside a commit, since these constructs, if any, protect regions only temporarily, and ensure that any regions that are removed inside them and are not protected when the execution enters them will be reclaimed before the execution exits them.

```
some [...]
  (c1): entry to the commit
    (a) push a commit frame
    (b) save the sequence number
    (c) save the pointer to the top disj frame
    (d) save the to-be-reclaimed old regions and their number
  ( the inner goal )

  (c2): commit point
    (a) reclaim the saved old regions
    (b) reclaim the new regions
    (c) restore the state of the disj stack
    (d) pop the commit frame

  (c3): failure point
    (a) restore status of the saved regions
    (b) pop the commit frame
```

Figure 9. RBMM runtime support for commit.

As before, our solution involves a new embedded stack, the *commit stack*. We push a new commit frame at (c1), and fill in its fixed fields, which will be discussed later. Following this will be the code that, for each region in the removed set of the commit goal, checks whether the region is already protected. If it is, that region is left alone. If it is not, we add the handle of the region to the commit frame's nonfixed part, and record the address where this handle is stored in the commit frame in the region's own header, in a new field called `commit_slot`. This way, when a region that should be reclaimed inside the commit actually survives to (c2) due to the protection of an inner disjunction, code at (c2) can iterate through all the region handles in the commit frame and reclaim those regions. However, we cannot do this for regions that are actually reclaimed inside the commit (whose remove instructions were executed in the last disjuncts). That is why, when we reclaim a region, we check whether its header's `commit_slot` field is null. If not, then it will contain the address of a pointer to the region header from a commit frame and the reclaim operation will replace that pointer in the commit frame with a null. Making the loop at (c2.a) ignore such nulled-out region handle pointers ensures that each region recorded in the commit frame's list is reclaimed exactly once, and that this will happen as soon as possible.

If the goal inside the commit fails, we need to undo the update of the saved regions' `commit_slot` fields, so at (c3.a) we reset them all to their original values. To make this possible, we save each original value in the commit frame next to the pointer to the region header from which it is taken. This effectively chains together all the entries referring to a given region in the commit stack. The reclaim operation will set to null not just the first slot in this chain, but all of them.

This mechanism is sufficient to correctly handle any old regions that are in the commit goal's removed set. To handle any new regions (regions created inside the commit) that are also removed inside the commit, we record the current region sequence number in the commit frame at (c1). When a new region is removed in the commit, if it is not protected, it is reclaimed. If it is protected, we mark it so that at the commit point we can reclaim it. We add a field `destroy_at_commit` to the region header, and we augment the `remove` instruction again so that when a protected, new region is removed in a commit, the `remove` instruction will set the region's `destroy_at_commit` field to true. At the (c2.b) part of the commit point, we traverse the region list until meeting an old region, and reclaim the new regions whose `destroy_at_commit` field is true.

We do not need to worry about instant reclaiming of new regions in the created set and of new allocations into regions in the allocated set of the commit, since that will be done by the construct(s) surrounding the commit.

At the commit point, the Mercury execution algorithm throws away all the remaining alternatives of the goal inside the commit. To reflect this, at (c2) we need to restore the embedded disj stack to the state it had at (c1). This is why at (c1), we save the current disj stack pointer in a fixed slot in the new commit frame, and at (c2), we restore the disj stack pointer from there. The regions protected by the disj frames thrown away by this action will be exactly the ones removed by the code at (c2.b).

The layout of commit frames is shown in Figure 10, with the fixed and nonfixed parts are separated by a thick line. The meaning

prev_commit_frame	*(previous commit frame)*
saved_seq_num	*(saved sequence number)*
saved_disj_sp	*(disj stack pointer)*
num_saved_region	*(number of saved regions)*
region_id	*(handle to a saved region)*
prev_commit_slot	*(original commit slot of the saved region)*
...	

Figure 10. The structure of a commit frame.

of the first two fields should be clear. The third field contains the value of the disj stack pointer at the time when the commit was entered. The last field gives the number of region handles and saved `commit_slot` fields actually stored by the code at (c1.d) in the nonfixed part.

5. Experimental evaluation

We have implemented the algorithms presented in the previous sections in the Melbourne Mercury compiler, specifically in the backend which generates low-level C code. We use region pages of size 2048 words, in which 2047 are available to store program data. When needed, we request blocks of 100 region pages from the OS. However, we get very similar results with other values of both parameters.

Our experiments used a set of small benchmark programs. With the exception of **boyer**, all these programs, were previously described and used in [10], though we changed the input data of **primes** and **queens** to increase their runtime and memory demands. While we would have liked to test our system with bigger, more realistic programs, we were prevented from doing so by the region transformation's current lack of support for higher order code and multi-module programs. The experimental machine was a PC with two 2.8 GHz Pentium 4 CPUs, 512 MB of RAM, running Debian GNU/Linux 3.1 SMP.

We measured the memory consumption of the RBMM system. The region behaviour of the benchmark programs was reported in [10] using a region simulator. Here we collected the data from the working RBMM system. For each benchmark, we give the total number of regions created during its execution, and the maximum number of regions coexisting during its run. We also include the total number of words allocated and the maximum number of words that coexist. *SLR* is the Size of the Largest Region and *S (%)* is the saving, calculated by 1 - Max words/Total words. The results in Table 1 are consistent with the results of [10]. RBMM achieves optimum memory management in **nrev** (which reverses a list of 5000 integers), in **primes** (which finds all primes less than 20000), and in **qsort** (which sorts a list of 100000 integers).

	Regions		**Words used**		**SLR**	**S (%)**
	Total	**Max**	**Total**	**Max**		
boyer	12	3	430,683	143,561	143,505	66.67
crypt	416	3	3,442	94	64	97.27
dna	2,082,005	8	18,926,797	4,590,797	4,096,000	75.74
life	50,303	102	894,336	8,208	6,486	99.08
nrev	5,002	2	25,015,000	10,000	10,000	99.96
primes	2,264	1	5,221,386	39,998	39,998	99.23
qsort	200,002	21	5,865,744	200,000	200,000	96.59
queens	4,545,702	2	121,453,230	114	90	99.99

Table 1. Memory use result.

We also compared the runtime performance of our benchmark programs compiled in two ways, using the Boehm garbage collector and using RBMM. (The Boehm collector uses 1024 word pages and heuristically expands the heap on demand.) To eliminate the uncertainty involved in measuring small times, we ran each program many times in a loop. Each benchmark has a row in Table 2 that gives the number of iterations, the actual execution times with Boehm gc and with RBMM, the Boehm system's gc time (all in seconds, all for user mode only), the number of collections executed by the Boehm collector, and the savings achieved by using RBMM instead of the Boehm collector (given by 1 - RBMM runtime / Boehm runtime).

	# Iter	**Boehm gc**			**RBMM**	**Saving**
		runtime	**gc time**	**# gc's**	**runtime**	
boyer	2,000	15.00	2.00	89	14.55	3.0%
crypt	100,000	17.43	2.14	98	14.74	15.4%
dna	40	16.67	5.06	219	14.72	12.2%
life	200	11.58	1.14	51	12.13	-4.7%
nrev	40	20.08	6.33	284	8.25	58.9%
primes	100	19.26	3.46	149	12.64	34.4%
qsort	80	13.44	3.79	143	7.27	45.9%
queens	5	18.47	3.81	172	12.24	33.7%

Table 2. Time result.

The RBMM system gets clearly better runtime for six out of eight benchmark programs, a little better in **boyer**, and slightly worse in **life**. On average, using RBMM is about 25% faster.

In four programs, **dna**, **nrev**, **primes**, and **qsort**, the compiler does not need to generate any extra code either to protect backward live regions or to do instant reclaiming. The programs are deterministic and the conditions of the if-then-elses in them are simple tests. RBMM does reclaim the temporary data in an optimal way for **nrev**, **primes**, and **qsort**. For **dna** some temporary data is not put in a separate region by the region analyser, which is the reason for the existence of a large region. The overhead of allocating in this region explains the relatively lower saving of 12%.

So for these deterministic programs, we get the benefits of RBMM with only a little overhead being required to support nondeterminism, such as for maintaining the region list and checking in the `remove` instruction to see if a region is protected. In these four programs, we gain an average speedup of 38% using RBMM compared to using the Boehm collector.

crypt and **queens** are the only two programs among our benchmarks that contain nondet disjunctions. We can consider them to be the exceptions because in practice, almost all Mercury predicates (98+%) have at most one solution. Most of the extra code is to support these disjunctions. (They contain only one commit operation each.) In **crypt**, the support for disjunction protects one region and takes care of instant reclaiming of new regions (reclaiming 95% of the words used in total). In **queens**, we again protect one region and do instant reclaiming of new regions (now 90% of the words used in total). Another 10% of words is reclaimed by the support for instant reclaiming of new allocations in existing regions, though this requires saving and restoring 12,356,378 region size records.

For these two programs, RBMM being faster shows that the overhead for the runtime support they need is clearly smaller than the time needed by the Boehm collector.

For the deterministic programs **boyer** and **life**, the only extra code is for supporting if-then-elses. As procedure calls appear in the conditions of if-then-elses, region size records of the input regions are stored in the ite frames (116,952 for **boyer** and 355,576 for **life**). This turns out to be pure overhead, as the programs only allocate in new regions and no instant reclaiming is needed. This is because we approximate the regions which may be allocated into. This probably explains the slight gain in **boyer** and the small loss in **life**. Additional experiments in which no region size records were saved at all, show that **boyer** and **life** run about 18% faster without any loss opportunities for reclaiming memory. We intend to investigate general algorithms for finding out which region size records are worth saving.

Compilation time is higher when using RBMM, with the increase ranging from 3% to 66%, and averaging 17%, almost all due to the program analysis (which has not yet been tuned). Comparing the object files generated by the two systems, the object files using RBMM are larger, with the increases ranging from 28% to 104%, and averaging 57%. Allocations are done by function call with both Boehm gc and RBMM, but as we showed in Section 4, RBMM also needs extra supporting code. For now, these use macros, since conditionally incrementing a virtual machine register is easier in macros (you cannot pass the address of a real register implementing a virtual machine register to a function). If code size ever becomes a problem we could move most of the work of these macros into functions. The memory used by embedded frames is negligible in all our benchmarks.

6. Conclusion

The work in [10] augmented Mercury programs with region information. We have taken that work further by extending the runtime system of Mercury to actually realize region-based memory management for Mercury programs [1]. Our extensions can support backtracking correctly without incurring significant overhead in deterministic programs. Our experiments show that using RBMM instead of the Boehm collector yields speedups for most of the benchmark programs. For some benchmarks, RBMM achieves optimum memory consumption.

Our work is not the first to provide runtime support for region-based memory management for logic programming languages. In [8], the authors presented extensions to the WAM to implement region-based memory management for Prolog. The main differences between their work and ours are that Mercury supports if-then-elses with conditions that can succeed more than once, and the Mercury implementation generates specialized code for many situations that Prolog handles with a more general mechanism (e.g., Mercury has separate implementations for nondet and semidet disjunctions). The first difference required new algorithms, while the second posed an engineering challenge in keeping overheads down, since any given overhead would hurt Mercury more than Prolog due to Mercury's higher speed.

We have met that challenge. Whereas [8] reports that introducing RBMM into their Prolog system gave a speedup for only three out of six benchmark programs, and the overall average speedup was only 8%, our results show speedups for seven out of eight benchmark programs, with an overall average speedup of 25%. We can therefore say that our implementation often incurs very modest runtime overhead.

The main limitation of our work is that currently, the program analysis underlying our system supports only a subset of Mercury. We intend to work on extending the analysis to handle the rest of the language, including higher-order code, foreign language code and multi-module programs. These extensions to the analysis will not require any changes to the runtime system we have presented. We also want to extend the program analysis to deal with backward liveness and to study how we can exploit that information to simplify the runtime system. We then can compare the effectiveness and complexity of this approach to protecting backward live regions.

7. Acknowledgements

We would like to thank the rest of the Mercury team and the anonymous referees. Quan Phan is supported by GOA /2003/08 and FWO Vlaanderen.

[1] Current, RBMM-enabled versions of Mercury can be downloaded from http://www.cs.mu.oz.au/research/mercury/download/rotd.html.

References

[1] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185. ACM Press, 1995.

[2] L. Birkedal, M. Tofte, and M. Vejlstrup. From Region Inference to von Neumann Machines via Region Representation Inference. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, 1996.

[3] H. Boehm, and M. Weiser. Garbage collection in an uncooperative environment. *Software - Practice and Experience*, 18:807–820, 1988.

[4] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *Proceedings of the 4th International Symposium on Memory Management*, pages 85–96. ACM Press., Oct. 2004.

[5] F. Henderson. Accurate garbage collection in an uncooperative environment. In *Proceedings of the 3rd International Symposium on Memory Management*, pages 150–156. ACM Press., 2002.

[6] F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Principles and Practice of Declarative Programming.*, pages 175–186. ACM Press., 2001.

[7] H. Makholm. A region-based memory manager for Prolog. In *Proceedings of the 2nd International Symposium on Memory Management*, pages 25–34. ACM Press., 2000.

[8] H. Makholm and K. Sagonas. On enabling the WAM with region support. In *Proceedings of the 18th International Conference on Logic Programming*. Springer Verlag., 2002.

[9] *Mercury language reference*. http://www.cs.mu.oz.au/research/mercury/information/doc-latest/mercury_ref.

[10] Q. Phan and G. Janssens. Static region analysis for Mercury. In *Proceedings of the 23rd International Conference on Logic Programming*, pages 317–332. Springer, 2007.

[11] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29(1-3):17–64, October-December 1996.

[12] M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17:245–265, 2004.

[13] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation.*, 132(2):109–176, Feb. 1997.