

A proposal for runtime region support for Mercury programs

Quan Phan
Gerda Janssens

Report CW482, March 2007



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

A proposal for runtime region support for Mercury programs

Quan Phan
Gerda Janssens

Report CW482, March 2007

Department of Computer Science, K.U.Leuven

Abstract

Region-based memory management is an interesting compile-time memory management technique. Implementing region-based memory management for a language is twofold, firstly developing a static region analysis that annotates programs with region instructions and secondly providing a region-aware runtime system that can run the region-annotated programs. In this report we develop the runtime support needed by the region-annotated programs which are the result of a region analysis that has been developed for Mercury. We implement the runtime support in a region simulator which allows us to evaluate the memory behaviour of a region-annotated program as if it is executed in a fully-implemented region-based memory management system. Using the region analyser and the region simulator we experiment with several benchmark programs. The obtained memory consumption results are very encouraging, in some programs optimal memory management is achieved.

Keywords : Program Analysis, Mercury, Region-based memory management, Region simulator, Runtime region support.

CR Subject Classification : D.3.4, I.2.3

A Proposal for Runtime Region Support for Mercury Programs

Quan Phan, Gerda Janssens

Department of Computer Science, K.U.Leuven
Celestijnenlaan, 200A, B-3001 Heverlee, Belgium,
{quan.phan,gerda.janssens}@cs.kuleuven.be

Abstract. Region-based memory management is an interesting compile-time memory management technique. Implementing region-based memory management for a language is twofold, firstly developing a static region analysis that annotates programs with region instructions and secondly providing a region-aware runtime system that can run the region-annotated programs. In this report we develop the runtime support needed by the region-annotated programs which are the result of a region analysis that has been developed for Mercury. We implement the runtime support in a region simulator which allows us to evaluate the memory behaviour of a region-annotated program as if it is executed in a fully-implemented region-based memory management system. Using the region analyser and the region simulator we experiment with several benchmark programs. The obtained memory consumption results are very encouraging, in some programs optimal memory management is achieved.

1 Introduction

The basic idea of region-based memory management (RBMM) is to store program terms in different regions of memory cells and when all terms in a region are dead, their memory can be deallocated by releasing the region as a whole. Region inferences, which derive the regions used by programs and their lifetime, and programming systems with RBMM support are popular and successful in functional programming world [10, 4, 1]. RBMM for imperative programming languages has also attracted many researchers in recent years [3, 2]. On the contrary, research on RBMM in logic programming has been limited. In [6, 5] the author developed an RBMM analysis for a non-standard implementation of Prolog. This ad hoc development implies that substantial changes will be needed to integrate the analysis into standard Prolog systems. In [7], Makhholm H. and Sagonas K. took a step further to implement RBMM for a standard WAM-based Prolog system. This work mainly focused on the extensions of the WAM instruction sets and the runtime data structures that are needed to run Prolog programs in RBMM context. It reused a type-based region inference which was developed for the functional programming language SML [4] to work with untyped Prolog, which could lead to poor memory behaviour for large programs in which type inference

for Prolog obtains imprecise type information. A dedicated static region analysis for the logic programming language Mercury has been developed in [8]. For a Mercury program, the analysis derives the regions used by the program and their lifetime, namely when they need to be created and when they can safely be removed. However, the region lifetime computed by the analysis is only correct with respect to the forward execution of programs. The idea to provide runtime support to make such a region analysis work correctly and efficiently with backtracking has been introduced and implemented in [6, 5, 7] in the context of Prolog.

This paper reports the runtime support for our static region analysis [8] (from now on referred to as the region analysis) in the context of Mercury (the implementation described in [9]). The support not only provides the safety for programs to run but also offers the ability of instant reclaiming in the context of RBMM. Moreover, we construct a region simulator which can mimic the behaviour of Mercury programs in an RBMM system. The simulator allows us to evaluate the memory performance of RBMM without having to fully implement it. Using the simulator we are able to experiment with several benchmark programs. The experimental results about memory consumption are very encouraging, for some programs we obtain optimal memory management.

In Section 2, the runtime support needed for RBMM is introduced. The detailed data structures and algorithms to support non-deterministic code and if-then-else are developed in Sections 3 and 4, respectively. The cooperation between the supports for non-deterministic code and if-then-else then is shown in Section 5. Section 6 describes our region simulator. The results of experiments with the simulator are presented and discussed in Section 7. Finally, Section 8 concludes.

2 Enhancing Mercury Runtime System

2.1 Regions and Region Management

In RBMM, the main memory is organised into regions. The basic memory management actions are creation/allocation of regions, allocation of cells in a region, and deallocation/removal of regions. We will describe how the regions themselves and the region management operations can be implemented.

For those two tasks we can follow the “standard” implementation in existing work about RBMM for SML and Prolog [6, 5, 7]. A brief summary is as follows. A region is implemented as a linked-list of fixed-size pages. Theoretically, a region can grow infinitely large in size. Practically, when a *region is allocated* it contains one page and it can be extended by adding more pages when necessary. A program maintains a global list of such pages. When a region needs a page it is taken from this global list. If this list runs out of pages, the program requests a big chunk of memory from the operating system, divides it into several pages, and add them to the list. When a *region is removed* the list of pages of the region is returned to the global list. Each region has a *management record* in the first

page, which contains a pointer to the newest page and the amount of free space in the newest page. An *allocation into a region* always happens in the newest page. When the free amount in this page is not enough for an allocation, the region is extended.

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 pages, and that if the remaining space of a page is not enough for an allocation, a new page needs to be added and the space is wasted. In some existing implementations of RBMM the page size is fixed to 16 or 32 words (64 or 128 bytes).

2.2 Runtime Support for Backtracking in RBMM

Handling backtracking in Mercury. The details of steering backtracking in Mercury can be found in [9]. Briefly, Mercury uses chronological backtracking. It uses a separate stack, called *nondet stack*, to deal with non-deterministic code. Before executing a piece of non-deterministic code a nondet frame is pushed on this stack. When the execution backtracks, it reads the top nondet frame, which is pointed to by the *maxfr* register, to decide what needs to be done. When the top nondet frame says that it has no more alternative, the frame is popped. (This is also known as pop on fail stack, contrary to the normal deterministic stack which is popped on succeed.) This mechanism is used to deal with disjunctions and *nondet* and *multi* procedures.

Special attention should be paid to the implementation of the *if-then-else*, which is a standard construct in Mercury. Its operational semantics is that the *condition* goal will be executed first, if it succeeds the *then* goal will be executed. Otherwise the *else* goal is run as if the condition goal had not been tried. While this process can be considered another kind of backtracking it is not handled by the nondet stack. The motivation for this unique treatment and the implementation details can be found in [9]. Another noticeable point with if-then-else in Mercury is that the execution will not commit to the first solution of the condition goal. It means that if the condition goal succeeds a number of times, the then goal will also be executed that number of times.

The support needed. If backtracking happens in a region-annotated program, which is generated by the region analysis, the following situation can occur: a region is removed in an execution branch while it is still needed in another branch when backtracking happens. This would cause illegal accesses to values, which have been released, in a later execution branch. To ensure the correctness of the execution, when the program backtracks the runtime support is required to guarantee that the removed region will be brought back to live as if it has never been removed.

There are two other situations which may cause memory leaks and are closely related to *instant reclaiming*.

1. A region is created in a branch but the execution fails before its supposed removal in this branch is reached. This causes a memory leak because such regions are not released.

2. Some allocations are made into a region in a branch and then the program switches to another branch, creating a space leak in the region.

Note that these situations do not make an execution unsafe as in the case with removals (i.e., access values through dangling pointers). Instant reclaiming is known to be an efficient method to reclaim memory in logic programs. Therefore it would be desirable that we can have a similar feature in the context of region-based memory management.

In summary, the runtime support will serve three tasks: undo the removal of a region, undo the creation of a region, and undo the allocations in a region when necessary. The overall effect of those actions will be to have the same memory state before the execution of each possible path at a choice point.

Next, we will develop the details of the runtime support for two kinds of backtracking in Mercury: non-deterministic code and if-then-else.

3 Runtime Support for Nondet and Multi Procedures

3.1 Undo Region Creation

Before executing a piece of nondet code, a nondet frame is created. The creations of the regions that exist *before* the creation of the frame do not need to be undone. It is because those regions are already there when the frame is pushed and should still be there when the execution backtracks to the frame.

We only need to undo the creations of regions which are created *after* the creation of the frame. To keep track of the creations that need to be undone we do as follows: when a region is created if the nondet stack is not empty, the creation will be recorded into a list, called *TerminationList*, at the top nondet frame. When the execution backtracks, the regions in the `maxfr.TerminationList`¹ will be removed.

3.2 Undo Allocation

The allocations into a region in the `maxfr.TerminationList` do not need to be undone because when a program backtracks, the whole region will be removed anyway.

We want to reclaim the allocations into regions that are not in the `maxfr.TerminationList` (i.e., the regions existed before the creation of the top nondet frame). Therefore, before allocation into such a region, we will save the region's current size into another list at the top nondet frame, called *SnapshotList*. Because we will only save the size of the region at the first allocation into it (i.e., the size right before the top nondet frame is created), at each allocation we check whether the region has an entry in the `maxfr.SnapshotList` and only store its snapshot if it is not yet there.

When the program backtracks, it goes through the `maxfr.SnapshotList` and restores the recorded regions to their previous sizes.

¹ We abuse the use of notations: `maxfr` here means the top nondet frame and `maxfr.TerminationList` means the *TerminationList* in the frame.

3.3 Undo Region Removal

As before, we will first look for removals that do not need to be undone. When no nondet frames are created between the creation and the removal of the same region, the removal does not need to be undone when the program backtracks. Consider the following pieces of non-deterministic code.

Example 1. Removals that do not need to be undone.

```
p(...):- ..., create R, no nondet code in between, remove R.
p(...):- ...
or
..., p(...), create R, no nondet code in between, remove R.
p(...):- ...
p(...):- ...
```

In both cases, R is in the TerminationList of p 's nondet frame and *remove R* is executed while the nondet frame of p is on top of the nondet stack. \square

R does not exist before the nondet frame of p . Therefore when the execution backtracks to the nondet frame of p , R is not needed. So in this situation *remove R* can remove the region straight away. We also need to delete the entry of R from the TerminationList so that the region will not be removed another time when the execution backtracks.

A removal of a region will need to be undone when some non-deterministic code appears between the creation and the removal of the region. It means that the region is not in the maxfr.TerminationList when the removal happens. Take a look at some pieces of non-deterministic code below.

```
Example 2. p(...):- create R, q(...), remove R.
p(...):- ...
q(...):- ...
q(...):- ...
or
..., p(...), create R, q(...), remove R.
p(...):- ...
p(...):- ...
q(...):- ...
q(...):- ...
```

Assuming that q uses R . If the first clause of q succeeds, *remove R* instruction will be executed. R is needed in the second clause of q therefore the removal needs to be undone. We can see that R is in the TerminationList of p 's nondet frame and when *remove R* is executed the nondet frame of q (not p 's) is at the top of the nondet stack. \square

It is difficult to actually remove a region and later on re-create the region with exactly the same content. One possible solution would be to ignore the operation of the *remove R* instruction in this situation. Because R is in the **Termination-List** of p 's nondet frame it will be removed later by *undo region creation* when the nondet frame of p is discarded.

In general, it would mean that we will ignore a removal of a region if the region is not in the `maxfr.TerminationList` when the removal happens. A finer solution will be to try to *shrink* a region when we cannot remove it. This means that we try to do *undo allocation* for the region to the size recorded in the `maxfr.SnapshotList` at the time the removal happens. In Example 2, the nondet frame of q is on top of the nondet stack when *remove* R is called. If there is no entry for R in the `SnapshotList` of the nondet frame of q , i.e., no allocations into R have occurred, the *remove* R instruction does nothing. Otherwise, R is resized to that saved size.

4 Runtime Support for if-then-else

The purpose of the support for if-then-else is to ensure the same memory state before the execution of the *if-then* part and of the *else* part. That is any changes to memory made by the condition goal will need to be undone if the condition goal fails and the else goal is reached. We will need to collect the information about those changes, called *undo* information.

4.1 Storing Undo Information for if-then-else

When supporting non-deterministic code we use the nondet stack to store the undo information. Similarly, we will also need a place to store that information for if-then-else. The condition of an if-then-else can contain other if-then-else constructs therefore the place will operate like a stack. Let us call it *ite* stack and call *ite_sp* the pointer to its top frame. When entering the condition of an if-then-else, a frame is pushed on the ite stack and the undo information for the if-then-else will be saved in this frame.

When is an ite frame popped? The scope of an ite frame of an if-then-else is exactly the scope of the condition goal. Therefore, we can immediately discard an ite frame of an if-then-else after the condition scope is passed. The first case is when the execution reaches the else goal of an if-then-else, i.e., the condition goal (either it is semidet or nondet) has failed to produce a solution and the execution commits to the else branch. The second case is when the execution reaches the then goal of an if-then-else with a semidet condition goal. In this case, the execution has passed the scope of the condition goal in the if-then part.

We cannot always discard the ite frame of an if-then-else with a nondet condition goal when the execution reaches the then goal. If the condition still has alternative(s), the ite frame cannot be discarded because the execution can backtrack to the condition goal. We will discard the ite frame when the then goal is reached and the condition goal has no more alternative. What Mercury currently does at the beginning of the then goal of an if-then-else with a nondet condition goal is to reset the redoip of the top nondet frame to point to the next alternative after the if-then-else. Now instead of just resetting the redoip to the next alternative after the if-then-else, we reset it to a label that first discards the ite frame then calls that alternative.

4.2 Undo Region Creation and Allocation

Region creations and allocations made by a condition goal can be dealt with in a similar manner as in the case of supporting non-deterministic code except that the nondet stack is now replaced with the ite stack. It means that when a region is created we will record the creation in the `ite_sp.TerminationList`² if the ite stack is not empty and that the first allocation into a region which is not in the `ite_sp.TerminationList` will be saved in the `ite_sp.SnapshotList` of the top ite frame.

When the execution reaches the else goal of an if-then-else, it uses the information in the `TerminationList` and `SnapshotList` in the top ite frame to undo the creations and allocations which have been made by the condition goal of the if-then-else.

4.3 Undo Region Removal for if-then-else

The removal of a region that is in the `ite_sp.TerminationList` does not need to be undone when the condition goal fails. It is because the creation and removal of such a region are an internal matter of the condition goal, hence the else goal is unaware of the region. So when removing a region, if the region belongs to the `ite_sp.TerminationList`, we will actually remove it and also remove its entry from the `ite_sp.TerminationList`.

If the condition goal of an if-then-else removes a region and the region is needed in the else branch, the removal may need to be undone. Similar to the case of supporting non-deterministic code, we can ignore the removal in this situation. It is worth pointing out that ignoring a removal will not harm the correctness of a program, it may only cause a memory leak. It would be beneficial if we can proceed such a removal when we know for sure that the else branch will not be selected, i.e., when the condition succeeds and the execution reaches the then goal.

We can achieve that effect by recording a removal of a region in the condition goal into a list, called *RemovalList*, in the top ite frame, if the ite stack is not empty. At the then branch we can go through the list and remove the recorded regions. It is as if we postpone the removal until we know for certain that the condition has succeeded. If the condition fails (to produce the first answer), the `RemovalList` will be of no use.

5 Cooperative Supports for Non-Deterministic Code and if-then-else

In Mercury, if-then-else and non-deterministic code can interleave therefore the RBMM supports for them need to cooperate when both nondet stack and ite stack are not empty.

² We abuse the use of notations: `ite_sp` means the top ite frame and `ite_sp.TerminationList` means the `TerminationList` in the frame.

5.1 Cooperative Region Management Operations

The pseudo code for region creation *create* is shown in Fig. 1. When both stacks are not empty we will need to record the creation into the TerminationLists in both of the top frames of the two stacks.

```

create R;
if (!empty(nondet stack)) {
    if (R not in maxfr.TerminationList) {
        add R into maxfr.TerminationList;
    } else {
    }
} else {
}

if (!empty(ite stack)) {
    if (R not in ite_sp.TerminationList) {
        add R into ite_sp.TerminationList;
    } else
    }
} else {
}

```

Fig. 1. Pseudo code of *create*.

Fig. 2 shows the pseudo code of the region allocation instruction, *allocate*. The first allocation into a region that is not in the TerminationLists of the maxfr and the ite_sp will be saved to their SnapshotLists, respectively.

When removing a region *R*, if *R* is in the maxfr.TerminationList: it will be actually removed if it is also in ite_sp.TerminationList, otherwise the removal is postponed by adding *R* into ite_sp.RemovalList. If *R* is not in the maxfr.TerminationList, it will be shrunk if it is in ite_sp.TerminationList, otherwise the shrinkage is postponed by adding *R* into ite_sp.RemovalList. The pseudo code of *remove* instruction is shown in Fig. 3. When a region is shrunk in a *remove* instruction, we will try to resize it to its old size which may have been saved in the maxfr.SnapshotList.

5.2 Cooperative Use of Undo Information

Commit removals in if-then-else. The ite_sp.RemovalList is used when the then goal of the current if-then-else is reached. It is like we commit the removals which would have happened in the condition goal.

If a nondet frame exists, those regions in ite_sp.RemovalList that are also in maxfr.TerminationList can actually be removed and they will also be eliminated from the maxfr.TerminationList so that they will not be removed twice. These

```

if (!empty(nondet stack)) {
    if (R not in maxfr.SnapshotList && R not in maxfr.TerminationList) {
        save the current size of R into maxfr.SnapshotList;
    } else {
    }
} else {
}

if (!empty(ite stack)) {
    if (R not in ite_sp.SnapshotList && R not in ite_sp.TerminationList) {
        save the current size of R into ite_sp.SnapshotList;
    } else {
    }
} else {
}
allocate into R;

```

Fig. 2. Pseudo code of *allocate*.

regions were added to `ite_sp.RemovalList` at (*) in Fig. 3. Those regions that are not in `maxfr.TerminationList` can only be shrunk because such regions will probably be needed at backtracking. They are the regions which were added to `ite_sp.RemovalList` at (**) in Fig. 3.

When the `nondet stack` is empty, the regions in `ite_sp.RemovalList`, which were added to `ite_sp.RemovalList` at (***) in Fig. 3, can be removed.

If any region in `ite_sp.RemovalList` also belongs to the `TerminationList` of another `ite` frame, we also need to remove its entry from the list to prevent the region from probably being removed once more. This situation can happen when this whole if-then-else (i.e., the if-then-else corresponding to the `ite` frame we are talking about) is in the condition of another if-then-else and the region is created in that condition. Note that a region will never be in both `RemovalList` and `TerminationList` of the same `ite` frame.

Undo creation and allocation at the else branch. The `ite_sp.TerminationList` and `ite_sp.SnapshotList` are used when the condition fails to produce the first answer and the then branch is not executed at all. That the condition fails implies that all the `nondet` frames which are created after the `ite_sp`, if any, have been discarded. At that time, if the `nondet stack` is not empty, the `maxfr` has had to be created before the `ite_sp`. Therefore, the regions in the `ite_sp.TerminationList`, which were created after the `ite_sp`, were also created after the `maxfr`, hence must also belong to the `maxfr.TerminationList`.

The regions in the `ite_sp.TerminationList` should not exist before executing the else branch so we will actually remove them. They also need to be removed from `maxfr.TerminationList` so that they will not be removed once more.

The regions in the `ite_sp.SnapshotList` will be resized to the sizes saved in this list.

```

if (!empty(nondet stack)) {
  if (R in maxfr.TerminationList) {
    // From nondet code's point of view, R can actually be removed
    if (!empty(ite stack)) {
      if (R in ite_sp.TerminationList) {
        remove R;
        remove R from ite_sp.TerminationList;
        remove R from maxfr.TerminationList;
      } else {
        // postpone removal(*)
        add R into ite_sp.RemovallList;
      }
    } else {
      remove R;
      remove R from maxfr.TerminationList;
    }
  } else {
    /* R not in maxfr.TerminationList, it's not OK to actually
    ** remove it, the best we can do is to shrink R
    */
    if (!empty(ite stack) {
      if (R in ite_sp.TerminationList) {
        shrink R;
      } else {
        // postpone shrinking (**)
        add R into ite_sp.RemovallList
      }
    } else {
      shrink R;
    }
  } else { // nondet stack is empty
    if (!empty(ite stack) {
      if (R in ite_sp.TerminationList) {
        remove R;
        remove R from ite_sp.TerminationList;
      } else {
        // postpone removal (***)
        add R into ite_sp.RemovallList;
      }
    } else {
      remove R;
    }
  }
}

```

Fig. 3. Pseudo code of *remove* instruction.

Undo creation and allocation at nondet backtracking. The information in the TerminationList and SnapshotList of maxfr are used before executing an alternative or before executing do_fail macro.

The regions in TerminationList are those that should not exist before executing an alternative, therefore we should actually remove them. They also need to be removed from the TerminationList of another ite frame if they happen to be there so that they will not be removed twice. Note that, when maxfr.TerminationList is used, a region in the list can never be in the RemovalList of another ite frame. The reason is that for a region in maxfr.TerminationList to be in the RemovalList of the ite frame of an if-then-else two following criteria must be met: a removal must happen to the region in the condition of the if-then-else and the region is not in the TerminationList of the ite frame of the if-then-else, i.e., the region is added to the RemovalList at (*) in Fig 3. The second criterion means that the creation of the region happens before the creation of the ite frame. Therefore, the sequence of actions leading to this situation has to be as follows.

creating of maxfr, create R, creating of ite frame, remove R, ...

The maxfr.TerminationList is used when the program backtracks to maxfr, at that point the if-then-else must have been tried, hence its corresponding ite frame must have been discarded.

The regions in maxfr.SnapshotList will be resized to the sizes saved in this list.

6 Region Simulator

Mercury runtime system is complex. It would require significant effort to integrate RBMM into the system. Therefore, it is beneficial if we can evaluate the memory efficiency of RBMM in Mercury before fully implement it. Our region simulator, which does not require substantial changes to the runtime system of Mercury to integrate with it, can imitate the memory consumption behaviour of Mercury programs as if they are executed in the context of RBMM.

Basically, the region simulator implements the cooperative RBMM runtime support as we described in Section 5. It maintains two stacks on its own: a nondet stack, which operates similar to the nondet stack in the Mercury runtime system and an ite stack. The simulator is implemented in C and Fig. 4 shows the prototypes of the operations provided by the simulator for managing regions, manipulating its stacks, and using the information on the stacks. The operations to manipulate the simulator's nondet stack are called from suitable places in the module handling the nondet stack in the Mercury runtime system in order to ensure that the simulator's nondet stack will behave exactly the same as the Mercury's nondet stack in terms of pushing and popping frames. The other methods will be called from Mercury programs, hence in Fig. 5 we define an interface to the C interface in Mercury code whose responsibility is mainly call-forwarding.

The analyser as presented in [8] will now generate region-annotated programs in valid Mercury code in which:

```

/* region management operations */
int create_region(void);
void remove_region(int regid);
void reg_alloc(int regid, int size);

/* operations for ite stack */
void create_ite_frame(void);
void discard_ite_frame(void);
void do_ite_removal(void);
void undo_ite_creations_allocations(void);

/* operations for nondet stack */
void create_nondet_frame(void);
void discard_nondet_frame(void);
void undo_nondet_creations_allocations(void);

```

Fig. 4. Region simulator interface.

```

% region management operations
:- pred create(int::out) is det.
:- pred remove(int::in) is det.
:- pred ralloc(int::in, int::in) is det.

% operations for ite stack
:- pred make_ite_frame is det.
:- pred commit_removal is det.
:- pred undo_ite_changes is det.

```

Fig. 5. Mercury interface to part of the region simulator interface.

- *create R* instructions are replaced by calls to *create(R)*, *remove R* instructions by calls to *remove(R)*.
- After each construction we add a call to *ralloc* to collect the number of words allocated.
- Procedure definitions and calls are extended with region parameters as extra parameters.
- *make_ite_frame* is added after *if*, *commit_removal* after *then*, *undo_ite_changes* after *else*.

An example of such simulate-able programs is shown in Fig 6³. Such simulate-able programs can be executed by the extended Mercury runtime system (i.e., the runtime enhanced with calls to the operations for handling the simulator’s nondet stack). The simulator will treat values of primitive types as if they are not stored in the heap. In particular, when dealing with a list of integers, the integers themselves are not put in a separate region but stored in the first words of the cons cells needed for the list skeleton.

³ The code has been modified from what was generated by the analyser for easy reading while changing no important aspects.

```

nrev(L, R,R1,R6) :-
(
  L = [],
  remove(R1),
  create(R6),
  R = []
;
  L = [H | T],
  nrev(T, V,R1,R5),
  create(R6),
  L1 = list.[H],
  ralloc(R6, 2),
  append(V, L1, R,R6,R5)
).

append(X, Y, Z,R3,R1) :-
(
  X = [],
  remove(R1),
  Z = Y
;
  X = [Xe | Xs],
  append(Xs, Y, Zs,R3,R1),
  Z = list.[Xe | Zs],
  ralloc(R3, 2)
).

makelist(N, L,R2) :-
( if
  make_ite_frame,
  N = 0
  then
  commit_removal,
  create(R2),
  L = list.[]
  else
  undo_ite_changes,
  N1 = N - 1,
  makelist(N1, L1,R2),
  L = [N | L1],
  ralloc(R2, 2)
).

main(!IO) :-
makelist(5000, L,R1),
nrev(L, R,R1,R2),
io.write_list(L, ",", io.write_int, !IO),
remove(R2).

```

Fig. 6. The simulate-able naive reverse program.

7 Experimental Results

When a Mercury program is executed it puts the terms that are created during execution on the heap. During forward execution the heap grows and only on backtracking instant reclaiming is done. With RBMM the terms will be put into regions and the regions can be freed (and reused) during a forward run of the program. In particular, it should be the case that temporary data is in regions which are freed as soon as the data is no longer needed.

In our experiments, we have included some deterministic programs that use some temporary data: **nrev** reverses a list of 5000 integers, **qsort** sorts a list of 100000 integers and **primes** finds all the primes less than 100; **dnamatch** and **life** are known to be difficult cases for region analysers. Two non-deterministic programs are used: **9-queens** program that first generates a permutation and then checks, and **crypt** that finds the unique answer to a cryptoarithmic puzzle⁴. The experiments allow us to measure the memory used by the benchmarks.

Tab. 1 shows the experimental results obtained by the region simulator for the annotated versions of the benchmarks. The experiments are done on a Pentium 4 2.8MHz with 512MB RAM running Debian GNU/Linux 3.1 machine, under a usual load.

⁴ The original and annotated source code of the benchmark programs can be found at <http://www.cs.kuleuven.be/~quan/benchmarks.tar.gz>

	nrev	qsort	prime	dnamatch	rdnamatch	life	rlife	crypt	queens
TR	5,002	200,002	29	2,082,005	2,083,005	50,303	50,403	418	7,689
MR	2	21	3	8	9	102	102	4	4
TW	25,015,000	5,865,744	916	18,537,685	18,541,685	894,336	894,336	3,442	159,234
MW	10,000	200,000	194	4,201,700	113,792	8,208	1,068	62	78
LR	10,000	200,000	194	4,096,000	64,000	6,486	390	32	60
S (%)	99.96	96.59	78.82	77.33	99.39	99.08	99.88	98.20	99.95

Table 1. Experimental results.

Our measurements reported in Tab. 1 are: the total number of regions (TR) created during the execution of a benchmark, the maximum number of regions (MR) that coexisted during its run, the total number of words (TW) that is allocated, the maximum number of words (MW) that coexisted, and the size (LR) in words of the largest region. Then savings (S) can be computed as $(TW - MW)/TW$.

The fact that TR is much larger than MR confirms that the data used by a program can indeed be divided into regions that can be freed timely during its execution. If Mercury runs the programs without memory management, it will actually need TW words. When using RBMM only MW are needed. The savings are impressive, about 92% on average. The large savings can partly be explained by the fact that the analysis did a good job in partitioning the heap into regions such that temporary data can be timely freed. Optimal memory management is achieved for **nrev**, **qsort** and **prime**, as the programs use no more memory than what is needed to store their input data. In a standard LP system, all memory management in **crypt** and **queens** will be handled well by *instant reclaiming* because backtracking happens very frequently. The savings with RBMM in these two benchmarks are with respect to the situation where instant reclaiming is not used therefore seem to be unfair. However, they show that the runtime support for backtracking can also provide the ability of instant reclaiming in the context of RBMM.

In **dnamatch** and **life**, while the savings seem acceptable, the memory management is actually suboptimal. In these programs, there is a region that exists for almost the whole runtime and contains all the temporary and final values of the computation of the programs' output, which make up a significant part of the maximum number of words used. This undesirable performance is due to the fact that the present algorithm fails to split temporary values from the final outputs in these programs. A well-known solution to this problem is to make such programs *region-friendly* by rewriting after studying their RBMM-related behaviour [10, 1, 4, 7]. **rdnamatch** is a region-friendly version of **dnamatch**, achieved by adding an extra predicate to copy the temporary values into regions different from the region of the final output. An orthogonal solution is to enhance the static analysis: **rlife**, a manually adapted version of the region-annotated **life**, illustrates the effect of such a possible improvement. Their data show a large reduction in the maximum number of words needed. The latter solution is likely more preferable because it is entirely automated, hence freeing logic programmers from caring about memory management tasks. Moreover, the

first solution implies extra time and memory costs of copying, while the second solution requires the same total number of words as before but the region analysis can distribute them more cleverly. We are working further in this direction.

Three of our benchmarks, **nrev**, **qsort**, and **dnamatch**, were also reported in [7] with the same inputs. We achieve optimal memory use in **nrev** and **qsort**, while in [7] the former was reported using maximally double and the latter 1.66 times the memory size of the input list. For **dnamatch**, we gain a saving of 77.33% compared to 33.5% shown in [7].

8 Conclusion

In this report we have developed the necessary runtime support for RBMM in the context of the typed logic programming language Mercury. The runtime support makes backtracking transparent to the static region analysis that was presented in [8]. The idea then has been implemented in a region simulator that can simulate the behaviour of Mercury programs as if they are running in an RBMM system. Mercury programs, which are annotated with RBMM instructions from the simulator's interface, can be run by the region simulator. Using the simulator we experimented with several benchmark programs. The memory use results of the benchmarks are positive, in some programs we obtain optimal memory consumption. This indicates that the runtime support could maintain the memory efficiency of RBMM in programs with backtracking. It should be noted that the experiments have been done only with small programs and that the lack of the completely implemented runtime supporting for RBMM makes us neither able to measure runtime performance and the internal cost of the region allocator nor to provide a thorough comparison between our approach and other memory management techniques available in Mercury, such as runtime garbage collection and compile-time garbage collection. We are working on integrating the presented algorithm into a standard version of the Mercury compiler and hope to lift this limitation soon.

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 1995 Conference on Programming Language Design and Implementation*, pages 174–185. ACM Press, 1995.
2. S. Chorem and R. Rugin. Region analysis and transformation for Java. In *Proceedings of the 4th International Symposium on Memory Management*, pages 85–96. ACM Press., October 2004.
3. D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.*, pages 282–293. ACM Press., 2002.
4. F. Henglein, H Makhholm, 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.

5. H. Makhholm. A region-based memory manager for Prolog. In *Proceedings of the 2nd International Symposium on Memory Management*, pages 25–34. ACM Press., 2000.
6. H. Makhholm. Region-based memory management in Prolog. Master’s thesis, University of Copenhagen, 2000.
7. H. Makhholm and K. Sagonas. On enabling the WAM with region support. In *Proceedings of the 18th International Conference on Logic Programming*. Springer Verlag., 2002.
8. Q. Phan and G. Janssens. Towards region-based memory management for Mercury programs. In *CICLOPS*, 2006.
9. 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.
10. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation.*, 132(2):109–176, February 1997.