Static region analysis for Mercury*

Quan Phan and 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 a form of compile-time memory management, well-known from the functional programming world. This paper describes a static region analysis for the logic programming language Mercury. We use region points-to graphs to model the partitioning of the memory used by a program into separate regions. The algorithm starts with a region points-to analysis that determines the different regions in the program. We then compute the liveness of the regions by using an extended live variable analysis. Finally, a program transformation adds region annotations to the program for region support. These annotations generate data for a region simulator that generates reports on the memory behaviour of region-annotated programs. Our approach obtains good memory consumption for several benchmark programs; for some of them it achieves optimal memory management.

1 Introduction

Memory management is a classic problem in the implementation of programming languages. Manual memory management using explicit constructs, such as *malloc/free* in C, is widely known to be error-prone. Therefore automatic memory management in the form of runtime garbage collection (RTGC) has become an integral part in many modern programming languages. The use of RTGC is even more important for declarative logic programming (LP) languages because these languages have no procedural constructs for memory management, and instant reclaiming based on backtracking is not feasible in many logic programs. While runtime garbage collectors for LP languages can reclaim more than 90% of the heap space of a logic program, they incur execution overhead because collectors often need to temporarily stop the main program.

Recently, there has been increasing interest in compile-time memory management to reduce the overhead and unpredictability of RTGC. This static method generally follows two approaches: compile-time garbage collection (CTGC) and region-based memory management (RBMM). CTGC looks for program points at which allocated memory cells are no longer used and instructs the program to reuse those cells for constructing new terms, reducing the memory footprint and in some cases achieving faster code. In LP, this idea has been used to reuse memory cells locally in the procedures of Mercury programs [9, 10, 8]. The basic idea of RBMM is to divide the heap memory used by a program into different regions.

* This work is supported by the project GOA/2003/08 and by FWO Vlaanderen.

The dynamically created terms and their subterms have to be distributed over the regions in such a way that, at a certain point in the execution of the program, all terms in a region are dead and the whole region can be removed. RBMM is a topic of intensive research for functional programming languages [16, 1, 4, 15] and more recently also for imperative languages [3, 2]. For LP languages, there has been only one attempt to apply RBMM to Prolog [7, 6, 5]. However, the algorithm for RBMM in [6, 5] was developed for a non-standard implementation of Prolog which would require substantial changes to have it implemented in any standard implementation. The authors of [7] fixed the problem by implementing RBMM in the context of a WAM-based Prolog system. Nevertheless, the work mainly concentrated on the runtime extensions needed by regions to run Prolog programs with RBMM. It reused a type-based region analysis 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 LP obtains imprecise type information. The limited research on RBMM in LP, therefore, suggests that it is worthwhile to investigate how a dedicated static region analysis can be developed and implemented in the context of typed logic programming languages.

Mercury is a pure LP language designed with a native, expressive type system, which makes it a natural context for this research. The main contribution of this paper is an algorithm that augments Mercury programs with region annotations. In addition, we have also implemented a region simulator to run the region-annotated Mercury programs and to evaluate the memory performance of several benchmark programs. We find that the algorithm can be practically implemented, and that, with RBMM, the memory savings are very encouraging.

In Section 2, we explain how memory management for Mercury can be based on the use of regions. The whole algorithm is composed of three phases, which are described in Sections 3, 4, and 5, respectively. The concept of the region points-to graph and the points-to analysis are given in Section 3. Section 4 defines the live region analysis which uses the region points-to graph of each procedure to precisely detect the lifetime of the regions. We do the transformation by adding RBMM annotations in Section 5. Section 6 briefly discusses how the presented algorithm can support Mercury programs with backtracking. The prototype implementation of the algorithm and the experimental results are shown in Section 7. Finally, Section 8 concludes.

2 Regions and Mercury

Mercury programs. We assume that the input of our program analysis is a Mercury program that has been transformed by the Melbourne Mercury Compiler (MMC) into *superhomogeneous* form, with the goals reordered and each unification specialised into a *construction* (<=), a *deconstruction* (=>), an *assignment* (:=), or a *test* (==) based on the modes [14]. The *qsort* program in this form then consists of the following procedures as shown in Fig. 1.

The use of regions: an example. We illustrate the usefulness of distributing terms over regions using the *qsort* example. Memory cells representing a list can

```
main(!IO) :-
(1) L<=[2,1,3],
(2) A<=[],
(3) qsort(L,A,S),
(4) io.write(S, !IO),

qsort(L,A,S) :-
(
(1) L=>[],
(2) S:=A
;
(3) L=>[Le|Ls],
(4) split(Le,Ls,L1,L2),
(5) qsort(L2,A,S2),
(6) A1<=[Le|S2],
(7) qsort(L1,A1,S)
).
```

```
split(X,L,L1,L2) :-
(
(1) L=>[],
(2) L1<=[],
(3) L2<=[]
;
(4) L=>[Le|Ls],
    (
    (5) X>=Le
    ->
    (6) split(X,Ls,L11,L2),
    (7) L1<=[Le|L11]
    ;
    (8) split(X,Ls,L1,L21),
    (9) L2<=[Le|L21]
    )
).
```

Fig. 1. *qsort* program in superhomogeneous form.

be divided into cells for the elements and those for the list skeleton. Observing the memory behaviour of the *qsort* procedure, we see that the output list has a new skeleton built up in the accumulator while its elements are those of the input list. In the *main* predicate, the input list L is no longer used after the call to *qsort*. This means that if the skeleton of the input list, the elements, and the skeleton of the output list (and the accumulator) are stored in three different regions we can safely free the memory occupied by the input list's skeleton by removing its region after the call. Take a closer look inside the *qsort* procedure at (4). The call to *split* creates two new lists with two new skeletons while the elements are also those of the input list. Therefore, if the two new skeletons are stored in regions different from the region of the input list's skeleton, the region can even be removed earlier, namely after this call to *split* inside *qsort*. So, by storing different components of the lists in separate regions we can do timely removal and recover dead memory sooner.

The *qsort* program with region support produced by our analysis is shown in Fig. 2 with the region annotations in bold. In analogy to program variables used to refer to memory cells, in RBMM, we use *region variables* to refer to (i.e., to be bound to) physical regions in memory. Two special instructions *create* and *remove* handle the creation and removal of regions. **create(R)** creates a region, and makes the region variable **R** bound to it. **R** must be unbound before and be bound after the operation. **remove(R)** removes the region to which **R** is currently bound. **R** must be bound before and be unbound after the operation. That region variables can become unbound makes them different from regular Mercury variables. In a procedure definition, $\{\ldots,\mathbf{R_i},\ldots\}$ is the list of *formal region parameters*. At a call site of the procedure the caller will provide the *actual region parameters* in the usual manner of parameter passing. The information about which variables are stored in which regions will be given in Fig. 3.

In Fig. 2, assume that in the *main* procedure, the list L's skeleton is in the region (bound to) **R1**, its elements in **R2**, and the skeleton of the accumulator in **R3**. In the *qsort* procedure, the region of the skeleton of the list L passed to *qsort* from *main* is removed in the base case branch of *split* in the call at (4). The two new skeletons of the lists *L1* and *L2* are allocated in two separate

```
main(!IO) :-
  create(R1), create(R2),
  (1) L<=[2,1,3],
  create(R3),
  (2) A<=[],
  (3) qsort(L,A,S){R1,R2,R3},
  (4) io.write(S, !IO),
  remove(R2), remove(R3).
qsort(L,A,S){R1,R2,R3} :-
(
(1) L=>[],
    remove(R1),
(2) S:=A
;
(3) L=>[Le|Ls],
(4) split(Le,Ls,L1,L2){R2,R1,R2,R4,R5},
(5) qsort(L2,A,S2){R5,R2,R3},
(6) A1<=[Le|S2],
(7) qsort(L1,A1,S){R4,R2,R3}
).
```

```
split(X,L,L1,L2){R5,R1,R2,R3,R4} :-
(
(1) L=>[],
    remove(R1),
    create(R3),
(2) L1<=[],
    create(R4),
(3) L2<=[]
;
(4) L=>[Le|Ls],
    (
    (5) X>=Le
    - >
    (6) split(X,Ls,L11,L2){R5,R1,R2,R3,R4},
    (7) L1<=[Le|L11]
    ;
    (8) split(X,Ls,L1,L21){R5,R1,R2,R3,R4},
    (9) L2<=[Le|L21]
    )
).
```

Fig. 2. Region-annotated *qsort* program.

regions. These regions are created by the base case branch of *split*, and removed (indirectly) by the calls at (5) and (7). If *L1* and *L2* are empty lists the removals will happen in the base case branch of *qsort*; otherwise, they will happen in the base case branch of *split*. The region of the output list's skeleton is the region of the accumulator, which is created in *main*.

3 Region Points-to Analysis

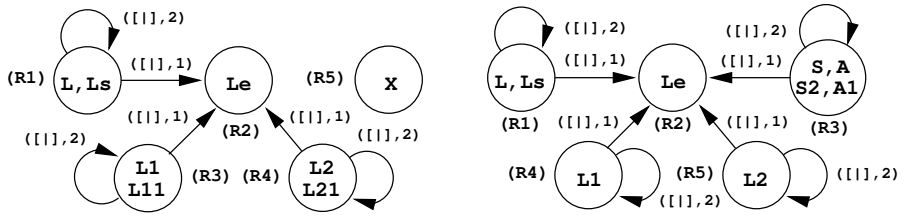
The goal of this analysis is to build, for each procedure, a region points-to graph that represents the partitioning into regions of the memory used by the procedure. The concept of a region points-to graph was introduced for Java in [2] and we adapted it to Mercury.

Region points-to graph. For a procedure p, a region points-to graph, $G = (N, E)$, consists of a set of nodes, N, representing regions and a set of directed edges, E, representing references between the regions. Each node has an associated set of variables of p which are stored in the region represented by the node. For a node n, its set of variables is denoted by $vars(n)$. The node n_X denotes the node such that $X \in vars(n_X)$. A directed edge $(m, (f, i), n)$ denotes an edge from m to n, which is labelled by the type selector (f, i) and represents the structured relation between the variables in the two nodes. The type selector (f, i) selects the i^{th} argument of the functor f [8].

The points-to graphs of *split* and *qsort* procedures are shown in Fig. 3. For *split*, we see that the skeletons of the lists L and Ls are in the same region and that the list elements are in the region pointed to by the edge labelled by $([], 1)$. The self-edge with the label $([], 2)$ is for the skeleton.

The region points-to graph $G = (N, E)$ of a procedure p is *compatible* if and only if G satifies the following invariants.

1. If a unification $X := Y$ is in p then X and Y are in the same node.



(a) *split* (b) *qsort*

Fig. 3. The points-to graphs of *split* and *qsort*.

2. If a unification $X = f(\ldots, Y_i, \ldots)$ (i.e., <= or =>) appears in p then $n_X, n_{Y_i} \in N$, and, for each i, there exists exactly one edge with the label (f, i) from n_X and $(n_X, (f, i), n_{Y_i}) \in E$.
3. If X is a variable bound to a term, Y is a variable bound to a subterm of that term and they are of the same type, then they are in the same node.
4. Every variable of p belongs to exactly one node and the variables in a node have the same type, which is regarded as the type of the node.
5. If p calls a procedure q, then the subgraph of the region points-to graph of p rooted at the nodes of the actual parameters must be a *conservative approximation* of the subgraph of the graph of q rooted at the nodes of the formal parameters of q.

The fifth invariant includes the effects of procedure calls in the region points-to graph of p. According to [2], a region points-to graph $G = (N, E)$ is a conservative approximation of $G' = (N', E')$ if there exists a function $\alpha : N' \rightarrow N$ such that $(n, (f, i), m) \in E'$ implies $(\alpha(n), (f, i), \alpha(m)) \in E \ \forall n, m \in N'$.

The reason for the third invariant is to ensure that the terms of recursive type are always stored in a fixed, finite number of regions. For a more thorough development of this design choice the reader is referred to Section 2.3 in [12].

The task of the region points-to analysis then is to produce a compatible graph for each procedure in a program. To update a region points-to graph $G = (N, E)$ the analysis uses two operations *unify* and *edge* defined as follows.

- $k = \mathit{unify}(n, m)$: unifies nodes n and m and returns the new node k.
  - $N \leftarrow N \setminus \{n, m\} \cup \{k\}$, k is a new node and $\mathit{vars}(k) = \mathit{vars}(n) \cup \mathit{vars}(m)$.
  - $E \leftarrow E$ where all appearances of m and n are replaced by k.
- $\mathit{edge}(n, \mathit{sel}, m)$: creates an edge with a label *sel* from node n to node m.
  - $G \leftarrow (N, E \cup \{(n, \mathit{sel}, m)\})$.

The region points-to analysis is flow-insensitive (i.e., the execution order of the literals in a procedure does not matter), and consists of an intraprocedural analysis and an interprocedural analysis. We will describe them in turn.

3.1 Intraprocedural Analysis

Assume that we are analysing a procedure p. Its region points-to graph $G = (N, E)$ is computed as follows.

1. Each variable in p is assigned to a separate node: for a variable X, n_X becomes a node in N and $vars(n_X) = \{X\}$.
2. The specialized unifications in p are processed one by one as follows:
   - An assignment $X := Y$: apply $unify(n_X, n_Y)$ to ensure the first invariant.
   - A test $X == Y$: do nothing.
   - A deconstruction $X => f(Y_1, \ldots, Y_n)$ or a construction $X <= f(Y_1, \ldots, Y_n)$: create the references from n_X to each of n_{Y_1}, ..., n_{Y_n} by adding the edges $edge(n_X, (f, 1), n_{Y_1})$, ..., $edge(n_X, (f, n), n_{Y_n})$.
3. The rules in Fig. 4 are fired whenever applicable. Rules P1 and P2 are to ensure the second invariant. Rule P3 enforces the third invariant.

$$\frac{k = unify(n, n') \quad (k, sel, m) \in E \quad (k, sel, m') \in E \quad m \neq m'}{unify(m, m')} \text{ (P1)} \qquad \frac{edge(n, sel, m) \quad (n, sel, m') \in E \quad m \neq m'}{unify(m, m')} \text{ (P2)} \qquad \frac{edge(n, sel, m) \quad (k_1, m) \in E^+ \quad (m, k_2) \in E^+ \quad k_1 \neq k_2 \quad type(k_1) = type(k_2)}{unify(k_1, k_2)} \text{ (P3)}$$

E^+ is the reflexive, transitive closure of E; $type(n)$ returns the type of node n.

Fig. 4. Intraprocedural analysis rules.

3.2 Interprocedural Analysis

The interprocedural analysis updates the region points-to graph G_p of a procedure p by integrating the relevant parts of the region points-to graphs of the called procedures into G_p at each call site. For a call $q(Y_1, \ldots, Y_n)$, the head of the defining procedure is $q(X_1, \ldots, X_n)$. The analysis is performed as follows.

1. Process each procedure call $q(Y_1, \ldots, Y_n)$ in p: integrate the graph of q, $G_q = (N_q, E_q)$, into the graph of p, $G_p = (N_p, E_p)$ by building the partial α mapping from N_q to N_p as follows:
   (a) Initialize the α mapping with $\alpha(n_{X_1}) = n_{Y_1}, \ldots, \alpha(n_{X_n}) = n_{Y_n}$. For those nodes that have been unified in G_q, the corresponding nodes in G_p should also be unified. This is achieved by applying rule P4 in Fig. 5 to ensure that α is a function.
   (b) In the graph G_q, start from each n_{X_i}, follow each edge once and apply rules P5 - P8 in Fig. 5 when applicable. Those rules complete the α mapping and copy the parts of G_q that are relevant to n_{X_i}'s into G_p. When two nodes of G_p are unified (rules P4 and P5) or an edge is added to G_p (rules P7 and P8) we need to apply rules P1 or P3 respectively to G_p in order to maintain the second and the third invariants. (Rule P2 is never applicable because its conditions cannot be satisfied.)
2. Repeat step 1 until there is no change in G_p.

When the algorithm terminates, i.e., when no rules can be applied and we reach a fixpoint, the resulting region points-to graphs for the program's procedures will all be compatible. This is because for each procedure, the analysis starts with a points-to graph that satisfies the fourth invariant, and no rules invalidate it. The first invariant is respected by the intraprocedural analysis. If

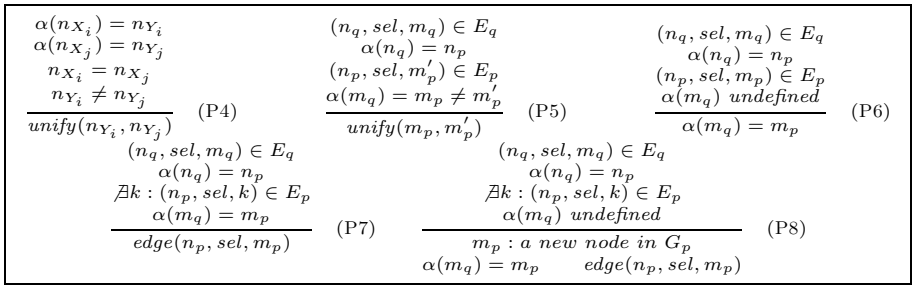


Fig. 5. Interprocedural analysis rules.

the second invariant does not hold then rules P1 and P2 must be applied and also if the third invariant is invalid then rule P3 is applied. Rule P4 ensures that at a call site an α mapping exists. Then if the fifth invariant does not hold at least one among the rules P5-P8 must be applicable. The termination of the algorithm can be shown by defining a partial order over the set of region points-to graphs and showing that the rules are actually monotonic and that there exists a maximum compatible points-to graph for each procedure. From now on, when we mention the region points-to graph of a procedure it means the compatible one obtained after both intra- and inter-procedural analyses.

For the *qsort* example, the region points-to graphs of *split* and *qsort* after the region points-to analysis are exactly as shown in Fig. 3.

4 Live Region Analysis

Each node in the region points-to graph of a procedure is named by a distinct region variable. A region variable being live means that it is bound to a physical region. During the execution of a program the memory cells holding the values of the variables must be allocated before they are used. Similarly, in RBMM, regions also need to be created before used, and should be removed when no long in use. The goal of live region analysis is to detect which region variables are live at each program point and to decide which regions are created and removed by each procedure.

We associate a *program point* with every literal in the body of a procedure p. An *execution path* in p is a sequence of program points, such that at runtime the literals associated with these program points are performed in sequence. We use the notions "before" and "after" a program point. Before a program point means the associated literal has not been executed yet, while after a program point means its literal has just been completed. The set of live region variables at a program point is computed via the set of live variables at the program point.

4.1 Live Region Variables at a Program Point

Live variables. A variable is live after a program point in a procedure p if:

- There exists an execution path in p containing the program point that instantiates the variable before or at the program point and uses it after the program point,

– OR it is an output variable of p, which is instantiated before or at the program point.

If we call $pre\_inst(i, P)$ the set of variables instantiated before the program point i in the execution path P, $post\_use(i, P)$ the set of variables used after i in P, $out(i)$ the set of variables instantiated by the goal at i, $out(p)$ the set of output variables of a procedure p then the set of live variables after i is:

$LV_{after}(i) = \{V \mid \exists P : V \in (pre\_inst(i, P) \cup out(i)) \cap (out(p) \cup post\_use(i, P))\}$.

If we call $in(i)$ the set of input variables to the literal at i, the set of live variables before i is:

$$LV_{before}(i) = (LV_{after}(i) \setminus out(i)) \cup in(i).$$

The LV_{before} of the first program point of an execution path of a procedure p is defined to be $in(p)$, the set of input variables of the procedure. The LV_{after} of the last program point is defined to be $out(p)$.

Live region variables. A region variable is live at a program point if its node is reachable from a live variable at the program point.

The set of nodes that are reachable from a variable is defined:

$$Reach(X) = \{n_X\} \cup \{m \mid \exists (n_X, m) \in E^*(X)\},$$

in which $E^*(X)$ is defined:

$$E^*(X) = \{(n_X, n_i) \mid \exists (n_X, sel_0, n_1), \ldots, (n_{i-1}, sel_{i-1}, n_i) \in E\}.$$

The live region variables sets before and after a program point i are defined:

$$LR_{before}(i) = \bigcup (Reach(X)) \ \forall X \in LV_{before}(i).$$
$$LR_{after}(i) = \bigcup (Reach(X)) \ \forall X \in LV_{after}(i).$$

4.2 Region Lifetime Across Procedure Boundary

The formal region parameters of a procedure are the region variables that are reachable from its head variables. For a procedure, we could safely require that the regions bound to by its formal region parameters are created and removed only by its callers. But we may achieve better memory use if we can give such a region a later creation or an earlier removal, or both, inside the procedure. To reason about this, for a procedure p, we define:

– $bornR(p)$ is the set of region parameters that are bound to regions which are created inside p, i.e., by p or by one of the procedures it calls.
– $deadR(p)$ is the set of region parameters that are bound to regions which are removed inside p.

When analysing a procedure p, initially, $bornR(p) = outputR(p) \setminus inputR(p)$; $deadR(p) = inputR(p) \setminus outputR(p)$, where $inputR(p)$ and $outputR(p)$ are the sets of region variables reachable from input and output variables, respectively.

The analysis then follows each execution path of p and applies the rules in Fig. 6 to any call q to update the $deadR$ and $bornR$ sets of q. A region variable is eliminated from $deadR(q)$ if it needs to be live after the call to q in p (i.e., the region to which it is bound must not be removed in the call to q) (rule L1); or if the region will be removed more than once by q because of the so-called "region alias", which means the same actual region parameter is used for more than one

formal region parameter (rule L2). A region variable is excluded from $bornR(q)$ if the region it is bound to is already live before the call to q (rule L3); or if q will create the region more than once due to region alias (rule L4). When there is a change to those sets of q, q needs to be analysed to propagate the change to its called procedures. Therefore, this analysis requires a fixpoint computation.

$$\frac{r \in LR_{before}(pp(l)) \quad r \in LR_{after}(pp(l)) \quad r = \alpha(r') \quad r' \in deadR(q)}{deadR(q) = deadR(q) \setminus \{r'\}} \text{ (L1)}$$

$$\frac{\alpha(r') = r \quad \alpha(r'') = r \quad r' \neq r'' \quad r' \in deadR(q)}{deadR(q) = deadR(q) \setminus \{r'\}} \text{ (L2)}$$

$$\frac{r \in LR_{before}(pp(l)) \quad r = \alpha(r') \quad r' \in bornR(q)}{bornR(q) = bornR(q) \setminus \{r'\}} \text{ (L3)}$$

$$\frac{\alpha(r') = r \quad \alpha(r'') = r \quad r' \neq r'' \quad r' \in bornR(q)}{bornR(q) = bornR(q) \setminus \{r'\}} \text{ (L4)}$$

$pp(l)$ returns the program point of the literal l.

Fig. 6. Live region analysis rules.

After this analysis, the set of region variables of a procedure, N, is partitioned into four sets: *deadR*, *bornR*, *constantR*, and *localR*, where $localR = N \setminus (inputR \cup outputR)$ and $constantR = N \setminus (deadR \cup bornR \cup localR)$. The first three sets are constituents of the set of region parameters of the procedure. The region parameters in *deadR* are live before a call to the procedure and the regions they are bound to can safely be removed inside the procedure. Those in *bornR* are unbound before a call to the procedure and will get bound inside the procedure. Those in *constantR* escape the procedure scope. The last set, *localR*, contains region variables that are absolutely internal to the procedure.

The algorithm to detect live region variables at each program point is an extension of live variable analysis, which is a standard, well-known program analysis [11]. The analysis in Section 4.2 aims to compute a shortest possible lifetime for a region. Its termination can intuitively be understood from the fact that each procedure uses a finite set of region variables, hence, initially *bornR* and *deadR* sets are also finite, and the analysis just reduces their size.

In the *qsort* program, *split* has three execution paths: $\langle(1),(2),(3)\rangle$, $\langle(4),(5),(6),(7)\rangle$, and $\langle(4),(8),(9)\rangle$; *qsort* has two: $\langle(1),(2)\rangle$ and $\langle(3),(4),(5),(6),(7)\rangle$. The LV and LR sets of *split* are in Table 1(a), of *qsort* in Table 1(b) (see also Fig. 2 and Fig. 3). Note that, in this example, it happens to occur that the set after one program point is always equal to the one before the next point in the same execution path. In general, this is not necessarily the case, consider for example a split point, the set of live region variables after it is the union of all the sets before its next points. The region parameter sets of the two procedures are: $deadR(split) = \{R1\}$, $bornR(split) = \{R3, R4\}$, $constantR(split) = \{R2, R5\}$; $deadR(qsort) = \{R1\}$, $bornR(qsort) = \phi$, $constantR(qsort) = \{R2, R3\}$.

5 Program Transformation

The main task of program transformation is to annotate the input program with *create* and *remove* instructions based on the region liveness information. It also annotates a procedure definition with formal region parameters known from the

pp	LV	LR
(1_b)	$\{X, L\}$	$\{R5, R1, R2\}$
$(1_a, 2_b)$	$\{\}$	$\{\}$
$(2_a, 3_b)$	$\{L1\}$	$\{R3, R2\}$
(3_a)	$\{L1, L2\}$	$\{R3, R2, R4\}$
(4_b)	$\{X, L\}$	$\{R5, R1, R2\}$
$(4_a, 5_b)$	$\{X, Le, Ls\}$	$\{R5, R2, R1\}$
$(5_a, 6_b)$	$\{X, Le, Ls\}$	$\{R5, R2, R1\}$
$(6_a, 7_b)$	$\{L2, Le, L11\}$	$\{R4, R2, R3\}$
(7_a)	$\{L1, L2\}$	$\{R3, R2, R4\}$
$(5_a, 8_b)$	$\{X, Le, Ls\}$	$\{R5, R2, R1\}$
$(8_a, 9_b)$	$\{L1, Le, L21\}$	$\{R4, R2, R3\}$
(9_a)	$\{L1, L2\}$	$\{R3, R2, R4\}$

(a) *split*

pp	LV	LR
(1_b)	$\{L, A\}$	$\{R1, R2, R3\}$
$(1_a, 2_b)$	$\{A\}$	$\{R3, R2\}$
(2_a)	$\{S\}$	$\{R3, R2\}$
(3_b)	$\{L, A\}$	$\{R1, R2, R3\}$
$(3_a, 4_b)$	$\{A, Le, Ls\}$	$\{R3, R2, R1\}$
$(4_a, 5_b)$	$\{A, Le, L1, L2\}$	$\{R3, R2, R4, R5\}$
$(5_a, 6_b)$	$\{Le, L1, S2\}$	$\{R2, R4, R3\}$
$(6_a, 7_b)$	$\{L1, A1\}$	$\{R4, R2, R3\}$
(7_a)	$\{S\}$	$\{R3, R2\}$

(b) *qsort*

Table 1. Live variable and live region variable sets in *qsort* program.

live region analysis. Actual region parameters at a call site can be derived from the formal region parameters of the called procedure and the α mapping at the call site.

Correctness conditions. To be correct, the transformation of a procedure p must ensure the following:

1. If a region variable is live at a program point, it must be bound to a region before that point.
2. If a region variable is not live at a program point, its binding status must be unbound before that point.
3. The regions to which the region parameters in $deadR(p)$ are bound will be removed inside p.
4. The regions to which those in $bornR(p)$ are bound will be created inside p.
5. No creation and removal will occur to those in $constantR(p)$ in the procedure.

Transformation. Each procedure is transformed by following its execution paths and applying the transformation rules in Fig. 7 to each program point so that the correctness conditions are respected. Assume that we are analysing a procedure p. Let l_i be the associated literal at a program point i in p. A literal can be either a specialized unification denoted by *unif* or a call (user-defined or built-ins). We assume that all the specialized unifications as well as calls to builtin operations do not remove or create any regions.

When a region variable first becomes live, namely when it is not live before i but is live after i, a region must be created and the region variable is bound to the region. If the region is created inside l_i, then no annotation is added at i. Otherwise the region is created either by a caller of p or by p itself. The former means that the region should not be created again in p, hence no annotation is added at i. The latter occurs when the region variable belongs to either $bornR(p)$ or $localR(p)$, and in this case we add a *create* instruction before l_i. This is reflected by the transformation rules T1 and T2.

When a region variable ceases to be live, the region it is currently bound to is removed. The first case is when the region variable is live before i but not live after i. If p does not remove the region, it is removed by a caller of p

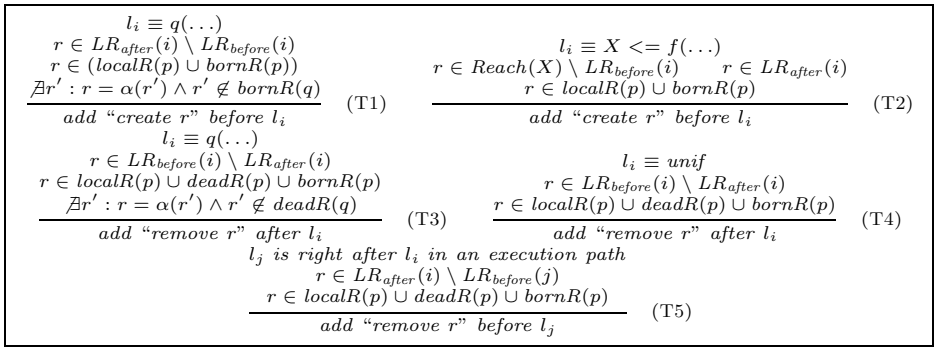


Fig. 7. Transformation rules.

and no annotation is introduced at i. Otherwise, the region is removed inside p. This means the region variable is in one of the *deadR*, *localR*, or *bornR* sets of p. There are two subcases: if l_i removes the region, then no *remove* instruction needs to be inserted at i; otherwise if p removes the region itself, we insert a *remove* instruction after l_i. The transformation rules T3 and T4 ensure this effect. While the reason for removing a region to which a region variable that belongs to the first two sets is bound is straightforward, the removal of a region that is bound to by a region variable in $bornR(p)$ is allowed because it is acceptable for p to remove the region after i and re-create it later on. The fact that region variable is in $bornR(p)$ ensures this. The second case is when the region variable is live after i, but not live before some program point j following i in a certain execution path. This can happen when i is a shared point among different execution paths and the region variable is live after i due to an execution path to which j does not belong. A *remove* instruction is added before l_j to remove the region as expressed by the transformation rule T5.

The result of the program transformation of the *qsort* program has been shown in Fig. 2. The addition of the *remove* instructions after the first program points in both *qsort* and *split* procedures results from the application of T4. Two *create* instructions inserted in the *split* procedure are effects of T2.

6 Support for Mercury Programs with Backtracking

The region analysis and transformation presented in Sections 3, 4, and 5 are correct for Mercury programs without backtracking. The region liveness analysis only takes into account forward execution and the transformation assumes that at runtime a program will follow only one execution path. To support backtracking the authors in [5, 7] described an enhanced runtime for Prolog with RBMM. The idea is to make backtracking transparent to the algorithm for non-backtracking programs by using a mechanism to undo changes to the region-based heap memory, restoring it to the previous state at the point to which the program backtracks. We reused this idea, developed the necessary details (which can be found in [13]) in the context of the Melbourne implementation of Mercury [14]. With this support, the algorithm for non-backtracking programs can be used unchanged to correctly support ones with backtracking.

7 Prototype Implementation and Experimental Results

We implemented an RBMM prototype using MMC version 0.12.0. It consists of an analyser that uses the region analysis to generate region-annotated programs and a *region simulator*. The analyser operates as one of the last analyses on the High Level Data Structure representation of the original source code and produces source code with region support. To have a working Mercury system with RBMM the runtime system of Mercury would need to be extended with support for regions, which is out of the scope of this paper. Instead, we developed a region simulator to experiment with region-annotated programs. The simulator can mimic the region operations in a region-annotated Mercury program as if the program were being executed by a working RBMM system. The simulator provides an interface with non-logical methods for region creation, removal, allocation into a region, and for supporting backtracking (see [13] for more details).

The program transformation emits a program (as valid Mercury code) in which *create* instructions are replaced by calls to the method for region creation in the simulator's interface and *remove* instructions by calls to the region removal method. After each construction a call to the region allocation method is added to collect the number of words allocated. Procedure definitions and calls are extended with region parameters as extra parameters. Calls to backtrack-supporting methods are inserted at suitable places. We also augment the Mercury runtime system to interact with the simulator so that the saving and restoration of region-based memory states can be imitated correctly.

Primitive types are not dealt with specially by the analyser, but they are by the simulator. 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.

When a Mercury program is executed it puts the terms that are created during execution on the heap. Assume that no RTGC is used, 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 the 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 cryptoarithmetic puzzle[1]. The experiments allow us to measure the memory used by the benchmarks.

Table 2 shows the experimental results obtained by the region simulator for the annotated versions of the benchmarks. The experiments were done on a PC

[1] The original and annotated source code of the benchmark programs can be found at http://www.cs.kuleuven.be/∼quan/benchmarks.tar.gz

with a 2.8 GHz Pentium 4 CPU, 512 MB of RAM running Debian GNU/Linux 3.1, under a usual load.

	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
SLR	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
AT (ms)	8	11	9	72	78	107	n/a	61	13
CT (ms)	251	243	223	329	343	361	n/a	307	273

Table 2. Experimental results.

Our measurements reported in Table 2 are: the total number of regions created during the execution of a benchmark (TR), the maximum number of regions coexisting during its run (MR), the total number of words allocated (TW), the maximum number of words that coexist (MW), and the size in words of the largest region (SLR). Then savings (S) can be computed as (TW-MW)/TW. We also give the times in milliseconds taken by our analysis (AT) and by the usual Mercury compilation of the benchmark (CT). For all benchmarks the analysis time is only a fraction of the compilation time.

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 of partitioning the heap into regions such that temporary data can be freed in a timely manner. 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 [16, 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 more preferable because it is entirely automated, hence freeing logic programmers from caring about memory management tasks. Moreover, the first solution requires extra time and memory for 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]. The reason for the better results is probably that our algorithm can remove a region earlier. For example, in the *qsort* procedure of the *qsort* program, the inference in [7] removes the region of the input list after the call to *split* (so the list and the two sublists are live at that point). In our case, that removal happens at the base case in the call to *split*, before creating the two sublists.

8 Conclusion

In this paper we have developed a region analysis algorithm for the typed logic programming language Mercury. Our approach was inspired by the work in [2]. The analyses in [2] take into account the data flow in a program in order to determine the regions and their lifetimes. Therefore the analyses had to be redefined for Mercury to deal with unification and a control flow which are fundamentally different from object manipulation and control flow in Java. Type information in Mercury has been exploited by the region analysis to achieve a finite region representation for recursive structures. Apart from that our algorithm allows interprocedural creation of regions which was not handled in [2]. The analyser annotates Mercury programs with instructions for RBMM. Experiments with the analyser for several small benchmarks (4-17 procedures, 70-500 lines of code) show that it takes a reasonable amount of time. We also implemented a region simulator which can run the annotated programs produced by the region analyser to imitate the effects of executing Mercury programs in an RBMM system. The memory use results of the benchmarks are positive, in some programs we obtain optimal memory consumption. This indicates that our method could be an interesting alternative memory management technique. 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 unable to measure runtime performance and the internal cost of the region allocator and to provide a thorough comparison between our approach and other memory management techniques available in Mercury, such as RTGC and CTGC. We are working on integrating the presented algorithm into the MMC and hope to lift this limitation soon.

We have not provided a formal proof for the whole algorithm and left it as future work. However, the foundations for the correctness of the algorithm have

been laid intuitively. Finally, we also would like to investigate some extensions to the presented algorithm: better region partition as mentioned in Section 7 and modular region analysis.

9 Acknowledgements

We would like to thank Zoltan Somogyi for his helpful comments on various parts of this work.

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. 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.

[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. 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.

[5] 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.

[6] H. Makholm. Region-based memory management in Prolog. Master's thesis, University of Copenhagen, 2000.

[7] 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.

[8] N. Mazur. *Compile-time garbage collection for the declarative language Mercury.* PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, May 2004.

[9] N. Mazur, G. Janssens, and M. Bruynooghe. A module based analysis for memory reuse in Mercury. In *Proceedings of Computational Logic*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1255–1269. Springer-Verlag, 2000.

[10] N. Mazur, P. Ross, G. Janssens, and M. Bruynooghe. Practical aspects for a working compile time garbage collection system for Mercury. In *Proceedings of the 17th International Conference on Logic Programming*, volume 2237 of *Lecture Notes in Computer Science*, pages 105–119. Springer, 2001.

[11] F. Nielson, H. R. Nielson, and C. Hankin. *The Principles of Program Analysis.* Springer, 1999.

[12] Q. Phan and G. Janssens. Towards region-based memory management for Mercury programs. In *CICLOPS*, 2006.

[13] Q. Phan and G. Janssens. A proposal for runtime region support for Mercury programs. Technical Report CW482, Department of Computer Science, Katholieke Universiteit Leuven, 2007.

[14] 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.

[15] 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.

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