More Precise Region-Based Memory Management for Mercury Programs *

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. Dividing the heap memory of programs into regions is the starting point of region-based memory management. In our existing work of enabling region-based memory management for Mercury, a program analysis was used to distribute data over the regions. An important goal of the analysis is to decide which program variables should end up in the same region. For a popular class of programs, it covetously puts program variables in the same region, while more memory could have been reused if they had been kept in separate ones. In this paper we define a new refined region analysis that is keen to keep program variables in separate regions by taking into account the different execution paths of a procedure. With the more precise, path-sensitive analysis we can reduce the memory footprint for several programs.

1 Introduction

Logic programming languages have a long tradition of freeing programmers from procedural chores such as manual memory management. A recent approach to automated memory management apart from runtime reference-tracing garbage collection is to rely on static analysis and program transformation that can approximate lifetime of program data and instruct the program to reuse dead memory at runtime. Region-based memory management (RBMM) follows this approach. It is based on statically dividing the heap memory into different parts, called regions, in which program terms are stored. Then the aim is to arrange this in such a way that the memory occupied by dead terms can simply be released by reclaiming their region as a whole. Recently region-based memory management has been made available in several mainstream logic programming systems, such as in Prolog [3] and in Mercury [4, 6].

In [4] and [6] we have developed an RBMM system for Mercury. The static region analysis and transformation was given in [4], which annotate programs with region instructions that take care of the timely creation and removal of regions and also of the adequate distribution of the terms over the regions. In [6] we described the runtime support needed for RBMM that could also handle backward execution. Although the system gave very promising results, in terms of memory consumption as well as runtime speedup, for almost all of the benchmarks [6],

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

the static region analysis in [4] sometimes too eagerly grouped variables into regions without taking into account different execution paths, reducing reuse opportunities.

Our contribution in this paper is the improvement of the region analysis and transformation in [4]. By distinguishing execution paths we obtain a more precise region allocation that ultimately leads to better memory reuse.

Section 2 motivates our new approach of dealing with different execution paths. Section 3 describes how we use, just as in [5], type-based graphs to model the regions of types. The concept of region points-to graphs and its extension with same-edges to keep apart regions are in Section 4. The region points-to analysis, Section 5, computes a safe region model and now is linked up with region liveness analysis in Section 6 to ensure the safety of same-edges. This link is described in Section 7. In Section 8 we show the impact of same-edges on the program transformation. Section 9 discusses and concludes.

2 Motivation

We assume the reader is familiar with Mercury [7]. The explicit declarations of types and modes in Mercury enables its compiler to convert all predicate definitions into procedures in superhomogeneous form in which unifications are specialized into `<=` for construction unifications, `=>` for deconstruction unifications, `==` for equality tests, and `:=` for assignments.

We will look at the relation between assignments and regions in particular. Consider the assignment `X := Y`, which binds the free variable `X` to the value of `Y`. For now we assume that the value of the variable `Y` is simply stored in one region. (We come back to the issue of storage of variables later in Section 3). As after the assignment `X` and `Y` are bound to the same value in a region, we can say that the two variables are in the same region.

The code in Figure 1 is a part of a Mercury program that manipulates lists of integers. We associate a *program point* with every literal (i.e., a specialized

```
% (in, in, out).                 % (in, out).
q(N, X, Z)  :-                  produce(X, Y) :-
( if                            (
(1) N > 0                       (1) X => [],
  then                          (2) Y <= []
(2) Z := X                      ;
  else                          (3) X => [Xe | Xs],
(3) produce(X, Y),              (4) produce(Xs, Ys),
(4) Z := Y                      (5) Y <= [Xe + 1 | Ys]
).                              ).
```

Fig. 1: The running example in the compiler-internal superhomogeneous form.

unification or a procedure call) in the body of a procedure. An *execution path* is a sequence of program points, such that at runtime the literals associated with these program points are performed in sequence. The procedure `q` has two execution paths: $\langle(1),(2)\rangle$ and $\langle(3),(4)\rangle$ in which different assignments for `Z` occur. In the second execution path `q` invokes `produce` that can assumably put its output argument `Y` in a region different from that of `X`. The code of

`produce` in Figure 1 follows this pattern. If after a call to `produce` its input is no longer needed, we can reclaim its memory by removing its corresponding region, without affecting the output. The call to `produce` at (3) in `q` is such a call. In `q`, after the assignment at (2) in the first execution path, that `Z` is bound to the list bound to by X implies that these two variables are in the same region. Similarly, at (4) in the second execution path we have that `Z` and `Y` are in the same region. So, it seems reasonable and it is actually safe to put *all* `X`, `Y` and `Z` in one region.

The region points-to analysis in [4] follows this "eager" approach. If the program follows the second execution path of `q`, the eager approach prohibits the reclamation of the memory of `X`, even when it is dead after the call to `produce` (recall that we assume that `X` and `Y` are independent). The memory for `X` could have been reclaimed if it had been kept apart in a different region from that of `Y` and `Z`.

Let us explore the idea of *keeping apart* the regions for `X`, `Y` and `Z` in order to be able to reclaim memory better. Thus we would like to keep `X`, `Y` and `Z` in different regions. An assignment like `Z := X` could then involve the copying of the value of `X` into the region of `Z`. Although this allows us to reclaim the memory of `X` in the else branch, it incurs overhead due to the copying, which is not desirable as it is linear to the size of the value. In this example, we can do better by *reusing* the region of `X` as the region of `Z` at (2) and the region of `Y` as the region of `Z` at (4), while the memory of `X` is reclaimed in `produce` as shown in Figure 2. We can view this as the result of distinguishing execution paths of

```
q(N, X@R1, Z@R3)  :-
( if
(1) N > 0
  then
(2) Z := X,
    reuse_for(R1, R3)
  else
(3) produce(X@R1, Y@R2),
(4) Z := Y,
    reuse_for(R2, R3)
).
```

```
produce(X@R4, Y@R5) :-
(
(1) X => [],
    remove(R4),
    create(R5),
(2) Y <= [] in R5
;
(3) X => [Xe | Xs],
(4) produce(Xs@R4, Ys@R5),
(5) Y <= [Xe + 1 | Ys] in R5
).
```

Fig. 2: Region-annotated version.

a procedure when dividing its variables into regions. In the first execution path, `X` and `Z` are in the same region while in the second one, `X` is in a region different from that of `Y` and `Z`.

In the annotated version, we use region variables `R`'s to refer to regions [6]. The region instruction `create(R)` creates a region and makes `R` bound to it while `remove(R)` removes the region that `R` is bound to. `reuse_for(R1, R2)` makes `R2` bound to the region bound to by `R1`, `R1` is considered dead after that. We use `@R` to illustrate the passing of regions as arguments and annotate construction unifications with `in R`, the region into which the allocation happens. Then the behaviour of the code is as follows. A caller of `q` prepares `X` in the region bound to by `R1` and expects `Z` in some region bound to by `R3`. If the first execution path of `q` is taken `R3` will be bound to the region `R1`. If the second path is, the call to `produce` removes `R1` and creates `R2` in its base case, then `R3` is bound to `R2`.

3 Modelling the Regions based on Types

In this section we explain the relationship between types and the storage of program variables in regions. The simplifications we make in the rest of this paper can easily be overcome, more details can be found in [5].

We use the type of a variable, which determines its values, to distribute such a value (i.e., the term bound to by the variable) over several regions. A list is stored in two regions: one for the list-skeleton and one for the elements of the list. A program, such as *quicksort* and *naive reverse* on lists, often creates several temporary lists but the elements of the input list are needed through out. When a temporary list is no longer needed, its list-skeleton as a whole can be freed by one single action namely by removing its region.

The regions can be derived from the type declarations [5]. Consider the following types.

```
:- type elem ---> f; g(int); h(list_int, int).
:- type list_elem ---> []; [elem | list_elem].
```

The principal functors of `list_elem` are `[]/0` and `[|]/2`. The memory needed to represent the `[|]/2` functor and its arguments is allocated into region *r1*. In the arguments we store pointers to the subterms. The first argument is a pointer to a value of type `elem`. Because this is a different type, it is stored into a different region *r2*. The second argument is again of type `list_elem` and we also store it in the region *r1*. In this way the list skeleton as a whole is in one region. The type `elem` has three principal functors: `f/0`, `g/1` and `h/2`. Suppose `L` has type `list_elem` and is bound to `[f,g(1),h([1,2],2)]`. To represent this list `L`, we need three regions. In the first we to store the principal functors of `list_elem`, the so-called list skeleton. The elements of the list are stored the second region, in particular the principal functors of the `elem` terms. Some arguments of the principal functors of `elem` have type `int` that does not require extra memory, so they are in the memory blocks of the `g/1` and `h/2` functors. The first argument of `h/2` is a `list` of `int`'s and needs one more region.

For each type we define a memory-storage scheme modelled by a **type-based region graph** $TG(N, E)$ with N a set of nodes and E a set of directed edges. A node stands for a region variable. Just as program variables get bound to ordinary Mercury terms during the execution of a program, region variables will get bound to (physical) regions. A directed edge from one node to another represents the fact that the region bound to by the region variable represented by the former node contains references into (points-to) the region bound to by the region variable represented by the latter one. The reference relation represented by the edges is actually defined by the type.

The type-based region graph for the type `list_elem` is shown in Figure 3. The `[|]` principal functor is stored in R^{list_elem}. Actually we need a block of two memory words. `[|]` has two arguments, the first having the type `elem` and the second having the same type `list_elem`. Thus we have two edges from R^{list_elem}, one pointing to R^{elem} where the principal functors of `elem` (`g/1` and `h/2`) are stored and the other is a self-edge. The edge labelled `(h,1)` is due to the first argument of the functor `h/2`.

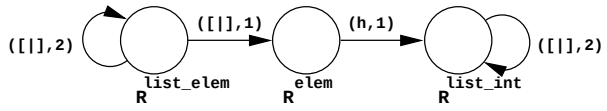

Fig. 3: The type-based region graph of the type `list_elem`.

Consider the following type:
`:- type t ---> ...; f(t1,..., ti,..., tn); ....`
If R^t is represented by the node n, then we have for each type `ti` that needs heap storage a node m representing R^{ti} and exactly one edge $(n, (f, i), m)$ with the label (f, i). We refer to n as the *principal node* of TG_t.

During the execution of the program, the regions used to allocate terms belonging to a type `t`, will be an instance of the memory-storage scheme modelled by TG_t, the type-based region graph for t.

4 Region Points-to Graph

We use the notion of a region points-to graph to model the memory used by a Mercury procedure. We use the set of type-based region graphs, each for each variable of the procedure, to adequately model the memory locations of the procedure's terms. However, we also need to model the sharing of (sub)terms among the program variables, which is created during the execution of the procedure. One of the contributions of this paper is the modelling of this sharing.

In Mercury, the instantiation of variables, therefore the sharing among them, is caused by unifications. We divide the sharing into two groups. First, a construction unification `X <= f(..., Y, ...)` allocates new memory for storing the functor `f` (actually the block of memory words corresponding to `f`) and creates sharing between `X` and `Y`. Also in a deconstruction unification `X => f(..., Y, ...)` `Y` is instantiated and `Y` shares with a subterm of `X`. The regions needed to store the values of `X` are given by its type-based region graph in which the edges point into the regions of subterms. In order to express this sharing between `X` and `Y`, we associate with a node n a set of program variables, $vars(n)$, whose principal functors are stored in the region that is bound to by the region variable that is represented by n. The sharing between `X` and `Y` (with `Y` the i^{th} argument of `X`) is then represented by having a node n_X with $X \in vars(n_X)$, an edge $(n_X, (f, i), n_Y)$ and $Y \in vars(n_Y)$. The *vars* set of a node can contain either zero, one or more than one variable. In the case where constructions and deconstructions involve variables of *recursive* types such as lists, e.g., `L <= [E | T]` or `L => [E | T]`, `L` and `T` are forced to end up in the same *vars* set. All the program variables in the *vars* set of a node may be allocated in the same region.

Second, an assignment unification `X := Y` binds `X` to `Y` and creates sharing between `X` and `Y`. Previously in [4], we did put the two variables in the same region, inspired by facilitating recursive types. In the new approach in this paper we keep their regions apart and just remember that they are candidates for reusing one for the other after this point. We represent this in a region points-to graph by a new kind of edges, called same-edges. The sharing created by `X :=`

`Y` is modelled by a *directed* edge $s(n_Y, i, n_X)$ with i the program point where `X := Y` is found. Note that normally the same-edge between n_X and n_Y should propagate to the regions of their corresponding subterms.

A **region points-to graph** for a set of variables V, $G(N, E, S)$, consists of a set of nodes, N, representing region variables, a set of directed edges, E, representing references between the regions bound to by these region variables and a set of directed same-edges, S, to model candidates for reuse. The nodes are annotated with *vars* sets: we have $V = \bigcup_{n \in N} vars(n)$. The node n_X denotes the node such that $X \in vars(n_X)$. The function $node(n_X, (f, i))$ returns the node m if $(n_X, (f, i), m) \in E$, otherwise it is undefined. The edges in S model regions that are possible candidates for reuse. Whether the reuse can safely be done depends on the liveness of the involved regions. This will become clear later.

The region points-to graphs of the procedures `q` and `produce` in our running example are in Figure 4. With same-edges (the dashed arrows) we can keep the

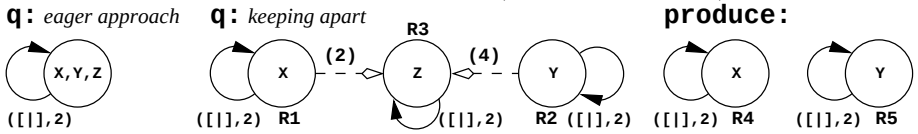

Fig. 4: Region points-to graphs of `q` and `produce`.

regions of `X`, `Y`, and `Z` separate (compare to the eager approach where we would have only one node with these variables in its *vars* set) while still be able to capture the fact that at some program points (i.e., at (2) and (4)) they are the same. Note again that we assume no regions for the integer elements of the list because they are stored right in the first word of a *cons* cell.

While locations are represented by the *vars* sets, all in all, *sharing* is represented in a region points-to graph in three ways. Firstly, the directed edges in E represent sharing of subterms. Secondly, that a *vars* set of a node may contain more than one variable represents the fact that these variables may be bound to the same term. Finally, sharing due to assignments is represented by the same-edges in S.

5 Region Points-to Analysis

The region points-to analysis computes a region model for a procedure and the whole program by capturing the locations and the sharing among variables. To capture sharing we use two operations: *unify* and *same_edge*. The operation *unify* is defined in Algorithm 1. Unifying n and m implies that the variables in $vars(n)$ and in $vars(m)$ are stored in the same region. To ensure that there is only one out-edge with a specific label from one node to another, the operation is recursive, i.e., unifying two nodes may cause more nodes to be unified.

The novelty in this paper (compare to [4]) is the second way to record sharing by using the *same_edge* operation that is defined by Algorithm 2. When we record a same-edge between two nodes we also need to recursively record same-edges for the corresponding nodes reached from them through corresponding edges.

Algorithm 1 $unify(n, m)$

Require: $G(N, E, S)$, $n, m \in N$.
Ensure: $G(N, E, S)$ with n representing the unified node.
$N = N \setminus \{m\}$
$vars(n) = vars(n) \cup vars(m)$
for all $(m, (f, i), k) \in E$ **do**
  if $(n, (f, i), k) \in E$ **then**
    $E = E \setminus \{(m, (f, i), k)\}$
  else
    $E = E \setminus \{(m, (f, i), k)\} \cup \{(n, (f, i), k)\}$
  end if
end for
for all $(k, (f, i), m) \in E$ **do**
  if $(k, (f, i), n) \in E$ **then**
    $E = E \setminus \{(k, (f, i), m)\}$
  else
    $E = E \setminus \{(k, (f, i), m)\} \cup \{(k, (f, i), n)\}$
  end if
end for
for all $s(m, i, k) \in S$ **do**
  if $s(n, i, k) \in S$ **then**
    $S = S \setminus \{s(m, i, k)\}$
  else
    $S = S \setminus \{s(m, i, k)\} \cup \{s(n, i, k)\}$
  end if
end for
for all $s(k, i, m) \in S$ **do**
  if $s(k, i, n) \in S$ **then**
    $S = S \setminus \{s(k, i, m)\}$
  else
    $S = S \setminus \{s(k, i, m)\} \cup \{s(k, i, n)\}$
  end if
end for
for all $l, l' \in N$ **do**
  if $(n, (g, j), l) \in E \wedge (n, (g, j), l') \in E \wedge l \neq l'$ **then**
    $unify(l, l')$
  end if
end for

Algorithm 2 $same\_edge(n, m, i)$

Require: $G(N, E, S)$, $n, m \in N$.
Ensure: $G(N, E, S)$ with same-edges between n and m and between any two corresponding nodes reachable from them.
$S = S \cup \{s(n, i, m)\}$
for all $(n, (f, i), k) \in E \wedge (m, (f, i), k') \in E$ **do**
  if $k \neq k' \wedge s(k, i, k') \notin S$ **then**
    $same\_edge(k, k', i)$
  end if
end for

5.1 Intraprocedural Analysis of a Procedure

The intraprocedural analysis initializes G_p and then captures the sharing created by the explicit unifications. Its definition is in Algorithm 3. The function $pp(l)$ returns the program point associated to the literal l.

Algorithm 3 *intraproc*(p): intraprocedural analysis of a procedure p

Require: p is in superhomogeneous form.
Ensure: Sharing created by explicit unifications is represented in G_p.
  $G_p = (\emptyset, \emptyset, \emptyset)$
  for all variable X in the procedure p **do**
    $G_p = G_p \uplus init\_rptg(X)$
  end for
  for all $unif \in p$ **do**
    if $unif \equiv (X := Y)$ **then**
      $same\_edge(n_Y, n_X, pp(unif))$
    else if $unif \equiv (X => f(Y_1, \ldots, Y_n)$ *or* $X <= f(Y_1, \ldots, Y_n))$ **then**
      for $i = 1$ to n **do**
        $unify(node(n_X, (f, i)), n_{Y_i})$
      end for
    end if
  end for

As we know the type of each variable in p, we initialize G_p by using the TG graphs of the variables. In Algorithm 3, we use the function $init\_rptg(X)$ that generates a region points-to graph for X from the type-based region graph of the type of X, $TG_{type(X)}$, by maintaining all the nodes and edges, but initializing the *vars* set of the node corresponding to the principal node in $TG_{type(X)}$ with $\{X\}$ and those of the other nodes with an empty set, generating a fresh region variable for each node in the region points-to graph, and setting the set of same-edges to empty.

The intraprocedural analysis adds all the sharing created by the unifications in the procedure to G_p. We ignore test unifications because they do not create any sharing. For construction and deconstruction unifications we unify the nodes corresponding with the sharing created by them. For an assignment we say that at its program point the two variables are bound to the same term by adding a same-edge from the node of the left-hand side variable to the node of the right-hand side one. This leaves the possibility that in other execution paths they are not necessarily bound to the same term, therefore they do not necessarily have to be in the same region either.

5.2 Interprocedural Analysis

The interprocedural analysis, Algorithm 4, updates G_p by integrating the *relevant* sharing information from the region points-to graphs of the called procedures into G_p.

For a call $q(Y_1, \ldots, Y_n)$, the head of the defining procedure is assumed to be $q(X_1, \ldots, X_n)$. The sharing among X_i's in G_q may not have been present in G_p as sharing among Y_i's. The interprocedural analysis makes sure that this will be the case. Firstly, it builds the function $\alpha : N_q \rightarrow N_p$ that maps the nodes of the formal arguments (X_i's) to the nodes of the corresponding actual arguments (Y_i's). Then these nodes are the starting points for the integration of the remaining sharing. This is done by following the relevant edges in G_q to

Algorithm 4 *interproc*(p): interprocedural analysis of a procedure p

Require: p is in superhomogeneous form.
Ensure: The sharing created by procedure calls is represented in $G_p(N_p, E_p, S_p)$.

```
repeat
    for all call site in p, at the program point i^p do
        Assume that the call is q(Y_1, ..., Y_n), with X_1, ..., X_n as corresponding formal arguments, and that G_q is available.

        % Build an α relation.
        for k = 1 to n do
            α(n_{X_k}) = n_{Y_k}
        end for

        % Ensure α is a function.
        for all X_i, X_j do
            if α(n_{X_i}) = n_{Y_i} ∧ α(n_{X_j}) = n_{Y_j} ∧ n_{X_i} = n_{X_j} ∧ n_{Y_i} ≠ n_{Y_j} then
                unify(n_{Y_i}, n_{Y_j})
            end if
        end for

        % Integrate sharing in G_q into G_p.
        In the graph G_q, start from each n_{X_i}, follow each edge once and apply the rules P1 and P2 in Figure 5 when applicable.
        for all s(n_q, _, m_q) ∈ S_q do
            if α(n_q) = n_p ∧ α(m_q) = m_p ∧ n_p ≠ m_p then
                same_edge(n_p, m_p, i_p)
            end if
        end for
    end for
until There is neither change in G_p nor in any of the α functions.
```

extend the α function to all the relevant nodes in G_q (rule P2) and to unify the relevant nodes in G_p (rule P1). Then we export the same-edges by relying on the α function at the call site. The program point of the same-edges is the program point of the call.

$$\frac{(n_q,(f,i),m_q)\in E_q \quad \alpha(n_q)=n_p \quad (n_p,(f,i),m'_p)\in E_p \quad \alpha(m_q)=m_p\neq m'_p}{unify(m_p,m'_p)}\ \text{(P1)} \qquad \frac{(n_q,(f,i),m_q)\in E_q \quad \alpha(n_q)=n_p \quad (n_p,(f,i),m_p)\in E_p \quad \alpha(m_q)\ undefined}{\alpha(m_q)=m_p}\ \text{(P2)}$$

Fig. 5: Interprocedural analysis rules.

For a whole program, we can first do the intraprocedural analysis for every procedure. Then given the fact that in the interprocedural analysis the analysis information is only propagated from graphs of callees to those of callers, we can do the interprocedural analysis for a program efficiently by decomposing the call-dependency graph into a tree of strongly connected components, and analysing the components in bottom-up order.

6 Region Liveness Analysis

After the region points-to analysis we know the region variables of each procedure and how the program variables are distributed over the regions to which these region variables are bound. As regions may need to exist through a sequence of procedure calls, e.g., a call may allocate memory into an existing region, we do pass region variables as arguments of procedures. We use the existing

region liveness analysis [4, 5] to decide which region variables are live at each program point and which region arguments become live or stop to be live in each procedure. In this section we summarize the relevant notions. Within the scope of a procedure we determine the local liveness of (program) variables and region variables in Section 6.1. The global liveness is discussed in Section 6.2.

6.1 Live Variables and Live Region Variables

We use the notions *before* and *after* a program point. Before a program point means right before the associated literal is going to be executed; while after a program point means its literal has just been completed. A program variable is live *before* a program point if it has been instantiated before the point and may be used in the future. A program variable is live *after* a program point if it has been instantiated before or at the point and may be used in the future. The live variable analysis computes for each program point i the set of variables that are live before i, $LV_{before}(i)$, and the set of variables that are live after i, $LV_{after}(i)$. The LV_{before} of the first program point(s) in a procedure `p` is defined to be the set of input arguments of `p`, $in\_args(p)$, while the LV_{after} of the last program point(s) in a procedure `p` is defined to be the set of output arguments of `p`, $out\_args(p)$.

A region variable being live means that it should be bound to a region and that it is possibly used in future (forward) execution. A region variable is live before (after) a program point if its node is reachable from a variable that is live before (after) the program point.

The set of nodes that are reachable from a variable is defined as follows.

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

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

$$E^*(X) = \{(n_X, n_i) \mid \exists (n_X, label_0, n_1), \ldots, (n_{i-1}, label_{i-1}, n_i) \in E\}.$$

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

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

6.2 Lifetime of Regions across Procedure Boundary

Region arguments are used to pass regions among procedure calls in order to achieve better memory reuse by keeping the lifetime of regions short. Therefore, the global liveness analysis part derives which region variables become live or cease to be live inside a procedure.

Consider a procedure q that is called by some procedure p, we define:

- $bornR(q)$ is the set of region variables of q that are mapped (by the α function at the call site) to region variables of p that definitely become live inside q, i.e., in q or in one of the procedures it calls.
- $deadR(q)$ is the set of region variables of q that are mapped to region variables of p that definitely cease to be live (or become dead) inside q.

– *outlivedR*(q) is the set of region variables of q that are mapped to region variables of p that outlive the call to q. They are live before the call and still live after the call.

The motivation is that, in the region-annotated program, the region variables of p that are mapped to by those in *bornR*(q) will get bound to a region during q and are still bound after q, the ones mapped to by those in *deadR*(q) are bound before the call to q and are safely removed during q, and the ones mapped to by those in *outlivedR*(q) are bound before the call and maintain their bindings throughout the call.

We call the set of the region variables that are local to p (not reachable from input or output variables), *localR*(p). The calling contexts of a procedure influence what a procedure can do to its non-local region variables. Therefore when analysing a procedure p, region variables that need to be live after the call to q in p are not allowed in *deadR*(q) but are put in *outlivedR*(q): the regions should not be reclaimed during the call. Similarly, regions already live before the call to q should not be in *bornR*(q) but in *outlivedR*(q): the regions already exist before the call. Also *region alias* has its impact. A typical case is when a procedure, e.g., $q(X_1, X_2)$, with $R_{X_1} \neq R_{X_2}$ is called as $q(Y_1, Y_2)$, with $R_{Y_1} \equiv R_{Y_2}$. Then R_{X_1} and R_{X_2} are neither in *deadR*(q) nor in *bornR*(q).

A procedure has exactly one *bornR* set and one *deadR* set suited for the most *restrictive* context. If the procedure is called in a less restrictive context, it will be the case that creation and removal will happen outside the call.

7 Good Same-Edges

The same-edges in our new region points-to graphs indicate regions that are candidates for reuse. For a procedure `q`, we keep a same-edge $s(R1, i, R2)$ with the intention that after the program point i we should *reuse* the region bound to by `R1` for `R2`, i.e., we make `R2` bound to the region currently bound to by `R1` and `R1` is considered unbound after that. However, it is only *safe* to do that if `R2` is not yet bound before i, if `R1` is not live after i, and if `q` is allowed to manipulate them. This implies the following *safeness conditions* for a same-edge $s(R1, i, R2)$: $R1 \in LR_{before}(i) \setminus LR_{after}(i) \wedge R2 \in LR_{after}(i) \setminus LR_{before}(i) \wedge R1 \in deadR(q) \cup localR(q) \wedge R2 \in bornR(q) \cup localR(q)$. The first two conditions can be seen as the local liveness requirements, while the global liveness requirements are in the last two. We call the same-edges that satisfy the conditions the *good* ones. Otherwise there is no point keeping them and their regions can be unified.

In Figure 6, we extend the running example with two calling contexts of `q` to show the effect of good and bad same-edges. In the first context, `X` is no longer used after the call to `q`. This program can be annotated as in Figure 7. Its behaviour is that the region of `X`, namely `R6`, is removed in the call to `q`. This is safe because `X` is no longer live after the call. `R7` is bound either to the region `R6` or to a new region created by the call to `produce` in `q` depending on which path is taken in `q`. But this does not matter because in `main` we can assume that `R6` is dead and `R7` becomes live. And we only need to reclaim `R7` afterwards.

```
% (in, in, out).
q(N, X, Z)  :-
( if
(1) N > 0
  then
(2) Z := X
  else
(3) produce(X, Y),
(4) Z := Y
).
```

```
% (in, out).
produce(X, Y) :-
(
(1) X => [],
(2) Y <= []
;
(3) X => [Xe | Xs],
(4) produce(Xs, Ys),
(5) Y <= [Xe + 1 | Ys]
).
```

```
% Calling context 1.
main(!IO) :-
(1) X <= [1],
(2) q(2, X, Z),
(3) L <= [Z],
(4) write(L, !IO).
```

```
% Calling context 2.
main(!IO) :-
(1) X <= [1],
(2) q(2, X, Z),
(3) L <= [X, Z],
(4) write(L, !IO).
```

Fig. 6: The running example extended with calling contexts of `q`.

```
q(N, X@R1, Z@R3)  :-
( if
(1) N > 0
  then
(2) Z := X,
    reuse_for(R1, R3)
  else
(3) produce(X@R1, Y@R2),
(4) Z := Y,
    reuse_for(R2, R3)
).
```

```
produce(X@R4, Y@R5) :-
(
(1) X => [],
    remove(R4),
    create(R5),
(2) Y <= [] in R5
;
(3) X => [Xe | Xs],
(4) produce(Xs@R4, Ys@R5),
(5) Y <= [Xe + 1 | Ys] in R5
).
```

```
main(!IO) :-
    create(R6),
(1) X <= [1] in R6,
(2) q(2, X@R6, Z@R7),
    create(R8),
(3) L <= [Z] in R8,
(4) write(L, !IO),
    remove(R7),
    remove(R8).
```

Fig. 7: Reuse region-annotated version for calling context 1.

In the second calling context, because `X` is still live after the call to `q`, the call is no longer allowed to remove `R6`. This means that `R1` is eliminated from $deadR(q)$ and therefore `R4` is excluded from $deadR(produce)$. The annotated version now is in Figure 8. In `produce` there is no `remove(R4)` after (1) and in

```
q(N, X@R1, Z@R3)  :-
((1) if N > 0
     then
 (2)   Z := X,
       reuse_for(R1, R3)
     else
 (3)   produce(X@R1, Y@R2),
 (4)   Z := Y,
       reuse_for(R2, R3)
).
```

```
produce(X@R4, Y@R5) :-
(
(1) X => [],
    create(R5),
(2) Y <= [] in R5
;
(3) X => [Xe | Xs],
(4) produce(Xs@R4, Ys@R5),
(5) Y <= [Xe + 1 | Ys] in R5
).
```

```
main(!IO) :-
    create(R6),
(1) X <= [1] in R6,
(2) q(0, X@R6, Z@R7),
    create(R8),
(3) L <= [X, Z] in R8,
(4) write(L, !IO),
    remove(R6),
    remove(R7),
    remove(R8).
```

Fig. 8: Wrong reuse region-annotated version for calling context 2.

`main remove(R6)` is added after (4). If the program follows the first execution path in `q`, `R6` and `R7` are bound to the same region. Therefore that region will be wrongly removed twice in `main`. In general, we cannot guarantee which execution path is taken at runtime. Therefore in this case it is not safe to make use of the (bad) same-edges.

A safe way to handle this situation is to force `X`, `Y`, and `Z` in `q` into the same region as in the eager approach in [4]. The annotated program is in Figure 9. If we used the eager version for the first calling context, we would miss the chance to reuse the memory of `X` whenever the second execution path of `q` is taken.

So after the region liveness analysis we will re-examine the region points-to graphs to eliminate the bad same-edges based on the above safeness conditions. If some same-edges are excluded from the region points-to graph of a procedure, we need to re-run the interprocedural analysis for the SCC that contains the

```
q(N, X@R1, Z@R1)  :-
( if
(1) N > 0
  then
(2) Z := X,
  else
(3) produce(X@R1, Y@R1),
(4) Z := Y
).
```

```
produce(X@R4, Y@R4) :-
(
(1) X => [],
(2) Y <= [] in R4
;
(3) X => [Xe | Xs],
(4) produce(Xs@R4, Ys@R4),
(5) Y <= [Xe + 1 | Ys] in R4
).
```

```
main(!IO) :-
    create(R6),
(1) X <= [1] in R6,
(2) q(0, X@R6, Z@R6),
    create(R8),
(3) L <= [Z] in R8,
(4) write(L, !IO),
    remove(R6),
    remove(R8).
```

Fig. 9: Region-annotated version with the eager approach.

procedure and for the SCCs that depend on it. The live variable analysis does not need to be run again. But the live region variable analysis and the analysis that computes *bornR* and *deadR* sets of the affected procedures needs to be re-executed. We can loop through these analyses until no same-edge is removed. At that time, all the remaining same-edges imply that reuses can safely happen.

8 Transformation

A region-annotated program is finally generated by a program transformation of the original program taking into account the information derived by the region points-to analysis and the liveness analysis.

The transformations for adding extra arguments for region arguments, for annotating construction unifications with region variables, and for introducing the `create` and `remove` instructions are exactly the same as presented in [5]. Here we focus on the introduction of the `reuse_for` instructions.

Because all the remaining same-edges are good ones, if the literal at the program point i of the same-edge $s(R1, i, R2)$ is an assignment, we just add `reuse_for(R1, R2)` after i. If the literal is a procedure call it means that the reuse happens inside the call and we do not need to make any change.

We can see the effect of same-edges at procedure calls when changing the code of `q` by replacing the assignment at (4) with a recursive call as in Figure 10. No `reuse_for` needs to be added after (4).

```
q(N, X@R1, Z@R3)  :-
( if
(1) N > 0
  then
(2) Z := X,
    reuse_for(R1, R3)
  else
(3) produce(X@R1, Y@R2),
(4) q(N - 1, Y@R2, Z@R3)
).
```

```
produce(X@R4, Y@R5) :-
(
(1) X => [],
    remove(R4),
    create(R5),
(2) Y <= [] in R5
;
(3) X => [Xe | Xs],
(4) produce(Xs@R4, Ys@R5),
(5) Y <= [Xe + 1 | Ys] in R5
).
```

```
main(!IO) :-
    create(R6),
(1) X <= [1] in R6,
(2) q(2, X@R6, Z@R7),
    create(R8),
(3) L <= [Z] in R8,
(4) write(L, !IO),
    remove(R7),
    remove(R8).
```

Fig. 10: No `reuse_for` after procedure calls.

9 Discussion and Concluding Remarks

In the paper we have presented an improvement for the region analysis and transformation in [4] that results in better memory reuse for a certain class of

programs. By making the region points-to analysis path-sensitive we achieve a more precise region model of memory use. This information then is verified against the liveness information to ensure that the transformation is sound. We formulate the local and global liveness safeness conditions for this purpose.

Implementing the improvement presented in this paper is part of future work. It should likely not require much extra runtime support from [6].

The class of programs that potentially benefit from this novel extension includes state transition programs in which a sequence of states are computed iteratively and typically we are only interested in the last state. A typical example is the program Game of Life in which a new generation is generated from a previous one based on a set of production rules. Its code follows the pattern of `q` and `produce` in Figure 10 in which `produce` implements the production rules and `q` is the loop for computing next generations. As reported in [6], without region reuse, the program needs maximally 8208 words to run. A closer look into the program reveals that the skeletons of all the generations, which are lists of live cells, are stored in one region with a size of 6486 words. This region is the biggest region of the program. The cells are stored in another region. Because they are actually shared among the generations, region reuse cannot help to separate them into different regions. However, region reuse can help with splitting the skeletons and then reclaiming them. We mimic region reuse for this program by replacing the assignment between two generations (like the one at (2) in `q`) with a call to a procedure that copies the list skeleton: the skeletons of the two lists are in two different regions. By evaluating the region reuse-mimicking program we measure that the maximal number of words used is reduced by 77% to 1856 words due to the fact that the garbage consisting of the skeletons of the old generations and of temporary data created during the process of generating a new one is now timely reclaimed.

Birkedal et al. [1] present *Storage Mode Analysis* that targets the same class of programs in functional programming. Being an extra phase after the region inference in [8] that puts all such states into the same region, the analysis then aims to *reset* the region before each iteration if it is safe to do so. The decision is also based on liveness information. However for Game of Life, it requires manually rewriting the program with a copying function so that the resetting is possible [2]. Henglein et al. [2] develop an expressive region type system that can accept several region-annotated versions for a program. Their region inference based on that type system also can produce an annotated version with the same region behaviour for Game of Live as ours, without requiring rewriting. One interesting open problem with their region type system is to have a strong region inference that, among a number of accepted annotated programs, can choose to generate an optimal one. We start from an analysis algorithm that performs well generally and extend it to obtain better results in a popular pattern of code, which is well known to be difficult for RBMM.

The attentive reader might get worried about programs such as `qsort` in Figure 11. In the approach in [4] all the variables involved with the accumulating parameter and the result are put in the same region. This is perfect because the accumulator is gradually built up to become the final result. Our new approach

```
% Original procedure.
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)
).
```

```
% Region-annotated version.
qsort(L@R1, A@R2, S@R3) :-
( (1) L => [],
  (2) S := A
      reuse_for(R2, R3)
;
  (3) L => [Le | Ls],
  (4) split(Le, Ls@R1, L1@R4, L2@R5),
  (5) qsort(L2@R5, A@R2, S2@R6),
  (6) A1 <= [Le | S2] in R6,
  (7) qsort(L1@R4, A1@R6, S@R3)
).
```

Fig. 11: A chance for optimization.

seems to be spoiling this. First notice that `A1` and `S2` are in the same region `R6` due to the construction. In the first execution path we have a same-edge at (2) from the region `R2` of `A` to the region `R3` of `S`. In the second execution path we have a same-edge at (5) from `R2` to `R6` and a same-edge at (7) from `R6` to `R3`. All same-edges are "safe". The resulting region-annotated program has a `reuse_for` instruction at (2). Actually, this instruction is an overhead. It would be removed by the following optimization step. For `qsort` in all execution paths we have a same-edge *path* from the region `R2` of `A` to the region `R3` of `S`. Thus, we could put all the involved variables again in one region. Note that the optimization condition does not hold for e.g., `q` in Figure 10. More benchmark programs will have to be studied in order to know the relevance and the impact of the above optimization step.

Acknowledgements. We would like to thank the anonymous referees for their helpful comments.

References

1. L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Principles of Programming Languages.*, pages 171–183. ACM Press., 1996.
2. 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.
3. 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.
4. 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.
5. Q. Phan and G. Janssens. Region-based memory management for Mercury programs. Part 1: Region analysis and transformation. Technical Report CW540, Department of Computer Science, Katholieke Universiteit Leuven, 2009.
6. Q. Phan, Z. Somogyi, and G. Janssens. Runtime support for region-based memory management in Mercury. In *Proceedings of the 2008 International Symposium on Memory Management*, pages 61–70. ACM Press., 2008.
7. 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.
8. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation.*, 132(2):109–176, February 1997.