# Region-based memory management for Mercury programs. Part 1: Region analysis and transformation

Quan Phan Gerda Janssens Report CW 540, April 2009



# Katholieke Universiteit Leuven Department of Computer Science

Celestijnenlaan 200A - B-3001 Heverlee (Belgium)

# Region-based memory management for Mercury programs. Part 1: Region analysis and transformation

Quan Phan Gerda Janssens Report CW 540, April 2009

Department of Computer Science, K.U.Leuven

#### Abstract

Region-based memory management is a technique to do compiletime memory management based on the idea of dividing the heap memory into different regions such that, at runtime, memory can be reclaimed automatically by destroying regions in their entirety. This report contains a static region analysis for the logic programming language Mercury. We define region points-to graphs, which represent the locations of terms and the sharing among them, to model the partitioning of the memory used by a program into separate regions. The static analysis starts with a region points-to analysis that determines the different regions in the program. We then compute the liveness of the regions by a region liveness analysis. Finally, a program transformation adds sufficient region annotations to the program for region support. In this report we also give the correctness proofs for the analysis, so that the safety of memory accesses can be guaranteed. The implementation of the runtime support for region-based memory management in the Mercury compiler will be described in a second report.

Keywords : Program Analysis, Mercury, Region-based memory management. CR Subject Classification : D.3.4, I.2.3

# Region-Based Memory Management for Mercury Programs. Part 1: Static analysis and transformation. \*

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 technique to do compiletime memory management based on the idea of dividing the heap memory into different regions such that, at runtime, memory can be reclaimed automatically by destroying regions in their entirety. This report contains a static region analysis for the logic programming language Mercury. We define region points-to graphs, which represent the locations of terms and the sharing among them, to model the partitioning of the memory used by a program into separate regions. The static analysis starts with a region points-to analysis that determines the different regions in the program. We then compute the liveness of the regions by a region liveness analysis. Finally, a program transformation adds sufficient region annotations to the program for region support. In this report we also give the correctness proofs for the analysis, so that the safety of memory accesses can be guaranteed. The implementation of the runtime support for region-based memory management in the Mercury compiler will be described in a second report.

# 1 Introduction

In region-based memory management (RBMM) the heap memory is organized into different areas, called regions. The program terms are allocated in these regions in such a way that the regions can be freed as soon as the terms they contain are dead. Consider the well-known *quicksort* program in Figure 1. One typical use for regions is to store temporary data. In this example split/4 creates two temporary lists that are input for the recursive calls to qsort. The first argument of qsort is an input list that is no longer needed when its sorted version is computed. For example, the temporary list L2 created by split can be put in a separate region that can be freed once qsort(L2,A,S2) has dealt with it.

To run this program with RBMM the following things need to be done:

1. Figure out how to distribute the program data over regions: our region points-to analysis (Section 5) computes the regions for the terms in a Mercury program.

 $<sup>^{\</sup>star}$  This work is supported by the project GOA/2003/08 and by FWO Vlaanderen.

Fig. 1: The *quicksort* program.

- 2. Decide when to create and to free regions: we derive the liveness information of the regions (Section 6).
- 3. Adapt the Mercury system to support RBMM: based on the region points-to analysis and the region liveness information the original Mercury program is transformed into a region annotated program (Section ??) which contains enough additional information to be executed by the Mercury system enhanced with RBMM runtime support.

The idea is that all the above steps are provided automatically in the RBMMenabled Mercury system. In this paper we present the program analyses while the runtime support will be described in a separate report.

# 2 Background

#### 2.1 Mercury

Mercury is a pure logic programming language intended for the creation of large, fast, reliable programs [14]. While the syntax of Mercury is based on the syntax of Prolog, semantically the two languages are very different due to Mercury's purity, its type, mode, determinism and module systems, and its support for evaluable functions. (Mercury treats functions as predicates with the return value as an extra argument, so in the rest of the paper we will talk only about predicates.)

Mercury has a strong Hindley-Milner type system very similar to Haskell's. Apart from some special types that are builtin (e.g., int), users can introduce types by type declarations such as in Example 1.

```
Example 1. The declaration of the type list_int.
:- type list_int ---> []; [int | list_int].
It declares the type for lists of integers, int is a builtin type in Mercury.
```

Mercury programs are statically typed; the compiler knows the type of every argument of every predicate (from declarations or inference) and every local variable (from inference).

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

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

*Example 2.* The *quicksort* program written in Mercury with declarations of types, modes, and determinisms for two essential predicates, **qsort** and **split**, is shown in Figure 2. Some specific elements in **main** are included for completeness but have no importance to the understanding of the paper. These include **!IO**, which are state variables representing the states of the world for declarative in-put/output, and **io.write**, a predicate from the **io** library module, that writes out a term.

main(!IO) :-	:- type split(int, list_int, list_int,
qsort([2, 3, 1], [], S),	list_int).
io.write(S, !IO).	:- mode split(in, in, in, out) is det.
	split(_, [], [], []).
<pre>:- type qsort(list_int, list_int, list_int).</pre>	split(X, [Le   Ls], L1, L2) :-
:- mode qsort(in, in, out) is det.	( if X >= Le then
qsort([], A, A).	<pre>split(X, Ls, L11, L2),</pre>
qsort([Le   Ls], A, S) :-	L1 = [Le   L11]
split(Le, Ls, L1, L2),	else
qsort(L2, A, S2),	<pre>split(X, Ls, L1, L21),</pre>
qsort(L1, [Le   S2], S).	L2 = [Le   L21]
-	)

Fig. 2: The quicksort program in Mercury.

The subset of Mercury [11] we deal with in this paper does not support higher order programming (including typeclasses), or predicates and functions defined by foreign language code. A complete description of the Mercury language can be found in [11].

#### 2.2 Mercury Code inside the Compiler

The compiler converts all predicate definitions into an internal form. For our subset of Mercury, this internal form is given by the following abstract syntax:

```
predicate P: p(x_1, \ldots, x_n) \leftarrow G

goal G

(G_1, \cdots, G_n) \mid x = f(y_1, \ldots, y_n) \mid p(x_1, \ldots, x_n)

(G_1, \cdots, G_n) \mid (G_1; \ldots; G_n) \mid not \ G \mid

(if \ G_c \ then \ G_t \ else \ G_e) \mid some \ [x_1, \ldots, x_n] \ G
```

The first three kinds of goals including unifications and calls are called literals or atoms. The rest are called compound goals, in which a sequence of goals separated by commas is a conjunction, while a sequence of goals separated by semicolons is a disjunction.

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

Inside the compiler, every goal (compound goals as well as literals) is annotated with mode and determinism information. For unifications, we show the mode information by writing <= for construction unifications, => for deconstruction unifications, == for equality tests, and := for assignments. The compiler reorders conjunctions as needed to ensure that goals that consume the value of a variable always follow the goal that produces its value. We show the *quicksort* program in this abstract syntax in Figure 3. For readability, we have chosen

```
main(!IO) :-
                                           split(X, L, L1, L2) :-
(1) L <= [2, 1, 3],
(2) A <= [],
                                           (1) L => [],
(3) qsort(L, A, S)
                                           (2) L1 <= [].
(4) io.write(S, !IO),
                                           (3) L2 <= []
gsort(L, A, S) :-
                                           (4) L \Rightarrow [Le | Ls].
                                           (5) ( if X \ge Le then
(1) L => [],
                                                   split(X, Ls, L11, L2),
                                           (6)
(2) S := A
                                                   L1 <= [Le | L11]
                                           (7)
                                                  else
(3) L => [Le | Ls],
                                           (8)
                                                   split(X, Ls, L1, L21),
(4) split(Le, Ls, L1, L2),
(5) qsort(L2, A, S2),
                                           (9)
                                                   L2 <= [Le | L21]
                                               )
(6) A1 <= [Le | S2],
                                           ).
(7) qsort(L1, A1, S)
).
```

Fig. 3: quicksort program in superhomogeneous form.

meaningful names for some additional variables that are added automatically by the Mercury compiler. Moreover, we also hide the fact that the Mercury compiler converts syntactic sugar, such as the list construction at (1) in main in Figure 3, into suitable unifications in the above-mentioned syntax. That construction would be extended as follows, which is lengthy and is of no interest in this paper.

V\_0 <= [], V\_1 <= 3, V\_2 <= [V\_1 | V\_0], V\_3 <= 1, V\_4 <= [V\_3 | V\_2], V\_5 <= 2, L <= [V\_5 | V\_4]

In the rest of the paper, we will ignore negation, since not G can be implemented as if G then fail else true in which fail and true are two builtin

goals, fail always fails while true always succeeds. Note that in Mercury (unlike in Prolog), the condition of an if-then-else may succeed several times. This will be clear from the determinism annotation on the goal representing the condition, and many parts of the compiler, including the implementation of RBMM, handle conditions of different determinisms differently.

Another situation in which determinism information is important is existential quantification. (Mercury also supports universal quantification, but the compiler internally converts all  $[x_1, \ldots, x_n]$  G to not some  $[x_1, \ldots, x_n]$  not G, so we do not have to deal with it.) If some  $[\ldots]$  G quantifies away all the output variables of G, then different solutions of G would be indistinguishable, so even if G can have more than one solution, some  $[\ldots]$  G will not. We call such a quantification a *commit*, and we handle commits differently from other quantifications.

# **3** Region-Based Memory Management for Mercury

We divide the task of realizing RBMM for Mercury between static analyses and transformation and dynamic runtime support. The goal of the static analyses and transformation is to annotate a normal Mercury program with information about regions. An annotated program contains information about the regions in which terms are constructed and when regions are created and freed. To obtain this information, we first use a region points-to analysis to detect the regions used by a program. After that we compute the lifetime of these regions by using a region liveness analysis. The program transformation uses these pieces of information to convert the program into a region-annotated program.

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

The runtime support for RBMM has two main purposes. Firstly, we need to enhance the runtime system of Mercury to work with the heap memory organized in terms of regions. Secondly, we provide support for backtracking in the context of RBMM, including the support for backward liveness and for instant reclaiming. The discussion of the necessity of this support in detail can be found in [13].

In region-annotated programs, which are the result of the static part, we use *region variables* to refer to regions, just as we use program variables to refer to values. To allocate a new region, we use the instruction create(R), which creates a region and binds the region variable R to it. To free a region we use the instruction remove(R), which frees the memory of the region to which

**R** is currently bound. Our regions can (and actually do) live across procedure boundaries and thus we pass region variables as extra arguments to procedure calls. Figure 4 shows the region-annotated *quicksort* program after our region transformation.

```
main(!IO) :-
                                             split(X, L@R1, L1@R3, L2@R4) :-
    create(R20), create(R21),
(1) L <= [2, 1, 3] in R20,
                                             (1) L => [],
    create(R22)
                                                 remove(R1),
(2) A <= [] in R22
                                                 create(R3).

(3) qsort(L@R20, A@R22, S@R22),
(4) io.write(S, !IO),

                                             (2) L1 <= [] in R3,
                                                 create(R4),
    remove(R21), remove(R22).
                                             (3) L2 <= [] in R4
qsort(L@R6, A@R8, S@R8) :-
                                             (4) L => [Le | Ls]
                                             (5) ( if X \geq Le then
(1) L => [],
                                                       split(X, Ls@R1, L11@R3, L2@R4),
                                             (6)
    remove(R6).
                                                      L1 <= [Le | L11] in R3
                                             (7)
(2) S := A
                                                    else
                                             (8)
                                                      split(X, Ls@R1, L1@R3, L21@R4),
(3) L => [Le | Ls],
                                             (9)
                                                      L2 <= [Le | L21] in R4
(4) split(Le, Ls@R6, L1@R9, L2@R10),
                                                 )
(5) qsort(L2@R10, A@R8, S2@R8),
                                             ).
(6) A1 <= [Le | S2] in R8,
(7) qsort(L1@R9, A1@R8, S@R8)
)
```

Fig. 4: Region-annotated quicksort program.

In the region-annotated code, we use the postfix **@Ri** to annotate both actual and formal arguments with their region variables. We also annotate each unification that constructs a new memory cell with the region in which the cell will be allocated. For example, in main, the skeleton of the list L is in the region (bound to by) R20, that of the accumulator A is in R22. We assume that the elements of the lists are in R21. In the call to qsort, R20 and R22 are passed as actual region arguments, corresponding to the formal arguments R6 and R8 in the definition of qsort. We do not need to pass the region of the elements because qsort and split just read from it. The region R20 is passed to qsort from main and is removed in the base case branch of split in the call to split at (4) in qsort. The two new lists L1 and L2 are allocated in two separate regions referred to by R9 and R10. These regions are created by the base case branch of split, and removed (indirectly) by the recursive calls to qsort 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 R22 of the resulting list is the region of the accumulator, which is created in main.

# 4 Region Modelling

## 4.1 Storing Terms in Regions Based on Their Types

As we want to distribute terms over different regions, we first discuss the representation of terms when the heap memory is divided into regions.

We assume that a term that can be represented by a single memory word does not require heap storage. A term that does not fit into a word is represented by a pointer to memory locations on the heap.

Our assumptions are compatible with the implementation of Mercury in the Melbourne Mercury Compiler (MMC). In the MMC, types, which are known at compile-time, give us information about the storage size of terms. Terms of primitive types such as int, char and of enumeration types of which *all* functors have arity zero are stored in one word. The principal functor of a term that needs heap space is stored by a *possibly-tagged* pointer to a block of memory words on the heap. The compiler knows all the functors in the type of the term. Therefore the principal functor does not need to be explicitly recorded in the block (as in a common implementation of Prolog) but can be tagged in the lowest bits of the pointer. When a type has only one functor the tag is even not needed. So the memory block on the heap is mostly for storing the arguments of the functor. When there are more functors than can be encoded by the available lower bits, the first word of the memory block is also used as a secondary tag to distinguish them.

Example 3. Consider the following types.

:- type elem ---> f; g(int); h(list\_int, int).

:- type list\_elem ---> []; [elem | list\_elem].

Figure 5 shows the representation of a term bound to by the variable L of the type list\_elem in the MMC. The box with a slim border is a location on the stack or in a register, one with a bold border is a location on the heap. Note the storage of a functor such as h/2 of the last element of the list. We need a two-word block for its arguments, the functor itself is stored implicitly in the tagged pointer.



Fig. 5: Term representation of L=[f, g(1), h([1, 2], 2)].

We now consider the storage of terms when the heap is split into regions. The idea is to use different regions to store different parts of a term so that we can reclaim the memory of a part by destroying its region as soon as the part becomes dead. In list-based programs, such as *quicksort*, we often see that a program creates several temporary lists but the elements of the input list are needed through out. Therefore it will serve our purpose if we store the elements and the skeletons in different regions. Generalizing from this we divide a term into regions based on its type. We will develop this idea in the next subsection.

In Figure 5 the regions used to store that term are visualized by the dashed lines. We put the two-word memory blocks making up the skeleton of the list L into one region because they have the type <code>list\_elem</code>. We also put *all* the elements, which have the type <code>elem</code>, into another region. Finally, the second subterm of the third element, which is a list of integers, is also stored in a separate region.

The representation of the list of integers here seems inconsistent with what we said in Section 3, where we have an extra, separate region for the integers. The reason for this is because in this section we want to give a region model as close as possible to the implementation of Mercury in the MMC. Strictly speaking, whether a type needs heap storage or not depends on implementation. For convenience, we take the liberty to switch between the two options. When talking about theoretical topics such as static analyses and transformation for convenience we generally assume that all types (also int) require heap storage. Otherwise, for example when we talk about regions for a type, we assume that its terms actually need to be stored on the heap. We will be more specific only if the context is not clear.

#### 4.2 Modelling Regions of a Type

We want to have a storage scheme that specifies how the terms of a type are stored. Consider a type t declared as follows.

```
:- type t ---> ...; f(t1,..., ti,..., tn); ...
```

We associate a region variable  $R^t$  to the type. The block of memory words corresponding to a principal functor, such as f, of a term of the type t is stored in the region bound to by  $R^t$ . In the rest of the paper we abbreviate this by simply saying that a principal functor is stored in  $R^t$ . The principal functor of an argument of f that has type ti is stored in the region bound to by  $R^{ti}$ , which is associated to ti. If a type t is recursive or mutually recursive we still use only one region variable  $R^t$ . This implies that any term of a recursive type is stored in a finite number of regions.

We model the storage scheme using 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. 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.

Consider the type-based region graph of the type t,  $TG_t$ , with the region variables  $R^t$ ,  $R^{ti}$ 's, and so on. If  $R^t$  is represented by the node n, then for each node m representing  $R^{ti}$ , we have exactly one edge (n, (f, i), m) with the label (f, i). We refer to n as the *principal node* of  $TG_t$ .

Example 4. The type-based region graph for the type list\_elem in Example 3 is shown in Figure 6. The [] principal functor is stored in  $R^{list\_elem}$ . It 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. The reader may want to compare this type-based region graph with the memory representation of a given term of this type in Figure 5.



Fig. 6: The type-based region graph of the type list\_elem.

*Example 5.* Consider the following types t1 and t2, which are mutually recursive.

:- type t1 ---> f(int, t2). :- type t2 ---> g(t1, int).

The type-based region graph for them is as in Figure 7.

Fig. 7: Type-based region graph of mutually recursive types.

# 4.3 Region Points-To Graph

Now that we have the region model for types, our next goal is to model the memory used by a Mercury program in terms of regions. A program consists of a set of procedures, each having its own set of program variables that, at runtime, are instantiated with relevant terms. Therefore we define the notion of a region points-to graph that models the memory used by a set of variables. The memory used by a procedure is then modelled by a region points-to graph for its variables. Finally, the memory model for the whole program is expressed through the region points-to graphs of its procedures.

In Mercury, the instantiation of variables is caused by unifications. A construction unification  $X \leq f(..., Y, ...)$  allocates new memory for storing the functor f (actually the block of memory words corresponding to f) and creates sharing between X and Y. In a deconstruction unification  $X \Rightarrow f(..., Y,$ ...) or an assignment unification Y := X, Y is instantiated and shares a subterm or the whole term with X, respectively. Hence the region points-to graph should capture the memory locations of the variables and the sharing among them.

A region points-to graph for a set of variables V, G(N, E), consists of a set of nodes, N, representing region variables and a set of directed edges, E, representing references between the regions bound to by these region variables. The edges here serve exactly the same purpose as those in a TG graph. However, each node n in the region points-to graph has an associated 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. Note that this set can be empty. We have  $V = \bigcup_{n \in N} vars(n)$ . The node  $n_X$  denotes the node such that  $X \in vars(n_X)$  and we refer to  $n_X$  as the *location* of X where the principal functor of the term that X is bound to is stored. The function  $node(n_X, (f, i))$  returns the node m if  $(n_X, (f, i), m) \in E$  otherwise it is undefined. Sharing is represented in a region points-to graph in two ways. Firstly, directed edges represent sharing of subterms and 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. An example of the latter is given by the variables of an assignment unification: they are bound to the same term and therefore they should be in the vars set of the same node. A region points-to graph represents sharing at the level of the regions.

**Definition 1 (Region-sharing in a region points-to graph).** Two variables X and Y region-share in a region points-to graph if there exists a node that can be reached from  $n_X$  and  $n_Y$ .

For convenience, we also say a node represents a region, by that we mean the region to which the region variable represented by the node is bound at runtime. Then we can say a functor is stored in a node meaning that the functor (i.e., the memory block corresponding to it) is stored in the region represented by the node.

For a procedure p, we often denote its region points-to graph by  $G_p(N_p, E_p)$ and  $G_p$  should represent the locations and sharing among all the variables in p. It is possible to form a region points-to graph for a procedure exactly from the type-based region graphs of all of its variables (whose types are known to the compiler). Although this region points-to graph adequately models the locations of the procedure's relevant terms it does not represent the sharing among them. Actually as we will see in Section 5 we use that region points-to graph as the starting point in our region points-to analysis of a procedure with the ultimate aim of producing a region points-to graph that also represents all the *possible* sharing among the procedure's variables.

*Example 6.* Consider the following sequence of code to construct the term that L in Example 3 is bound to. The type of K is of no importance.

X <= [1, 2], X <= [1, 2], Y := X, Z <= h(Y, 2), L <= [f, g(1), Z], K <= k(Z), ...

The region points-to graph that represents the memory manipulated by this sequence is shown in Figure 8. X and Y are in the vars set of the same node because the assignment makes Y point to the term to which X is bound. The direct sharing between Z and Y, and between L and Z is represented by the edges between their corresponding nodes. The indirect sharing between L and Y is modelled by the fact that  $n_Y$  is reachable from  $n_L$  through the directed edges. The sharing between L and K is represented by the fact that  $n_Z$  is reachable from both  $n_L$  and  $n_K$ .



Fig. 8: Modelling of sharing information.

# 5 Region Points-To Analysis

The region points-to analysis aims at computing for each procedure in a Mercury program a region points-to graph that represents the locations of its variables and the sharing among them.

The region points-to analysis is unification-based and 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. Both analyses make use of a unify operation, which is defined in Algorithm 1, to capture sharing by unifying nodes in a region points-to graph. To ensure that there is only one outedge 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.

Algorithm 1 unify(n,m)

```
Require: G(N, E), n, m \in N.
Ensure: G(N, E) 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 l, l' \in N do
          if (n, (g, j), l) \in E \land (n, (g, j), l') \in E \land l \neq l' then unify(l, l')
           end if
   end for
```

We will describe the analyses in turn with the assumption that we are analysing a procedure p.

Recall that, when describing the static region analysis and transformation, for convenience, we make the assumption that *all* terms are stored on the heap and therefore we need regions for them. In a concrete implementation, such as in [13], if certain terms do not need heap storage, their corresponding regions can just be ignored.

### 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 2.

As we know the type of each variable in p, we initialize  $G_p$  by using the TG graphs of the variables. In Algorithm 2, 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, and generating a fresh region variable for each node in the region points-to graph.

The intraprocedural analysis adds all the sharing created by the unifications in the procedure to  $G_p$ . For assignment, construction and deconstruction unifi-

**Algorithm 2** intraproc(p): intraprocedural analysis of a procedure p

cations we unify the nodes corresponding with the sharing created by them. We ignore test unifications because they do not create any sharing.

#### 5.2 Interprocedural Analysis

The interprocedural analysis, Algorithm 3, updates  $G_p$  by integrating the *rele*vant region-sharing information from the region points-to graphs of the called procedures into  $G_p$ .

```
Algorithm 3 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).
  repeat
         for
              all call site in p do
                Assume that the call is q(Y_1, \ldots, Y_n), with X_1, \ldots, X_n as corresponding formal arguments, and that G_q is available.
                % Build an \alpha relation.
                for k = 1 to n do
                       \alpha(n_{X_k}) = n_{Y_k}
                end for
                % Ensure \alpha is a function.
for all X_i, X_j do
                       if \alpha(n_{X_i}) = n_{Y_i} \wedge \alpha(n_{X_j}) = n_{Y_j} \wedge n_{X_i} = n_{X_j} \wedge n_{Y_i} \neq n_{Y_j} then
                              unify(n_{Y_i}, n_{Y_i})
                       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 9 when applicable.
          end for
  until There is neither change in G_p nor in any of the \alpha functions.
```

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 region-sharing among  $X_i$ 's in  $G_q$  may not have been present in  $G_p$  as region-sharing among  $Y_i$ 's. The interprocedural analysis makes sure that

this will be the case. Firstly, it builds the function  $\alpha : N_q \to N_p$  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 region-sharing. This is done by following the relevant edges in  $G_q$  to extend the  $\alpha$  function to all the relevant nodes in  $G_q$  (rule P2) and to unify the relevant nodes in  $G_p$  (rule P1).



Fig. 9: 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 calldependency graph into a tree of strongly connected components, and analysing the components in bottom-up order. Algorithm 4 illustrates this approach.

The points-to graphs of the split and qsort procedures in the *quicksort* program in Example 2 are shown in Figure 10. For split, the region points-to analysis can detect that the two sublists L1 and L2 can be in separate regions that are different from the region of the input list L. For qsort, the input list, the two temporary lists, and the resulting list are all in different regions. That the resulting list is in the same region as the accumulator and the temporary lists S2 and A1 is reasonable because the resulting list is gradually built up from them.



Fig. 10: The region points-to graphs of split and qsort.

#### 5.3 Correctness of the Region Points-To Graphs

We will prove that the region points-to analysis of a program terminates and that the resulting region points-to graphs for the procedures in the program are *correct*, i.e., they represent all the locations of the terms and the sharing among the terms.

#### **Theorem 1.** The region points-to analysis of a program terminates.

*Proof (Termination).* An  $\alpha$  function at a call site is a mapping from a subset of the nodes in the callee's region points-to graph to a subset of the nodes in the caller's region points-to graph. Therefore if we can show that the sets of nodes are finite then so is the  $\alpha$  function. This then implies that the termination of the region points-to analysis solely depends on the finiteness of the region points-to graphs.

For any procedure, the Algorithm 2 starts with a region points-to graph having a finite number of nodes and edges. The analysis uses only the *unify* operation (Algorithm 1) to change the graphs. It always decreases the number of nodes and does not increase the number of edges. Therefore the analysis must, at some point, terminate. In the most extreme cases, the final region points-to graph of a procedure contains only one node and self-edges, if any.

**Theorem 2.** The resulting region points-to graphs of the region points-to analysis of a program represent all the locations of the terms that are possibly constructed during the execution of the program and the possible sharing among the terms.

The theorem has two parts, one about locations and the other about sharing. We prove each part separately.

*Proof (Locations).* During the execution of a program a variable can get bound to a compound term. However that compound term must be built step-by-step using construction unifications. In such a step, a construction unification allocates memory to store only the principal functor that the variable on its left-hand side is bound to. Therefore to show that terms have their locations it should be enough to show that all variables have their locations. We will prove that the region points-to analysis ensures that every variable is assigned a location.

Consider a procedure. The region points-to analysis of the procedure starts with the intraprocedural analysis (Algorithm 2) that assigns a set of nodes to each variable based on the type-based region graph of the type of the variable. These nodes represent the regions where a term to which the variable is possibly bound is stored. Moreover, the variable is assigned a location by the fact that it is added to the *vars* set of the node where the principal functor of the term it is bound to is stored. During the analysis this node may be removed from the graph when it is unified with another node. However, regardless of where this happens, in the intraprocedural or in the interprocedural analysis, the *unify* operation ensures that the remaining node now represents the location of the variable.

Now, for the second part of Theorem 2, we will show that all sharing between the terms is represented in the region points-to graphs. For a procedure, the sharing among its variables is created either by the explicit unifications in the procedure or by the ones hidden in procedure calls. The lemma below deals with explicit unifications.

Lemma 1 (Sharing created by explicit unifications). If a unification explicitly appears in a procedure, the sharing created by the unification is represented in the region points-to graph of the procedure.

*Proof.* The explicit unifications are dealt with by Algorithm 2, the intraprocedural analysis. Test unifications do not create sharing, so we can ignore them. Consider an assignment unification, Algorithm 2 unifies the nodes of its left and right variables and keeps these two variables in the *vars* set of the unified node. This represents their sharing.

The other case is with a unification of this form  $X = f(\ldots, X_i, \ldots)$ . If we assume  $node(n_X, (f, i)) = m$ , Algorithm 2 does  $unify(m, n_{X_i})$  when handling the unification. This adds  $X_i$  to vars(m). After the unification, the edge  $(n_X, (f, i), m)$ , which was already in the region points-to graph, has become  $(n_X, (f, i), n_{X_i})$ . This represents the sharing between X and  $X_i$ .

For procedure calls, we consider a procedure p that invokes q,  $X_i$ 's are the formal arguments and  $Y_i$ 's are the actual ones. We call  $G_p^{sub}(N_p^{sub}, E_p^{sub})$  the subgraph of the region points-to graph of p rooted at the nodes of  $Y_i$ 's and  $G_q^{sub}(N_q^{sub}, E_q^{sub})$  the subgraph of the region points-to graph of q rooted at the nodes of  $X_i$ 's.

In order to prove that all the region-sharing in  $G_q^{sub}$  is also in  $G_p^{sub}$ , we first observe two arbitrary formal arguments  $X_i$  and  $X_j$  that share. By Definition 1, this means that there exists a node in  $G_q^{sub}$  that can be reached from  $n_{X_i}$  and  $n_{X_j}$ . There are two cases, either  $n_{X_i} = n_{X_j}$ , i.e., the sharing between  $X_i$  and  $X_j$  is represented in  $G_q^{sub}$  by the fact that they are in the *vars* set of the same node, or  $n_{X_i} \neq n_{X_j}$ , i.e., the sharing among them is represented by intermediate edges to the common node.

The following lemma shows that the region-sharing in the first case is brought to  $G_p^{sub}$ .

**Lemma 2.** The region-sharing between the formal arguments that are in the vars set of a node  $n_q \in N_q^{sub}$  is also in  $G_p^{sub}$ .

Proof. The interprocedural analysis (Algorithm 3) first builds an  $\alpha$  relation that represents the connection between  $G_q^{sub}$  and  $G_p^{sub}$ . This initial  $\alpha$  relation connects the nodes of the formal arguments with the nodes of the corresponding actual arguments. In this  $\alpha$  relation, it is possible that a node in  $G_q^{sub}$  whose vars set contains more than one formal argument is connected to more than one node of the actual arguments in  $G_p^{sub}$ . The region-sharing of such formal arguments (represented by the fact that they are in the same vars set) is brought into  $G_p^{sub}$  when Algorithm 3 turns the  $\alpha$  relation into a function. So the actual arguments corresponding to the formal arguments that are in the vars set of a node  $n_q$  in  $G_q^{sub}$  are in the vars set of a node  $n_p$  in  $G_p^{sub}$  and  $\alpha(n_q) = n_p$ .

For the second case, we first introduce the following lemma.

**Lemma 3.** If n and m are in  $N_q^{sub}$  such that  $(n, (f, i), m) \in E_q^{sub}$  and  $\alpha(n) \in N_p^{sub}$  then  $\alpha(m) \in N_p^{sub}$  and also  $(\alpha(n), (f, i), \alpha(m)) \in E_p^{sub}$ .

*Proof.* In a well-typed Mercury program, an actual argument has the same type as its corresponding formal one. Therefore if (n, (f, i), m) is in  $E_q^{sub}$  there must exist a node  $k \in N_p^{sub}$  such that  $(\alpha(n), (f, i), k)$  is in  $E_p^{sub}$ . If  $\alpha(m) = k$ , our proof is done. If  $\alpha(m) = m' \neq k$ , the algorithm applies rule P1 to unify k and m'. After that we have the desirable effect  $\alpha(m) = k$ . If  $\alpha(m)$  is undefined, the algorithm applies rule P2 to produce  $\alpha(m) = k$ , which is also the desirable result.  $\Box$ 

Lemma 3 essentially shows that the  $\alpha$  function is extended for all the nodes in  $N_q^{sub}$  and all the intermediate edges connecting these nodes in  $E_q^{sub}$  have their counterparts in  $G_p^{sub}$ .

**Theorem 3 (Sharing created by procedure calls).** All the region-sharing in  $G_a^{sub}$  is also in  $G_n^{sub}$ .

*Proof (Sharing created by procedure calls).* The proof of Theorem 3 follows from Lemma 2 and Lemma 3.  $\hfill \Box$ 

Now we can continue with the proof of the sharing-among-terms part of Theorem 2.

*Proof (Sharing among terms).* The proof of the second part of Theorem 2 follows from Lemma 1 and Theorem 3, which show that the sharing created by explicit unifications as well as by procedure calls in a procedure is all represented in the region points-to graph of the procedure.

When a procedure is recursive or mutually recursive, it is possible that the region points-to graph of a called procedure (recursive or mutually recursive) has not fully represented the sharing among its formal arguments. However, if a program ever creates sharing, ultimately this creation must involve a unification. And we already showed by Lemma 1 that this sharing is represented in the region

points-to graph of the procedure containing the unification. By Theorem 3, we showed that that sharing will also be represented in the region points-to graphs of any procedures that invoke the procedure. Therefore even the procedure is recursive, the sharing created by the recursive calls will finally be represented in its region points-to graph.  $\hfill\square$ 

In the next sections when we mention region points-to graphs we mean the ones obtained by the region points-to analysis.

#### 5.4 Regions That Are Allocated Into in A Procedure

During the region points-to analysis of a procedure we can track the regions that are *possibly* allocated into in the procedure. A construction unification is the only construct in Mercury that allocates memory. When processing a construction unification  $X \ll f(...)$  we mark the node  $n_X$  as *allocated*. When two nodes are unified, if one node is marked as allocated then the unified node is also marked as allocated. At a call site, if an actual node is mapped to by an allocated node, it is marked as allocated. For a procedure p, we call this set of nodes *allocation(p)*. In the *quicksort* example of Figure 3 and Figure 10, *allocation(split)* = {R3, R4}, *allocation(qsort)* = {R8, R9, R10},

### 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 pass region variables as arguments of procedures. The set of region arguments is a subset of the set of region variables derived by the region points-to analysis.

A region variable being live means that it should be bound to a region and is possibly used in future (forward) execution. The goal of the region liveness analysis is 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.

Consider a procedure p. We associate a program point with every literal in the body of 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 denote an execution path by  $\langle l_1, \ldots, l_n \rangle$ , in which the  $l_i$ 's are the literals involved, the indexes *i*'s reflect the order among the literals in this execution path (not their associated program points). The function pp(l) returns the program point associated to *l*. 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. The set of live region variables at a program point is computed via the set of live variables at the program point. We also use the  $in\_args(atom)$  and  $out\_args(atom)$  functions that respectively return the sets of input and output

	$in\_args$	out_args
$X \le f(X_1, \dots, X_n)$	$\{X_1,\ldots,X_n\}$	$\{X\}$
$X \Longrightarrow f(X_1, \dots, X_n)$	$\{X\}$	$\{X_1, \ldots, X_n\}$
X == Y	$\{X, Y\}$	Ø
X := Y	$\{Y\}$	$\{X\}$

Table 1: Input and output arguments of unifications.

arguments to *atom*. For specialized unifications they are defined in Table 1. If *atom* is a procedure's head, they return formal arguments, whereas if *atom* is a call they return actual ones. Those sets can be computed from the mode information of Mercury procedures.

#### 6.1 Live Region Variables at a Program Point

In this subsection we specify the analysis that computes the sets of region variables that are live before and after every program point in a procedure. The liveness of a region variable at a program point is determined by the liveness of the variables that are stored in the corresponding region.

**Live variables.** A variable is live *before* a program point if it has been instantiated before the point and may be used in the future. A 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 for a procedure p is defined in Algorithm 5. It is a

#### **Algorithm 5** lva(p): live variable analysis of a procedure p.

```
Require: p in superhomogeneous form.
Ensure: The sets of live variables before (LV_{before}) and after (LV_{after}) all program points in p.
  for all program point i in p do
         LV_{before}(i) = LV_{after}(i) = \emptyset
  end for
  for all ep \equiv \langle l_1, \ldots, l_n \rangle in p do
         for j = n downto 1 do
                i = pp(l_j)
if j = n then
                       LV_{after}(i) = out\_args(p)
                \mathbf{else}
                       LV_{after}(i) = LV_{after}(i) \cup LV_{before}(pp(l_{j+1}))
                end if
                if j = 1 then
                       LV_{before}(i) = in_{args}(p)
                else
                       LV_{before}(i) = (LV_{after}(i) \setminus out\_args(l_i)) \cup in\_args(l_i)
                end if
         end for
  end for
```

backward computation of live variables, in which we follow each execution path (ep) backwards from its last program point. At each program point we apply the suitable formula to update the  $LV_{after}$  and  $LV_{before}$  sets of the program point.

The  $LV_{before}$  of the first program point(s) is defined to be  $in_{args}(p)$  while the  $LV_{after}$  of the last program point(s) is defined to be  $out_{args}(p)$ .

**Live region variables.** 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:

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

Then the live region variable analysis of a procedure is specified in Algorithm 6.

#### **Algorithm 6** lra(p): live region variable analysis of a procedure p

```
Require: LV_{before} and LV_{after} of all program points in p.

Ensure: The sets of live region variables before (LR_{before}) and after (LR_{after}) all program points in p.

for all program point i in p do

LR_{before}(i) = LR_{after}(i) = \emptyset

for all X \in LV_{before}(i) do

LR_{before}(i) = LR_{before}(i) \cup Reach(X)

end for

for all X \in LV_{after}(i) do

LR_{after}(i) = LR_{after}(i) \cup Reach(X)

end for

end for

end for
```

#### 6.2 Lifetime of Regions across Procedure Boundary

As said before we use region arguments to pass regions among procedure calls. For a procedure, the region variables reachable from its arguments are candidates for region arguments. But as we will see later not all of them may need to be. This subsection introduces an analysis that decides which region variables become live or cease to be live inside a procedure. With this information we can give shorter lifetime for regions, achieving better memory reuse.

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  $\alpha$  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.
- outlived R(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.

20

The motivation is that, in the transformed 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. An alternative approach would be that p create those mapped to by ones in bornR(q) just before and remove those mapped to by ones in deadR(q) right after the call to q. In this approach, the corresponding regions have longer lifetime.

For 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 the variables in  $in\_args(p)$  and  $out\_args(p)$ , respectively. This is an overestimate in which all the region variables that contain input terms but are not involved with output terms are assumed to become dead in p, while all the region variables where output terms are stored but are not yet bound at the entry of p are assumed to become live in p. We call the set of the region variables that are local to p (not reachable from input or output variables), localR(p), which is computed by  $N_p \setminus (inputR(p) \cup outputR(p))$ . Initially,  $outlivedR(p) = inputR(p) \cap outputR(p)$ . It is clear that the  $N_p$  is composed of four disjoint sets, localR(p), bornR(p), deadR(p), and outlivedR(p).

The calling contexts of a procedure influence what it can do to its non-local region variables. Therefore when analysing a procedure p, the analysis applies the rules in Figure 11 to any call q in p to correct the *deadR* and *bornR* sets of q according to the calling context. A region variable is moved from deadR(q) to outlivedR(q) if it needs to be live after the call to q in p (i.e., the region it is bound to before the call must not be reclaimed during the call) (rule L1); or to avoid wrong removals due to the so-called "region alias", which generally means that a region is referred to by more than one region variable (rule L2). 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  $R_{Y_1} \equiv R_{Y_2}$ .

A region variable is moved from bornR(q) to outlivedR(q) if it is already live before the call to q (rule L3); or to avoid wrong creations, again 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. After a fixpoint is reached, each procedure has exactly one *bornR* set and one *deadR* set that are 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.

In the quicksort program in Example 2, 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 paths:  $\langle (1), (2) \rangle$  and  $\langle (3), (4), (5), (6), (7) \rangle$ . <sup>1</sup> The LV and LR sets of split are in Table 2(a), of qsort in Table 2(b) (see also Figure 3 and Figure 10). Note that, in this example, it happens to be 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

<sup>&</sup>lt;sup>1</sup> For convenience, we use program points to describe execution paths.

$\frac{r \in LR_{before}(pp(l))}{r \in LR_{after}(pp(l))}$ $\frac{r = \alpha(r')  r' \in deadR(q)}{deadR(q) = deadR(q) \setminus \{r'\}}$ $outlivedR(q) = outlivedR(q) \cup \{r'\}$	(L1)	$\frac{\alpha(r') = r}{r' \neq r''}  \begin{array}{c} \alpha(r'') = r \\ r' \in deadR(q) \\ \hline \\ \hline \\ \hline \\ \hline \\ \hline \\ \\ \hline \\ \\ \\ \\ \\ \\ $	(L2)
$\frac{r \in LR_{before}(pp(l))}{r = \alpha(r')  r' \in bornR(q)}$ $\frac{r' \in bornR(q)}{bornR(q) = bornR(q) \setminus \{r'\}}$ $outlivedR(q) = outlivedR(q) \cup \{r'\}$	(L3)	$\frac{\begin{array}{ccc} \alpha(r') = r & \alpha(r'') = r \\ r' \neq r'' & r' \in bornR(q) \\ \hline bornR(q) = bornR(q) \setminus \{r'\} \\ outlivedR(q) = outlivedR(q) \cup \{r'\} \end{array}$	(L4)

l is the literal at a program point, which is a call to q(...).

Fig. 11: Region liveness analysis rules.

pp	LV	LR	]			
$(1_b)$	$\{X, L\}$	$\{R5, R1, R2\}$	] pi	,	LV	LR
$\begin{array}{c} (1_a, 2_b) \\ (2_a, 3_b) \\ (3_a) \\ (4_b) \\ (4_a, 5_b) \\ (5_a, 6_b) \\ (6_a, 7_b) \\ (7_a) \\ (5 8_1) \end{array}$	$\{ \{ L1 \} \\ \{ L1, L2 \} \\ \{ X, L \} \\ \{ X, Le, Ls \} \\ \{ X, Le, Ls \} \\ \{ L2, Le, L11 \} \\ \{ L1, L2 \} \\ \{ X, Le, Ls \} $	$\{\}$ $\{R3, R2, R4\}$ $\{R5, R1, R2\}$ $\{R5, R2, R1\}$ $\{R5, R2, R1\}$ $\{R4, R2, R3\}$ $\{R3, R2, R4\}$ $\{R5, R2, R1\}$	$\begin{array}{c} & & \\ \hline & (1_{a}) \\ (1_{a}, \\ (2_{a}) \\ (3_{b}) \\ (3_{a}, \\ (4_{a}, \\ (5_{a}, \\ (6_{a}, \\ \end{array}) \\ \end{array}$	$2_b)$ $2_b)$	$ \begin{array}{c} L, A \\ \{L, A\} \\ \{A\} \\ \{S\} \\ \{L, A\} \\ \{A, Le, Ls\} \\ \{A, Le, L1, L2\} \\ \{Le, L1, S2\} \\ \{L1, A1\} \end{array} $	
$(8_a, 9_b)$	$\{L1, Le, L21\}$	$\{R3, R2, R4\}$	(7 a	)	$\{S\}$	$\{R8, R7\}$
$(9_a)$	$\{L1, L2\}$	$\{R3, R2, R4\}$	J		(b) <i>qs</i>	ort
	(a) <i>spli</i>	t				

Table 2: Live variable and live region variable sets in *quicksort* program.

not necessarily the case, for example consider a split point before a disjunction or before an if-then-else, the set of live region variables after it is the union of all the sets before its next points. When computing deadR and bornR sets, R5is eliminated from deadR(split) due to the application of the rule L1 to the call to split inside qsort. The final result is as in Table 3.

	localR	bornR	deadR	outlived R
split	Ø	$\{R3, R4\}$	$\{R1\}$	$\{R2, R5\}$
qsort	$\{R9, R10\}$	Ø	$\{R6\}$	$\{R7, R8\}$
111 (	<u> </u>	C + 1 +	<u>с</u> .	• 11

Table 3: Division of the set of region variables.

### 6.3 Correctness

Algorithm 6 that detects live region variables locally at each program point is an extension of live variable analysis, which is a standard, well-known program analysis [10]. Theorem 2 guarantees the locations of variables and their possible

22

sharing, which are represented in terms of region points-to graphs. Therefore Algorithm 6 computes all the live region variables by starting from the live variables and collecting all the reachable region variables using the region pointsto graphs.

The analysis in Section 6.2 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 finite, and the analysis just reduces their size. The rules in Figure 11 enforce all the cases where a caller of a procedure needs to restrict what the callee can do to its region variables. The eager application of the rules therefore ensures that, for a procedure, the *bornR* and *deadR* sets obtained after a fixpoint is reached contain exactly the region variables it will safely create and remove, respectively.

# 7 Program Transformation

The purpose of the program transformation is to annotate all the procedures in a program with sufficient information about regions. For a procedure the transformation needs to accomplish the following tasks.

- Extend the procedure definition with the formal region arguments.
- Annotate its construction unifications with the region variables of their lefthand side variables.
- Annotate its procedure calls with actual region arguments.
- Insert *create* and *remove* instructions at suitable points.

The second task is straightforward because the information about which program variables are in which regions is available after the region points-to analysis. The aim is that the memory needed by a construction unification will be allocated in the region that is bound to by the region variable of its left-hand size variable.

We elaborate the other tasks in the next two subsections.

## 7.1 Region Arguments

We need to make the region variables in *bornR* and *deadR* arguments because they are going to be created and removed inside the procedure. Other than these region variables, we also need to pass as arguments the region variables that are reachable from the input and output variables *and* are allocated into in the procedure, called *allocR*. This set of region variables can be computed by  $allocR = (inputR \cup outputR) \cap allocation$  (Section 5.4). Note that *allocR* is not necessarily disjoint with *bornR*, *deadR* and *outlivedR*.

So all in all, the set of formal region arguments of a procedure is  $deadR \cup bornR \cup allocR$ . In the quicksort program,  $allocR(split) = \{R1, R2, R3, R4\} \cap \{R3, R4\} = \{R3, R4\}$ ,  $allocR(qsort) = \{R6, R8\} \cap \{R8\} = \{R8\}$ , and the region arguments are  $\{R1\} \cup \{R3, R4\} \cup \{R3, R4\} = \{R1, R3, R4\}$  and  $\{R6\} \cup \emptyset \cup \{R8\} = \{R6, R8\}$ , respectively.

The actual region arguments to a procedure call are derived from the formal region arguments of the called procedure and the  $\alpha$  function at the call site.

#### 7.2 Insertion of *create* and *remove* instructions

When manipulating regions our intention is that a region is created and removed only by the *create* and *remove* instructions, respectively. When a region is created, the region variable in the *create* instruction is bound to it. To remove a region, we call *remove* on the region variable that is bound to the region. We can consider *create* and *remove* special Mercury procedures. Other than that, regions can be indirectly created and removed in procedure calls, which invoke the two instructions. Unifications neither create nor remove regions.

The insertion of the instructions is specified by Algorithm 7 in which the transformation rules in Figure 12 are applied to the literal at each program point. We use the function literal(i) to refer to the literal at the program point i.

```
Algorithm 7 Insertion of region instructions in a procedure p.
Require: p in superhomogeneous form; the analysis information is ready.
  for all program point i in p\ \mathbf{do}
        l = literal(i)
         apply rule T6 to l
if l \equiv unif then
                apply rule T4 to l
                if l \equiv X \le f(\dots) then
               apply rule T2 to l
end if
         else
                apply rules T1 and T3 to l
         end if
  end for
  for all ep \equiv \langle l_1, \dots, l_n \rangle in p do
for j = 1 to n - 1 do
                apply rule T5 to l_j, l' \equiv l_{j+1}
         end for
  end for
```

$ \begin{split} &l \equiv q(\ldots) \\ &r \in LR_{after}(pp(l)) \setminus LR_{before}(pp(l)) \\ &r \in localR(p) \cup bornR(p) \cup deadR(p) \\ &\frac{r = \alpha(r') \to r' \not\in bornR(q)}{add \ ``create(r)'' \ before \ l} \end{split} $	(T1)	$\frac{l \equiv X <= f(\ldots)}{\substack{r \in LR_{after}(pp(l)) \setminus LR_{before}(pp(l)) \\ r \in localR(p) \cup bornR(p) \cup deadR(p)}}$ add "create(r)" before l	(T2)
$\begin{split} l &\equiv q(\ldots) \\ r &\in LR_{before}(pp(l)) \setminus LR_{after}(pp(l)) \\ r &\in localR(p) \cup deadR(p) \cup bornR(p) \\ \hline \frac{r &= \alpha(r') \to r' \not\in deadR(q)}{add \ ``remove(r)'' \ after \ l} \end{split}$	(T3)	$ \begin{split} l &\equiv \textit{unif} \\ r \in \textit{LR}_{before}(pp(l)) \setminus \textit{LR}_{after}(pp(l)) \\ r \in \textit{localR}(p) \cup \textit{deadR}(p) \cup \textit{bornR}(p) \\ \hline & add "remove(r)" ~after~l \end{split} $	(T4)
$\frac{l' \text{ is next to } l \text{ in an execution path}}{r \in LR_{after}(pp(l)) \setminus LR_{before}(pp(l'))} \\ \frac{r \in localR(p) \cup deadR(p) \cup bornR(p)}{add \text{ "remove}(r) \text{"before } l'}$	(T5)	$\frac{r \in VR(pp(l)) \setminus LR_{after}(pp(l))}{r \in localR(p) \cup deadR(p) \cup bornR(p)}$ add "remove(r)" after l	(T6)

Fig. 12: Transformation rules.

#### 24

Each program point is associated with three sets of region instructions: a set of remove instructions added before the program point, a set of create instructions added before it, and a set of remove instructions added after it. We assume that the region instructions in the first set are executed before the ones in the second set. The reason for this will become clear in Section 7.3

Consider a literal l at a program point i in a procedure p. 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, 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. Only for the latter case we need to add a *create* instruction before l and this occurs when the region variable belongs to either bornR(p), localR(p) or deadR(p). This is reflected by the transformation rules T1 and T2. While the reason for creating a region to which a region variable that belongs to bornR(p) or localR(p) is bound is straightforward, the creation of a region that is bound to by a region variable R in deadR(p) is needed because it is acceptable for p to remove the region bound to by R at some point before l, if that is safe, and re-create R right before l. The new region will be removed later due to the fact that R is in deadR(p).

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 pand 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* 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. The transformation rules T3 and T4 ensure this effect. Similar to the case with creation, the removal of a region that is bound to by a region variable in bornR(p) is necessary because it is acceptable for p to safely remove the region after i and re-create it later on. The fact that the region variable is in bornR(p) ensures this. The second case is when the region variable is live after i, but not live before the literal l' that is next to l 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 l' does not belong. A remove instruction is added before l' to remove the region as expressed by the transformation rule T5.

We illustrate the effects of the re-creation of regions by two procedures in Figure 13 and their region-annotated counterparts in Figure 14. We include the definition of the function length, which returns the number of elements of the input list, for completeness. It is of no importance to what we are illustrating. We also assume that there is no region for integers. Therefore the focus is only on the variables B and C in the procedure p and V and X in q, which are of the type list\_int (see Example 1). Each pair of them is assigned to the same region variables, R1 in p and R2 in q due to the respective assignments at the program points (3). p and q are unrelated and used to demonstrate different situations.

```
% p(in, out).
                           % q(in, out).
                           q(X, Y) :-
(1) Z := length(X),
p(A, B) :-
                                                          length(L) = N :-
                                                             L == [], N := 0
L => [_ | T],
(1) C <= [1],
                                                          (
     ( if
                                ( if
                                                          ;
                                    Z == 1
(2)
         Α
           == 1
                           (2)
                                                              N := length(T) + 1
       then
                                   then
                                                          ).
(3)
         B := C
                           (3)
                                    V := X
       else
                                  else
         B <= [2]
(4)
                           (4)
                                    V <= [1]
    ).
                                )
                           (5) Y := Z + length(V).
```

Fig. 13: Effect of re-creation of regions.

```
p(A, B@R1) :-
                              q(X@R2, Y) :-
    create(R1)
                              (1) Z := length(X),
(1) C <= [1] in R1,
                                  ( if
    ( if
                              (2)
                                      Z == 1
(2)
        A == 1
                                    then
                              (3)
                                      V := X
      then
(3)
        B := C
                                     else
      else
                                      remove(R2),
        remove(R1),
                                       create(R2),
        create(R1),
                              (4)
                                      V <= [1] in R2
(4)
        B <= [2] in R1
                                  ).
                              (5) Y := Z + length(V),
    ).
                                  remove(R2).
```

Fig. 14: Effect of re-creation of regions: region-annotated version.

Assume that p can create R1, i.e., no calling context forces it otherwise. So R1 is in bornR(p). In Figure 14, the create instructions added for it before (1) and (4) are due to the rule T2. The remove instruction added before (4) is due to the rule T5. So if the else branch is reached, R1 that was live after (1) is no longer live before (4) and we can reclaim the memory occupied by the list [1] by removing R1 before re-creating R1 and allocating the list [2].

For q, assume that R2 is in deadR(q). Before the program point (4) R2 is not live, the **remove** instruction is added due to the rule T5. As R2 is live after (4) the **create** instruction is added before (4) due to the rule T2. The **remove** instruction after (5) is added due to the rule T4. So if the else branch is reached, we can reclaim the memory of the input list X by removing R2 before recreating it to construct V.

Handling of instantly-dead variables. In a program, we may have variables that are instantiated at some point but never used after that. We call them instantly-dead variables. In logic programming in general and in Mercury in particular, they can be void or singleton variables. A void variable's name generally starts with the underscore (\_, e.g., see Figure 2) to explicitly tell the compiler that we do not care about its value. A singleton variable is often a mistake of the programmer. Mercury compiler issues a warning when it detects

a singleton variable and to avoid the warning the correct action is to turn it into a void variable. Because it is useless to do a construction to an instantly-dead variable, i.e., when the left-hand side variable of a construction unification is instantly-dead, we assume that construction unifications whose left-hand side variables are instantly-dead have been eliminated before our region analysis and transformation. (In fact, the Mercury compiler does apply this kind of optimization.)

For the purpose of this work, we care only about the fact that they get instantiated and not used in the future. Being instantiated means that we need regions to store their terms, and we do want to remove the regions. That such variables are not used in the future makes them not live and we may not rely on the their liveness to remove related regions. That is why we have the rule T6 to deal with this case. We assume that at each program point *i* we have the set of such instantly-dead variables, VV(i). (*i* is the point they get instantiated.) We then compute VR(i), the set of region variables that are reachable from the variables, by  $\bigcup_{V \in VV(i)} Reach(V)$ . The basic idea of T6 is to remove a region variable reachable from an instantly-dead variable right after the point where the variable gets instantiated if the region variable is not reachable from any of the live variables after the point.

The result of the program transformation of the *quicksort* program in Example 2 has been shown in Figure 4. 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.

#### 7.3 Correctness of Region-Annotated Programs

In region-annotated programs, the computational behaviour of the original programs is not changed. Only the memory locations of terms are different. We therefore restrict the correctness of region-annotated programs to the correctness of memory access, i.e., the safety of read and write accesses to terms. Before arguing about this safety we prove a theorem about the bindings of live region variables.

**Theorem 4.** Consider a procedure p in a program P. We call P' the regionannotated program that is produced by applying the analyses and transformation in Sections 5, 6, and 7 to P, in which p' is the region-annotated version of p. If a region variable is live before (after) a program point i in p', then in p' it is bound to a region before (after) i.

To prove Theorem 4, we formulate several propositions.

**Proposition 1.** If program point *i* is right before program point *j* in some execution path of a procedure then  $LV_{before}(j) \subseteq LV_{after}(i)$  (\*) and  $LR_{before}(j) \subseteq LR_{after}(i)$  (\*\*).

*Proof.* (\*) follows immediately from Algorithm 5. (\*\*) follows from (\*) and Algorithm 6.  $\Box$ 

**Proposition 2.** When the literal at program point *i* is a unification, we have the following two properties. If it is a construction unification then  $LR_{before}(i) \subseteq LR_{after}(i)$ . If  $LR_{before}(i) \subset LR_{after}(i)$  (strict subset) then the literal is a construction unification.

Proof. Consider a construction unification of the form  $X \leq f(X_1, \ldots, X_n)$ . By definition (Algorithm 5)  $LV_{before}(i) = LV_{after}(i) \setminus \{X\} \cup \{X_1, \ldots, X_n\}$ . So we can compute  $LR_{before}(i) = (\bigcup Reach(V) \forall V \in LV_{after}(i) \land V \neq X) \cup (\bigcup Reach(X_i) \ i = 1..n)$ . We can also write  $LR_{after}(i) = (\bigcup Reach(V) \ \forall V \in LV_{after}(i) \land V \neq X) \cup Reach(X)$ . Algorithm 2 ensures that the edges from  $n_X$  to  $n_{X_i}$  are in the region points-to graph, therefore  $Reach(X) \supseteq (\bigcup Reach(X_i) \ i = 1..n)$ . So  $LR_{before}(i) \subseteq LR_{after}(i)$ .

To prove the second property we will show that if the unification is not a construction unification then  $LR_{before}(i) \supseteq LR_{after}(i)$ .

Consider an assignment unification of the form X := Y. From Algorithm 2 we have that X and Y are in the same node in the region points-to graph, therefore Reach(X) = Reach(Y). By definition  $LV_{before}(i) = (LV_{after}(i) \setminus \{X\}) \cup \{Y\}$ , then  $LR_{before}(i) = (\bigcup Reach(V) \forall V \in LV_{after}(i) \land V \neq X) \cup Reach(Y)$ . We can write  $LR_{after}(i) = (\bigcup Reach(V) \forall V \in LV_{after}(i) \land V \neq X) \cup Reach(X)$  and therefore  $LR_{before}(i) = LR_{after}(i)$ .

Consider a test unification of the form X == Y. In this case  $LV_{before}(i) = LV_{after}(i) \cup \{X, Y\}$  then it is trivial that  $LR_{before}(i) \supseteq LR_{after}(i)$ .

Consider a deconstruction unification of the form  $X => f(X_1, \ldots, X_n)$ . Here  $LV_{before}(i) = (LV_{after}(i) \setminus \{X_1, \ldots, X_n\}) \cup \{X\}$ , and we have  $LR_{before}(i) = (\bigcup Reach(V) \forall V \in LV_{after}(i) \setminus \{X_1, \ldots, X_n\}) \cup Reach(X)$ . We can write  $LR_{after}(i) = (\bigcup Reach(V) \forall V \in LV_{after}(i) \setminus \{X_1, \ldots, X_n\}) \cup (\bigcup Reach(X_i) \ i = 1..n)$ . We have shown that  $Reach(X) \supseteq (\bigcup Reach(X_i) \ i = 1..n)$ . Therefore  $LR_{before}(i) \supseteq LR_{after}(i)$ .

**Proposition 3.** If the literal at program point *i* is a unification and there exists a region variable *R* such that  $R \notin LR_{before}(i)$  and  $R \in LR_{after}(i)$ , then  $LR_{before}(i) \subset LR_{after}(i)$  (strict subset).

*Proof.* The existence of a region variable R such that  $R \notin LR_{before}(i)$  and  $R \in LR_{after}(i)$  means that the literal can neither be an assignment, a test, or a deconstruction unification because if it would  $LR_{before}(i)$  would have been a superset of or equal to  $LR_{after}(i)$  (shown in the proof of Proposition 2).

If the unification is a construction then  $LR_{before}(i) \subseteq LR_{after}(i)$  (Proposition 2). Then that there exists an R such that  $R \notin LR_{before}(i)$  and  $R \in LR_{after}(i)$  means that  $LR_{before}(i) \subset LR_{after}(i)$ .

Now we can give the proof for Theorem 4.

*Proof.* Hypothesis: Assume that Theorem 4 is true globally at all the points that are reached before the (local) program point i in p in an execution of the program.

Consider a region variable R.

If R belongs to outlivedR(p), according to the Hypothesis it is bound to a region at the entry to p. Because no create(R) or remove(R) is added in p and none of the procedures called by p creates or removes R, it is bound to the same region at all points in p, certainly including the points where it is live.

Consider the other case in which R belongs to either of localR, bornR, or deadR.

- 1. Consider a region variable R that is live before i, i.e.,  $R \in LR_{before}(i)$ .
  - When i is the first program point, R must be reachable from a variable in  $in\_args(p)$  (Algorithms 5 and 6). In the context of a caller of p, the region variable of the caller that R is mapped to is live before the call. By the Hypothesis we have that it is bound to a region before the call and therefore R is bound to the region at the entry to p. The transformation rule T5, which adds a *remove* instruction before a program point, is not applicable to the first program point. Therefore no remove instruction is added before i, meaning that R is bound to a region before i.
  - If *i* is not the first program point then *R* is in  $LR_{after}(h)$  where *h* is the program point right before *i* in an execution path (Proposition 1). According to our hypothesis, *R* is bound to a region after *h*. Again, the rule T5 is not applicable because *R* is in both  $LR_{after}(h)$  and  $LR_{before}(i)$ , therefore *R* is bound before *i*.
- 2. Consider a region variable R that is live after i, i.e.,  $R \in LR_{after}(i)$ . Assume that l is the literal at i.
  - (a) Consider the case in which R is not in  $LR_{before}(i)$ .
    - If l is a unification, from Proposition 3 we have that  $LR_{before}(i) \subset LR_{after}(i)$  and then from Proposition 2 it must be a construction unification.

A create(R) instruction is added before l according to rule T1. This means that R is bound to a region before l. Recall that we assume that the set of *create* instructions are executed right before l after the execution of the set of *remove* instructions, if any. Therefore R is bound before l. In addition to that, l is a construction unification so it does not remove any regions, meaning that R is still bound to the region after l.

- Consider the case in which l is a procedure call to q. If R is mapped to by a region variable in bornR(q), the region variable is live after any last program point of q. By the Hypothesis we can say that the region variable is bound to a region at the exit of q. So R is bound to that region after the call.

Otherwise, create(R) is added before l by rule T1, which means that R is bound to a region before l (again no *remove* instruction is executed in between create(R) and l).

Because R is not live before the call, it is not reachable from any actual input arguments to the call to q. Therefore it is not mapped to by a region variable of q that belongs to deadR(q). So we have

that R is not mapped to by any region variables of q that are either in bornR(q), in deadR(q), or in localR(q), which contains only region variables local to q. This means that R is not removed in q.

In both subcases above, the rules T3, T4 and T6 will not be applicable because R is in  $LR_{after}(i)$ . Therefore no remove(R) is added after l. So we can conclude that R is bound to a region after l.

(b) Consider the case in which R is in  $LR_{before}(i)$ . We showed in 1. that R is bound to a region before i.

If l is a unification it does not remove R. If l is a call to q, because R is in both  $LR_{after}(i)$  and  $LR_{before}(i)$ , R is neither mapped to by a region variable in deadR(q) nor in bornR(q) (Rules L1 and L3). So l does not remove it.

Again, no remove(R) is added after l because R is in  $LR_{after}(i)$ . Therefore we conclude that R is bound to the same region after l.

**Theorem 5.** In region-annotated programs, an allocation of memory, i.e., a memory write access, is safe.

*Proof.* An allocation of memory involves a construction unification. From Theorem 2 we know the region variable containing the left-hand side variable of the construction unification, i.e., where the being-constructed functor is stored. We say that the construction is safe if that region variable is bound before the construction unification.

Consider the program point associated to the construction unification. With the assumption that the left-hand side variable is not instantly dead, it must be live after this point where it is instantiated. And therefore its region variable is also live after this point (Algorithm 6). By Theorem 4 the region variable is bound to a region after the program point. Because the construction unification does not create regions, the region must have been created before and is available at the construction.  $\Box$ 

**Theorem 6.** When a variable appears as an input argument to a literal at a program point, we say that the variable is read at that point. In region-annotated programs, when a variable is read at a program point the term it is bound to is available.

*Proof.* When a variable is read at a program point, the Mercury compiler ensures that it has been instantiated before that. From Theorem 2 we know the region variables where the terms that the variable is possibly bound to is stored. They are the region variables reachable from the variable.

Because the variable is read at that point we consider it a live variable before that point, and therefore the region variables reachable from it are also live before the point (Algorithms 5 and 6).

Consider a variable X that is read at a program point i in a procedure p. The fact that X is bound in p is either because it is an input argument of p or because it is the output argument of some literal in p. Consider some execution path of p. In the first case, X is live before the first program point of the path. Because

it is an input argument the Mercury compiler ensures that it never appears as the output argument of any literal in p. So according to Algorithm 5, we have that X is live in the scope from before the first program point up to before i. Similarly in the second case, we have that X is live in the scope from after the literal up to before i. This means that all the region variables reachable from Xare live during the same scope. Therefore none of them get removed during the scope because the rules T3, T4, T5, and T6 are not applicable and no procedure calls in the scope remove any of them due to the rule L1.

So the term that X is bound to is available at i and the read at i is safe.  $\Box$ 

# 8 Conclusion

In this paper we have presented a fully automated static region analysis and transformation for Mercury programs. We also argued for the correctness of the region-annotated programs with respect to forward execution by showing that memory access in the context of regions are safe. We intend to support backtracking at runtime. However integrating backward liveness into the region analysis is an alternative approach. The runtime support for regions and backtracking that allows us to execute the region-annotated programs will be discussed in a separate paper. A first attempt for such a system has been reported in [13].

#### References

- A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the* ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 174–185. ACM Press, 1995.
- L. Birkedal, M. Tofte, and M. Vejlstrup. From Region Inference to von Neumann Machines via Region Representation Inference. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, 1996.
- H. Boehm, and M. Weiser. Garbage collection in an uncooperative environment. Software - Practice and Experience, 18:807–820, 1988.
- 4. S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *Proceedings of the 4th International Symposium on Memory Management*, pages 85–96. ACM Press., Oct. 2004.
- 5. F. Henderson. Accurate garbage collection in an uncooperative environment. In *Proceedings of the 3rd International Symposium on Memory Management*, pages 150–156. ACM Press., 2002.
- 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.
- H. Makholm. A region-based memory manager for Prolog. In Proceedings of the 2nd International Symposium on Memory Management, pages 25–34. ACM Press., 2000.
- 8. H. Makholm. Region-based memory management in Prolog. Master's thesis, University of Copenhagen, 2000.

- 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.
- F. Nielson, H. R. Nielson, and C. Hankin. The Principles of Program Analysis. Springer, 1999.
- 11. Mercury language reference. http://www.cs.mu.oz.au/research/mercury/ information/doc-latest/mercury\_ref.
- 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.
- Q. Phan, Z. Somogyi, and G. Janssens. Runtime support for region-based memory management for Mercury. In *Proceedings of the 2008 International Symposium on Memory Management*, pages 61–70. ACM Press., 2008.
- 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.
- M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg. A retrospective on regionbased memory management. *Higher-Order and Symbolic Computation*, 17:245–265, 2004.
- M. Tofte and J.-P. Talpin. Region-based memory management. Information and Computation., 132(2):109–176, Feb. 1997.

32