

**A proposal for region-based memory
management for deterministic Mercury
programs**

*Quan Phan
Gerda Janssens*

Report CW 424, September 2005



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

A proposal for region-based memory management for deterministic Mercury programs

Quan Phan
Gerda Janssens

Report CW 424, September 2005

Department of Computer Science, K.U.Leuven

Abstract

This paper presents an approach for region-based memory management for Mercury programs. First, region analysis based on a points-to graph determines the different regions in the program. Second, the liveness of the regions is computed. Finally, a program transformation adds region annotations to the program for the region-based memory management. Some small Mercury programs are analysed manually and discussed to show the promising benefits of our approach.

While the approach is developed here mainly for deterministic Mercury programs the paper also discusses possible extensions to support non-determinism, module-based region analysis, and most noticeably the combination with compile-time garbage collection in the context of Mercury programs.

Keywords : Program Analysis, Mercury, Region-based memory management.

CR Subject Classification : D.3.4, I.2.3

A proposal for region-based memory management for deterministic 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

September 16, 2005

Abstract

This paper presents an approach for region-based memory management for Mercury programs. First, region analysis based on a points-to graph determines the different regions in the program. Second, the liveness of the regions is computed. Finally, a program transformation adds region annotations to the program for the region-based memory management. Some small Mercury programs are analysed manually and discussed to show the promising benefits of our approach.

While the approach is developed here mainly for deterministic Mercury programs the paper also discusses possible extensions to support non-determinism, module-based region analysis, and most noticeably the combination with compile-time garbage collection in the context of Mercury programs.

1 Introduction

This report describes an algorithm that starts with a Mercury logic program and ends up producing an output program with region-based memory management (RBMM). The input Mercury program is in normal form, has been optimized to have specialized forms of unification, and has goals reordered so that input variables are ground before any operations. The algorithm as described here is for deterministic programs but it is designed with support for non-deterministic programs as well as for

combination with compile-time garbage collection (CTGC) in mind. The algorithm is composed of three phases. The first phase is a goal-independent analysis of each procedure. This analysis detects the region structure of memory used by a procedure and represents this information in terms of region points-to graph. The precision of splitting memory into different regions will have a large impact on the quality of the whole algorithm. The second phase uses the region points-to graph of each procedure to precisely detect the lifetime of regions. The lifetime information is composed of the set of live regions at each program point and the sets of regions that a procedure creates and removes. The third phase is the transformation of the input program to a program with region support. Based on the information about the lifetime of regions it inserts statements to create regions, remove regions, and rename regions. The ability of the transformation to create regions right before they need to be live and to remove regions right after they become dead can theoretically reduce significantly the memory consumption of programs.

The structure of the report is as follows. Section 2 presents some basic notions of our approach. Section 3 introduces the concept of region points-to graph and the points-to analysis. Section 4 presents the live region analysis. Section 5 is about the transformation that adds annotations to programs. Section 6 shows the detailed analyses of some small Mercury programs. Finally, section 7 presents possible extensions of the approach and discuss the benefits of combining region-based memory management and compile-time garbage collection.

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

2 The basic approach

In this section we give an overview of the working context, what we want to achieve and the reasons behind them.

2.1 Term representation

One basic point when we talk about memory management is how terms are represented in memory. This aspect depends on a specific implementation of a language. Therefore we introduce our view of term representation when the heap memory is organised in terms of regions in the context of Melbourne Mercury Compiler. The way a term stored in regions is controlled by two factors. The first factor is the type of the term, which defines its structure and the second is how a program is going to use it. We explain this by the following example. We define two types as follows:

```
:- type list(T) --- > []:[T]list(T).
:- type ex --- > foo(int).
```

Consider a variable L of type $\text{list}(\text{ex})$. There are three structured components in the type $\text{list}(\text{ex})$: the list backbone, the functor $\text{foo}/1$, and the integer value inside foo . The term bound to L (as any other value of the type) can be stored finitely in maximum three corresponding regions, one for the list backbone, one for the functor $\text{foo}/1$, and the other for the integer value. Assume $L = [\text{foo}(1), \text{foo}(2)]$ the memory representation of L using three regions is shown in Figure 1. By allocating a term in many different regions we can do the removal of regions right after some parts of the term die. Clearly the way we divided a term into different regions here is based on its structure. Now assume that the program never accesses the integer inside $\text{foo}/1$. For example it can be a program that takes out only the elements of L , without caring about their content. Then in that case we may store the term in only two regions, one for the list backbone and the other for the $\text{foo}/1$ functor and its argument. This is shown in Figure 2. Please do not get the impression of de-reference. The idea here is different, we can and want to do that because the program does not access the argument of $\text{foo}/1$ and this representation is more economical. So when it is the case, even if the argument of $\text{foo}/1$ is no longer of primitive type it is still stored in the same region of $\text{foo}/1$. To see why it is reasonable to do

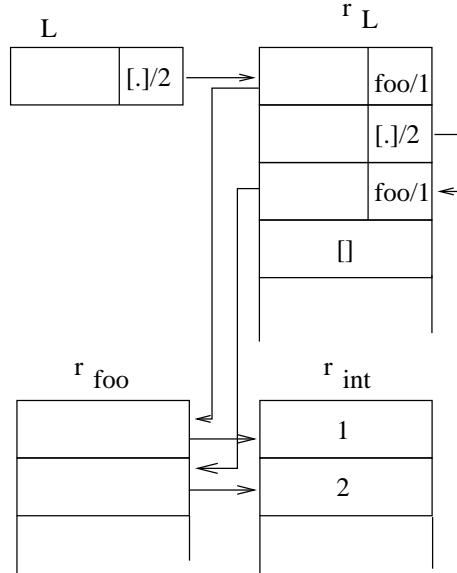


Figure 1: Representation of term using the maximum number of regions.

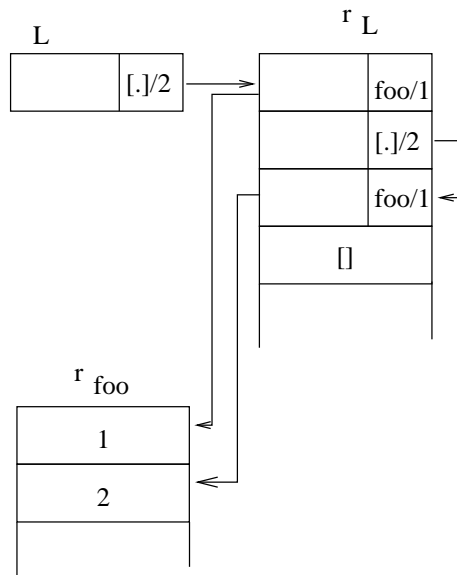


Figure 2: Representation of term using fewer regions.

like that imagine a procedure that receives an input variable of type `list(ex)`, returns an output variable of type `ex`. Certainly, it never touches the integer in `ex`. For this procedure the input variable can be stored in only two regions. In the calling context where the input variable is constructed, there are two situations. The caller constructs the variable in either two or three regions (The situation where the caller just puts it into one region can never happen because of the way we do region points-to analysis). If it is in two regions then obviously there will be no problem. When the input variable is in three regions it is still safe for the called procedure to regard only two regions because it never accesses the other one. These two factors will be captured by region points-to graph, which guides how terms need to be stored. Generally they will be stored in as many regions as needed but for some cases less regions can be used.

2.2 Merging regions

When analysing a program, at the beginning we can assume each variable appearing in the program is in a different region (to be precise we have to say a variable bound to a term that is stored in a region. But for short, we will just say a variable is stored in a region). Then based on their types and behaviour of the program we establish the relations between their regions. One important point in RBMM is which variables must be in the same region.

We want to allocate terms into different regions so that we can do timely removal of them. But the way a language is implemented has certain control on this matter. In an obviously accepted way of implementing computer languages, when an uninstantiated variable gets bound to a value, if that value does not exist in the heap then new memory will be allocated for it. But if that value is already there then the variable will be made point to the existing value.

In the context of RBMM, if the value exists then it must be in some existing region and some variable(s) is pointing to it. If the implementation of a language makes the uninstantiated variable point to the existing value, it forces the variable to be in the same region of the existing value. According to the above assumption, the uninstantiated variable has been assigned to a region different from the region of the existing value, therefore the effect

is like we “merge” the two regions. That is the basic reason of merging two regions of two variables when we build the region points-to graph. In principle, it means that if we want to (when we need a finite representation of recursive data structures) or have to (because of the accepted way computer languages are implemented) put two variables into the same region then we merge their regions into one.

There is a good reason to allocate a term into regions based on its structure. This is because programs likely treat values of the same structure in a term the same. By putting values of the same structure into the same region we can hopefully remove the region when a program no longer needs that part of the term while the other parts are still needed. We can think of a finer level of control, where each value is monitored by its own memory cells. But this approach soon becomes intractable. Region is seemingly a reasonable level of abstraction.

3 Region points-to analysis

The goal of this analysis is to build the region points-to graph for each procedure. The concept of region points-to graph used here was introduced for Java in [1]. Its details have been modified in the context of Mercury therefore we discuss it again here to make the paper self-contained and easier to read. Region points-to graph, $G = (N, E)$, consists of a set of nodes (N) representing regions and a set of directed edges (E) representing the references between the regions. A node is named by the variables that point to memory in the region corresponding to the node. From now on to avoid the lengthy expressions we use the concepts of node and of the region corresponding to a node interchangeably. An edge is labelled by a type selector [4], which represents the structured relation between variables in two regions. For example, if X is a variable of type `foo` that is defined:

```
:- type foo(T) --- > f(T), Y is a variable of type T and X = f(Y), then the points-to graph representing this is as in Figure 3. The points-to graph is the graphical representation of the splitting of the memory used by a procedure into regions.
```

The region points-to analysis is made up of two analyses. One is a flow-insensitive intraprocedural

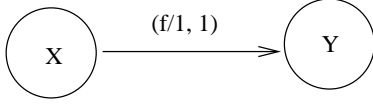


Figure 3: Points-to graph.

analysis that only deals with unifications, ignoring procedure calls and the other is an interprocedural analysis that integrates points-to graphs of the called procedures (callee) with that of the calling procedure (caller). The interprocedural analysis requires a fixpoint computation to calculate points-to graphs for recursive procedures. There are two operations used by the points-to analysis: *unify* and *edge*, which are defined as follows. If an operation is applied to a starting graph $G = (N, E)$ and ends up at a new graph $G' = (N', E')$ then

- *unify*(n, m): unify nodes n and m in the graph.
 - $N' = N - \{n, m\} \cup \{n \cup m\}$
 - $E' = \{(n', sel, m') \mid \exists (n, sel, m) \in E : n \subseteq n' \wedge m \subseteq m'\}$
- *edge*(n, sel, m): create an edge with label sel from node n to node m .
 - $G' = (N, E \cup \{(n, sel, m)\})$.

3.1 Intraprocedural analysis

To specify this analysis, assume that we are analysing a procedure p with points-to graph $G = (N, E)$. The analysis works as follows.

1. At the beginning, each variable in p is assigned to a separate node: $X \rightarrow n_X$, n_X becomes a node in N .
2. The unifications in the procedure are processed one by one as follows:
 - $X := Y$: *unify*(n_X, n_Y),
 - $X == Y$: do nothing,
 - $X => f(X_1, \dots, X_n)$: create references from n_X to each of n_{X_1}, \dots, n_{X_n} by drawing edges as follows
 - edge with label $(f/n, 1)$ from n_X to n_{X_1}

– ...

– edge with label $(f/n, n)$ from n_X to n_{X_n} ,

- $X \leq f(X_1, \dots, X_n)$: create references from n_X to each of n_{X_1}, \dots, n_{X_n}
 - edge with label $(f/n, 1)$ from n_X to n_{X_1}
 - ...
 - edge with label $(f/n, n)$ from n_X to n_{X_n} .

3. The following rules are fired whenever applicable

- *R1* :

after *unify*(n, n')
 if
 $m \neq m' \wedge$
 $(n, sel, m), (n', sel, m') \in E$
 then
unify(m, m')

- *R2* :

after *edge*(n, sel, m)
 if
 $m \neq m' \wedge$
 $(n, sel, m') \in E$
 then
unify(m, m')

- *R3* : This rule is to deal with variables that have recursive types.

after *edge*(n_X, sel, n_Y)
 if
 $(n_Z, n_Y) \in E^* \wedge$
 $n_Y \neq n_Z \wedge$
 $type(Y) = type(Z)$
 then
unify(n_Y, n_Z)

in which E^* is the transitive closure of E and $type(X)$ returns the type of variable X .

3.2 Interprocedural analysis

When interprocedural analysis is executed for a procedure, the following assumptions are made:

- The points-to graphs of the callees are available or are the current points-to graph in case the procedure is recursive.
- For a call $q(Y1, \dots, Yn)$, the formal declaration of q/n is $q(X1, \dots, Xn)$.

The interprocedural analysis is performed as follows.

1. Process each procedure call in the procedure, for a call $q(Y1, \dots, Yn)$: integrate the graph of q/n , $G_e = (N_e, E_e)$, into the graph of the currently analysed procedure, $G_r = (N_r, E_r)$ by building the partial α mapping from N_e to N_r as follows.

- – $\alpha(n_{X1}) = n_{Y1}$
– ...
– $\alpha(n_{Xn}) = n_{Yn}$
- In the graph G_e , start from each n_{Xi} , follow each edge once and apply the following rules when appropriate
 - R4 :
if
 $\alpha(n_e) = n_r \wedge$
 $(n_e, sel, m_e) \in E_e \wedge$
 $(n_r, sel, m_{r'}) \in E_r \wedge$
 $\alpha(m_e) = m_r \neq m_{r'}$
then
 $unify(m_r, m_{r'})$
 - R5 :
if
 $\alpha(n_e) = n_r \wedge$
 $(n_e, sel, m_e) \in E_e \wedge$
 $(n_r, sel, m_r) \in E_r \wedge$
 $\alpha(m_e)$ undefined
then
 $\alpha(m_e) = m_r$
 - R6 :
if
 $\alpha(n_e) = n_r \wedge$
 $(n_e, sel, m_e) \in E_e \wedge$
 $\forall p : (n_r, sel, p) \notin E_r \wedge$
 $\alpha(m_e) = m_r$
then
 $edge(n_r, sel, m_r)$
 - R7 :

if
 $\alpha(n_e) = n_r \wedge$
 $(n_e, sel, m_e) \in E_e \wedge$
 $\forall p : (n_r, sel, p) \notin E_r \wedge$
 $\alpha(m_e)$ undefined
 $m_r = FreshNode(G_r)$
then
 $\alpha(m_e) = m_r,$
 $edge(n_r, sel, m_r)$

- Record renaming of regions at this call site.

– R8 :

if
 $\alpha(n_{Xi}) = n_{Yi} \wedge$
 $\alpha(n_{Xj}) = n_{Yj} \wedge$
 $n_{Xi} = n_{Xj} \wedge$
 Xi is an input variable of q/n
 Xj is an output variable of q/n
then
 $rename(n_{Yi}, n_{Yj}, G_r).$

– $rename(p, q, G(N, E))$: a help procedure

First: record rename p to q
Then: follow G from p and q , each time with the same selector, do either of the followings.

- (a) if $(p, sel, m), (q, sel, n) \in E \wedge m \neq n$
then $rename(m, n, G).$
- (b) if $(p, sel, m) \in E \wedge p \neq m \wedge \nexists n : (q, sel, n) \in E$
then $edge(q, sel, m).$
- (c) if $(q, sel, m) \in E \wedge q \neq m \wedge \nexists n : (p, sel, n) \in E$
then $edge(p, sel, m).$
- (d) if $(p, sel, p) \in E \wedge (q, sel, q) \notin E$
then $edge(q, sel, q).$
- (e) if $(q, sel, q) \in E \wedge (p, sel, p) \notin E$
then $edge(p, sel, p).$

The reason of R8 is as follows. Consider a case when the caller provides a callee with an input region and expects the result of the callee in a different region but the callee by its own puts

its output into the input region. We have two options here. We can either unify the input region and the output region in the caller's points-to graph (because from the behaviour of the callee we know that they are actually the same) or keep the two regions separate in the caller's graph but let the callee inform the caller about that fact. The first option is safe but too conservative, causing less precision in the case, for example, the two regions are the same in only one branch of execution of the callee but not in the other branches. Rule R8 realizes the second option, where the callee renames the input region to the one expected by the caller. The effect of this rule can be seen in the analysis and transformation of the program Game of Life in Section 6.

2. The procedure will be analysed iteratively until there is no change in its points-to graph.

4 Live region analysis

The goal of live region analysis is to detect live regions at each program point and to compute the information about which regions will be created and removed by each procedure. These pieces of information will be used to transform the original program to the program with region-based memory management. To define this analysis we assume the definitions of program point and execution path for Mercury as in Nancy's thesis [4]. The set of live regions at a program point is computed via the set of live variables at the program point. So first we define the concept of live variables at a program point.

4.1 Live variables at a program point

A variable is live at a program point if:

- There exists an execution path containing the program point that instantiates the variable before the program point and uses it at or after the program point
- OR it is an output variable that is instantiated before the program point.

If we define $pre_inst(pp, P)$ the set of variables instantiated before the program point pp in the execution path P , $post_use(pp, P)$ the set of variables

used at or after pp in the execution path P , $out(p)$ the set of output variables of a procedure p then the set of live variables at a program point i is:

$$LV(i) = \{V \mid \exists P : (V \in pre_inst(i, P)) \wedge (V \in out(p) \vee V \in post_use(i, P))\}$$

We define two exceptional cases of the above formula.

- If $i = first(P)$ then $LV(i) =$ set of input variables of p . $first(P)$ returns the first program point in P .
- Each execution path of a procedure is extended to one more program point called "out". $LV(out)$ is the set of output variables of a procedure.

4.2 Live regions at a program point

A region is live at a program point if it is reachable from a live variable at the program point. The set of regions that are reachable from a variable is defined:

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

in which $E^*(X)$, the transitive closure of E from X , is defined:

$$E^*(X) = \{(n_X, n_i) \mid \exists(n_X, sel_0, n_1), \dots, (n_{i-1}, sel_{i-1}, n_i) \in E \wedge sel_0 \in TT_{type}(X), sel_1 \in TT_{type}((X, sel_0)), \dots, sel_{i-1} \in TT_{type}((X, sel_0 \bullet sel_1 \bullet \dots \bullet sel_{i-2}))\}.$$

The set of live regions at a program point i is defined:

$$LR(i) = \bigcup(Reach(X)) \forall X \in LV(i).$$

4.3 The analysis

This analysis computes the set of live regions (LR) at each program point and for each procedure the set of regions that the procedure may create, called *bornR* and the set of regions that it may remove, called *deadR*. We define several sets of regions, which live region analysis will care about, for each procedure p with its points-to graph $G = (N, E)$:

- $inputR(p)$ is the set of regions reachable from input variables.

- $outputR(p)$ is the set of regions reachable from output variables.
- $bornR(p) = outputR(p) - inputR(p)$ is the set of output regions that the procedure or any of the procedures it calls may create.
- $deadR(p) = inputR(p) - outputR(p)$ is the set of input regions that the procedure or any of the procedures it calls may remove.
- $localR(p) = N - input(p) - output(p)$.

The analysis can be performed in two passes. The first pass is to compute live variables (LV) at each program point. The second pass computes live regions (LR) at each program point. For a program point that is a call to procedure q we assume that the called procedure, q , has been analysed so the sets $deadR(q)$ and $bornR(q)$ have been computed. These sets of q (callee) may be updated by the rules defined below when analysing p (caller). While computing live regions the analysis will try to apply the rules when applicable. To specify the rules we define some help functions:

- $pp(l)$: return the program point of the literal l ,
- $next(pp(l))$: return the next program point of l in an execution path. As each execution path is already extended to one more final point if l is the last literal then $next(pp(l)) = out$.

The rules and their reasons are defined as follows.

- $L1$: if a region that is supposed to be removed by the called procedure is still live after the call then it is excluded from the $deadR$ set of the called procedure.

if
 $l \equiv q(\dots) \wedge$
 $r \in LR(pp(l)) \wedge$
 $r \in LR(next(pp(l))) \wedge$
 $r = \alpha(r') \wedge$
 $r' \in deadR(q)$
then
 $deadR(q) = deadR(q) - \{r'\}$

- $L2$: if a region that is supposed to be created by the called procedure has already been created by the calling one then it is excluded from the $bornR$ set of the called procedure.

if
 $l \equiv q(\dots) \wedge$
 $r \in LR(pp(l)) \wedge$
 $r = \alpha(r') \wedge$
 $r' \in bornR(q)$
then
 $bornR(q) = bornR(q) - \{r'\}$

5 Program transformation

The goal of program transformation is to introduce "create" and "remove" statements based on region liveness information. The transformation is executed for each procedure. It follows each execution path and applies the following transformation rules when appropriate. In the specification of the rules below assume we are analysing procedure p .

- $T1$: if an output region for a procedure call (q) is not live and it is not created by the called procedure (q), it is created before the call.

if
 $l \equiv q(\dots) \wedge$
 $R = apply_renaming(LR(pp(l))) \wedge$
 $r \notin R \wedge$
 $r \in LR(next(pp(l))) \wedge$
 $r = \alpha(r') \wedge$
 $r' \notin bornR(q)$
then
add "create r " before l

$apply_renaming(LR)$: apply the renaming information, if available, at the call site to the set of regions LR and return the renamed set of regions.

- $T2$: if a region is not live before a unification (construction) but it is live after and is a local region or a born region, the region is created before the unification.

if
 $l \equiv X \leq f(\dots) \wedge$
 $r_S \notin LR(pp(l)) \wedge$
 $r_S \in LR(next(pp(l))) \wedge$
 $(r_S \in localR(p) \cup bornR(p)) \wedge$
 $X \in S$
then
add "create r_S " before l

- $T3$: if a region is live before a procedure call but it is not live after the call, and the procedure does not remove it then the caller will remove that region.

if

$$l \equiv q(\dots) \wedge$$

$$R = \text{apply_renaming}(LR(pp(l)) \wedge$$

$$r \in R \wedge$$

$$r \notin LR(\text{next}(pp(l))) \wedge$$

$$(r \in \text{local}R(p) \cup \text{dead}R(p))$$

$$\nexists r' : (\alpha(r') = r \wedge r' \notin \text{dead}R(q))$$

then

add "remove r " before $\text{next}(pp(l))$

- $T4$: if a region is live before a unification but it is not live after and it is in the dead region set or is a local region, then it is removed after the unification.

if

$$l \equiv \text{unif} \wedge$$

$$r \in LR(pp(l)) \wedge$$

$$r \notin LR(\text{next}(pp(l))) \wedge$$

$$(r \in \text{local}R(p) \cup \text{dead}R(p))$$

then

add "remove r " before $\text{next}(pp(l))$

6 Analysis and transformation of some programs

All the programs have only one module and are written explicitly in the normal form, with all unifications already specialized, and goals reordered. The analysis performed here does not consider the combination with reuse (CTGC) and the extension to deal with non-determinism.

6.1 Naive reverse

This program is the deterministic version of naive reverse (nrev), where nrev predicate is declared with mode (in, out) and append predicate with mode (in, in, out).

append(X, Y, Z) :-

(

- (1) $X \Rightarrow []$,
- (2) $Z := Y$

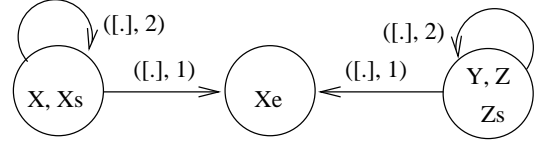


Figure 4: The points-to graph of append after the intraprocedural analysis.

;

- (3) $X \Rightarrow [Xe \mid Xs]$,
- (4) append(Xs, Y, Zs),
- (5) $Z \Leftarrow [Xe \mid Zs]$

).

nrev(L, R) :-

(

- (1) $L \Rightarrow []$,
- (2) $R \Leftarrow []$

;

- (3) $L \Rightarrow [H \mid T]$,
- (4) nrev(T, V),
- (5) $L1 \Leftarrow [H]$,
- (6) append(V, L1, R)

).

6.1.1 Region points-to analysis

The analysis for append:

At the beginning, each variable appears in append is assigned to a different node. Therefore we have the following nodes: $n_X, n_Y, n_Z, n_{Xe}, n_{Xs}, n_{Zs}$.

1. Intraprocedural analysis:

(1): do nothing.

(2): $\text{unify}(n_Z, n_Y)$.

(3): $\text{edge}(n_X, ([.], 1), n_{Xe})$

$\text{edge}(n_X, ([.], 2), n_{Xs}) : \text{type}(X) = \text{type}(Xs)$
so $\text{unify}(n_X, n_{Xs})$ (R3).

(5): $\text{edge}(n_Z, ([.], 1), n_{Xe})$

$\text{edge}(n_Z, ([.], 2), n_{Zs}) : \text{type}(Z) = \text{type}(Zs)$ so
 $\text{unify}(n_Z, n_{Zs})$ (R3).

The points-to graph of append after the intraprocedural analysis is shown in Figure 4.

2. Interprocedural analysis:

(4): This is the recursive call to append(Xs, Y, Zs). So we assume the current points-to graph

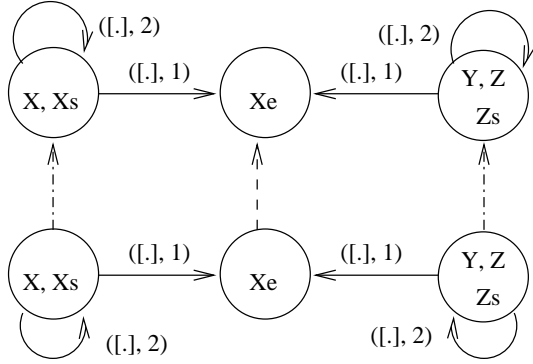


Figure 5: The mapping from formal to actual regions. The dash-dot arrows are mapping from the call, the dash arrow is due to rule R5. The caller's graph is above and the callee's graph is below.

of append and produce the mapping from regions of formal parameters to those of actual parameters.

$$\alpha(n_X) = n_{Xs}, \alpha(n_Y) = n_Y, \alpha(n_Z) = n_{Zs}.$$

From rule R5 we have $\alpha(n_{Xe}) = n_{Xe}$. The mapping and the resulting graph after this step is shown in Figure 5.

Fixpoint reached.

The analysis for nrev:

Nodes: $n_L, n_R, n_H, n_T, n_V, n_{L1}$.

1. Intraprocedural analysis:

(1): do nothing.

(2): do nothing.

(3): $edge(n_L, ([.], 1), n_H)$

$edge(n_L, ([.], 2), n_T) : type(L) = type(T)$ so $unify(n_L, n_T)$ (R3).

(5): $edge(n_{L1}, ([.], 1), n_H)$

The points-to graph of nrev after the intraprocedural analysis is shown in Figure 6.

2. Interprocedural analysis:

(6): $append(V, L1, R)$

$$\alpha(n_X) = n_V, \alpha(n_Y) = n_{L1}, \alpha(n_Z) = n_R.$$

By applying the rules we have:

$$\alpha(n_{Xe}) = n_H \text{ (R5),}$$

$$edge(n_{L1}, ([.], 2), n_{L1}) \text{ (R6),}$$

$$edge(n_V, ([.], 2), n_V) \text{ (R6),}$$

$$edge(n_R, ([.], 2), n_R) \text{ (R6),}$$

$$edge(n_V, ([.], 1), n_H) \text{ (R6),}$$

$$edge(n_R, ([.], 1), n_H) \text{ (R6),}$$

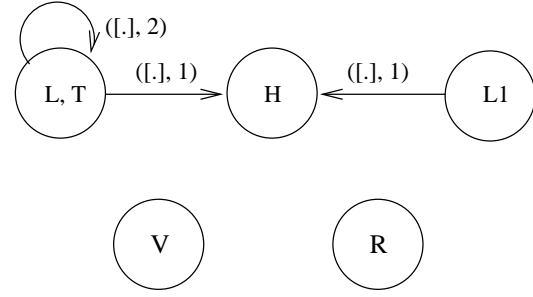


Figure 6: The points-to graph of nrev after the intraprocedural analysis.

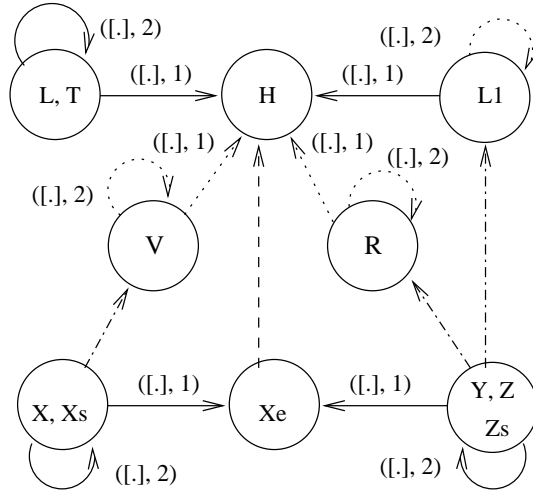


Figure 7: The interprocedural analysis of nrev, at program point (6). The dot arrows are references due to rule R6.

renaming n_{L1} to n_R (R8, only the first action happens, none of the cases in the second action satisfied).

The resulting graph after this step is shown in Figure 7.

(4): $nrev(T, V)$

$$\alpha(n_L) = n_T, \alpha(n_R) = n_V.$$

$$\alpha(n_H) = n_H \text{ (R5).}$$

The resulting graph after this step is shown in Figure 8.

Fixpoint reached.

6.1.2 Live region analysis

Live region analysis for append:

The deterministic version of append has 2 execu-

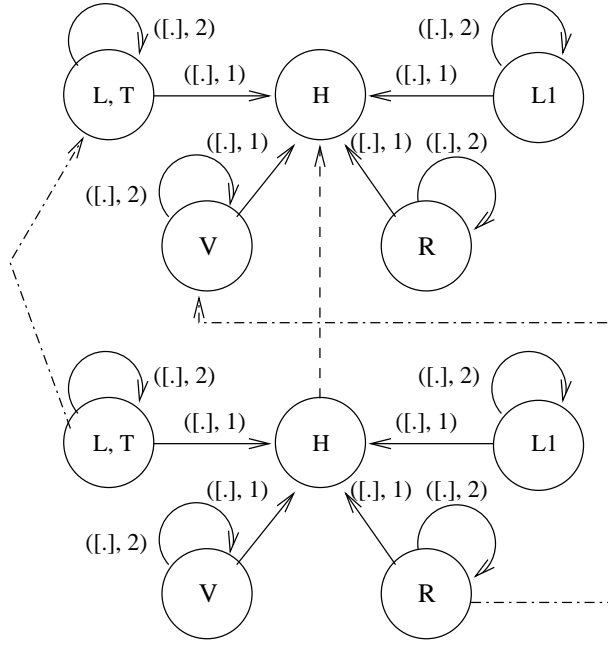


Figure 8: The interprocedural analysis of *nrev*, at program point (4).

tion paths:

- (1) (2) out
- (3) (4) (5) out

At the beginning, the sets of regions of *append* are:

- $inputR(append) = \{n_{\{X, X_s\}}, n_{X_e}, n_{\{Y, Z, Z_s\}}\}$
- $outputR(append) = \{n_{X_e}, n_{\{Y, Z, Z_s\}}\}$
- $deadR(append) = \{n_{\{X, X_s\}}\}$
- $bornR(append) = \phi$
- $localR(append) = \phi$

The results of live variable and live region analyses are summarized in the below tables for each execution path.

pp	LV	LR
(1)	$\{X, Y\}$	$\{n_{\{X, X_s\}}, n_{X_e}, n_{\{Y, Z, Z_s\}}\}$
(2)	$\{Y\}$	$\{n_{X_e}, n_{\{Y, Z, Z_s\}}\}$
out	$\{Z\}$	$\{n_{X_e}, n_{\{Y, Z, Z_s\}}\}$
pp	LV	LR
(3)	$\{X, Y\}$	$\{n_{\{X, X_s\}}, n_{X_e}, n_{\{Y, Z, Z_s\}}\}$
(4)	$\{X_e, X_s, Y\}$	$\{n_{\{X, X_s\}}, n_{X_e}, n_{\{Y, Z, Z_s\}}\}$
(5)	$\{X_e, Z_s\}$	$\{n_{X_e}, n_{\{Y, Z, Z_s\}}\}$
out	$\{Z\}$	$\{n_{X_e}, n_{\{Y, Z, Z_s\}}\}$

No rules are applicable at program point (4) (the only procedure call in *append*) in the live region analysis for *append*.

The sets of dead and born regions after this analysis.

$$deadR(append) = \{n_{\{X, X_s\}}\},$$

$$bornR(append) = \phi.$$

Live region analysis for *nrev*:

Execution paths:

- (1) (2) out
- (3) (4) (5) (6) out

Sets of regions:

- $inputR(nrev) = \{n_{\{L, T\}}, n_H\}$
- $outputR(nrev) = \{n_H, n_R\}$
- $deadR(nrev) = \{n_{\{L, T\}}\}$
- $bornR(nrev) = \{n_R\}$
- $localR(nrev) = \{n_V, n_{L1}\}$

The results of live variable and live region analyses for each execution path.

pp	LV	LR
(1)	$\{L\}$	$\{n_{\{L, T\}}, n_H\}$
(2)	$\{\}$	$\{\}$
out	$\{R\}$	$\{n_H, n_R\}$
pp	LV	LR
(3)	$\{L\}$	$\{n_{\{L, T\}}, n_H\}$
(4)	$\{H, T\}$	$\{n_{\{L, T\}}, n_H\}$
(5)	$\{H, V\}$	$\{n_V, n_H\}$
(6)	$\{V, L1\}$	$\{n_V, n_{L1}, n_H\}$
out	$\{R\}$	$\{n_R, n_H\}$

No rules are applicable at program point (4) and (6).

The sets of dead and born regions are:

$$deadR(append) = \{n_{\{X, X_s\}}\},$$

$$bornR(append) = \phi,$$

$$deadR(nrev) = \{n_{\{L, T\}}\},$$

$$bornR(nrev) = \{n_R\}.$$

6.1.3 Transformation

For *append*:

- (1): rule T4 applied, so add “remove r_X ” before
- (2). Note that no “remove r_X ” is added before (5) because n_X belongs to $deadR(append)$, i.e. the callee will remove it, not the caller.

For nrev:

(1): add remove r_L before (2) (T4). Note that r_H is not removed even it is not live at (2), this is because n_H does not belong to $deadR(nrev)$.

(2): add create r_R before (2) (T2).

(5): add create r_{L1} before (5) (T2).

At (6) because the renaming, which has been recorded at this call site in region points-to analysis, renames n_{L1} to n_R the rule T1 fails to apply, preventing the transformation from creating region r_R .

The transformed program is as follows:

append(X, Y, Z) :-

```
(
  (1) X => [],
  remove r_X,
  (2) Z := Y
;
  (3) X => [Xe | Xs],
  (4) append(Xs, Y, Zs),
  (5) Z <= [Xe | Zs]
).
```

nrev(L, R) :-

```
(
  (1) L => [],
  remove r_L,
  create r_R,
  (2) R <= []
;
  (3) L => [H | T],
  (4) nrev(T, V),
  create r_{L1},
  (5) L1 <= [H],
  (6) append(V, L1, R)
).
```

6.1.4 Comparison with the result of Henning and Kostis

The program *nrev* with region support created by their prototype in [3] is shown below.

append(X,Y,Z) c(X0, R10, R11) :-

```
(
  X = [],
  Z = Y
;
  X = R10.[X0.Xe | R10.Xs],
  append(Xs,Y,Zs) c(X0, R10, R11),
```

Z = R11.[Xe | Zs]

).

nrev(L,R) c(X0)i(R9)o(R0) :-

```
(
  L = [],
  release R9,
  new R0,
  R = []
;
  L = R9.[X0.H | R9.T],
  nrev(T, V) c(X0)i(R9)o(R4),
  new R0,
  append(V, R0.[H], R) c(X0, R4, R0),
  release R4
).
```

In this work the release of the region of the list backbone of the first input list of *append* is performed in *nrev*, a caller of *append*. This means that the lifetime of the region here is longer than in our result, where the removal happens inside *append*. I speculate that this is because their region inferencer cannot do a better job. The impact of this imprecision on memory use is not significant here but it is the case in the *qsort* example.

6.2 qsort

The analysis and transformation for *qsort* is done with the assumption that values of primitive types are also stored in the heap memory. The version of *qsort* here is written for type list(int). *split* is declared with mode (in, in, out, out) and *qsort* with mode (in, in, out).

split(X, L, L1, L2) :-

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

```

)
).
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)
).

```

6.2.1 Region points-to analysis

The analysis for split:

Nodes: $n_X, n_L, n_{L1}, n_{L2}, n_{Le}, n_{Ls}, n_{L11}, n_{L21}$.

1. Intraprocedural analysis:
 - (1), (2), (3): do nothing.
 - (4): $edge(n_L, ([\cdot], 1), n_{Le})$
 $edge(n_L, ([\cdot], 2), n_{Ls}) : type(L) = type(Ls)$ so
 $unify(n_L, n_{Ls})$ (R3).
 - (5): do nothing.
 - (7): $edge(n_{L1}, ([\cdot], 1), n_{Le})$
 $edge(n_{L1}, ([\cdot], 2), n_{L11}) : type(L1) = type(L11)$
so $unify(n_{L1}, n_{L11})$ (R3)
 - (9): $edge(n_{L2}, ([\cdot], 1), n_{Le})$
 $edge(n_{L2}, ([\cdot], 2), n_{L21}) : type(L2) = type(L21)$
so $unify(n_{L2}, n_{L21})$ (R3)
2. Interprocedural analysis:

The interprocedural analysis for split does not change the shape of the points-to graph produced by intraprocedural analysis.

(6): $split(X, Ls, L1, L2)$
 $\alpha(n_X) = n_X, \alpha(n_L) = n_{Ls}, \alpha(n_{L1}) = n_{L11},$
 $\alpha(n_{L2}) = n_{L21}$.
By applying the rules we have:
 $\alpha(n_{Le}) = n_{Le}$ (R5).

(8): $split(X, Ls, L1, L2)$
 $\alpha(n_X) = n_X, \alpha(n_L) = n_{Ls}, \alpha(n_{L1}) = n_{L1},$
 $\alpha(n_{L2}) = n_{L21}$.
By applying the rules we have:
 $\alpha(n_{Le}) = n_{Le}$ (R5).

Fixpoint reached and the graph is shown in Figure 9 (This is also the graph after intraprocedural analysis).

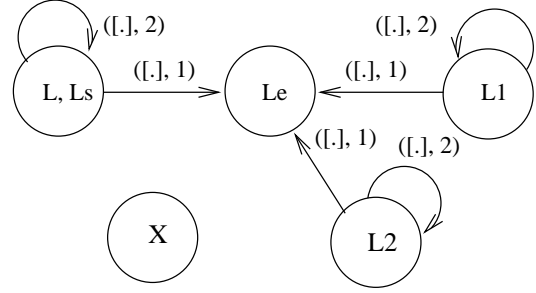


Figure 9: The points-to graph of split.

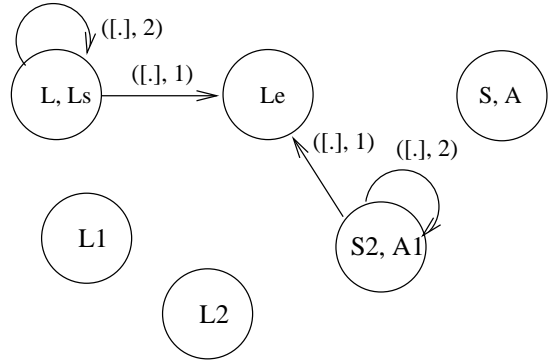


Figure 10: The points-to graph of qsort after intraprocedural analysis.

The analysis for qsort:

Nodes: $n_L, n_A, n_S, n_{Le}, n_{Ls}, n_{L1}, n_{L2}, n_{S2}, n_{A1}$.

1. Intraprocedural analysis:
 - (1): do nothing.
 - (2): $unify(n_S, n_A)$
 - (3): $edge(n_L, ([\cdot], 1), n_{Le})$
 $edge(n_L, ([\cdot], 2), n_{Ls}) : type(L) = type(Ls)$ so
 $unify(n_L, n_{Ls})$ (R3)
 - (6): $edge(n_{A1}, ([\cdot], 1), n_{Le})$
 $edge(n_{A1}, ([\cdot], 2), n_{S2}) : type(A1) = type(S2)$
so $unify(n_{A1}, n_{S2})$ (R3)

The graph after this step is shown in Figure 10.
2. Interprocedural analysis:
 - (4): $split(Le, Ls, L1, L2)$
 $\alpha(n_X) = n_{Le}, \alpha(n_L) = n_{Ls}, \alpha(n_{L1}) = n_{L1},$
 $\alpha(n_{L2}) = n_{L2}$.
By applying the rules we have:
 $\alpha(n_{Le}) = n_{Le}$ (R5).
 $edge(n_{L1}, ([\cdot], 1), n_{Le})$ (R6).
 $edge(n_{L1}, ([\cdot], 2), n_{L1})$ (R6).

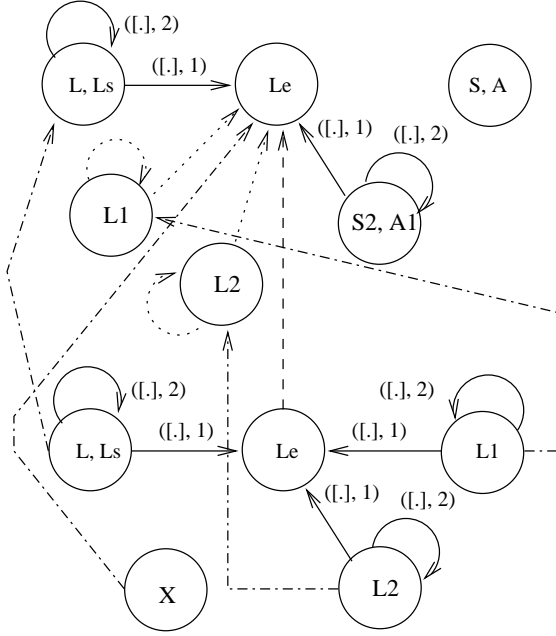


Figure 11: The interprocedural analysis of qsort, at program point (4). The dot arrows are references due to rule R6.

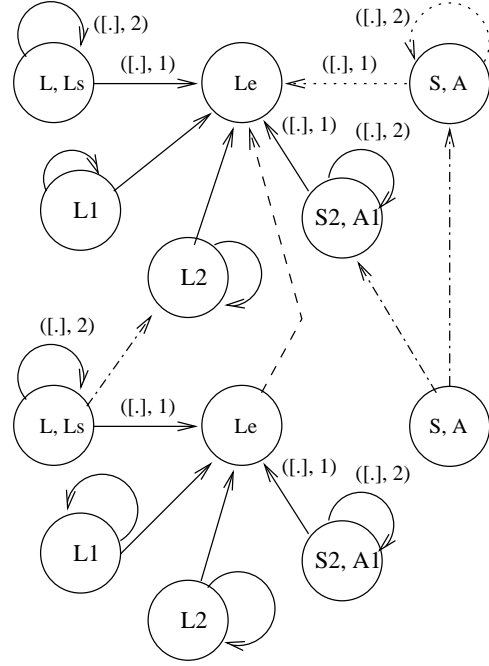


Figure 12: The interprocedural analysis of qsort, at program point (5). The dot arrows are references due to rule R8 (renaming).

$edge(n_{L2}, ([., 1], n_{Le}))$ (R6).

$edge(n_{L2}, ([., 2], n_{L2}))$ (R6).

The graph after this step is shown in Figure 11.

(5): qsort(L2, A, S2)

$\alpha(n_L) = n_{L2}$, $\alpha(n_A) = n_A$, $\alpha(n_S) = n_{S2}$.

By applying the rules we have:

$\alpha(n_{Le}) = n_{Le}$ (R5).

renaming n_A to n_{S2} (R8): causing $edge(n_A, ([., 1], n_{Le}))$ (R8, c),

$edge(n_A, ([., 2], n_A))$ (R8, e).

The graph after this step is shown in Figure 12.

(7): qsort(L1, A1, S)

$\alpha(n_L) = n_{L1}$, $\alpha(n_A) = n_{A1}$, $\alpha(n_S) = n_S$.

By applying the rules we have:

$\alpha(n_{Le}) = n_{Le}$ (R5).

renaming n_{A1} to n_s (R8, only the first action happens, none of the cases in the second action satisfied).

This step does not change the shape of the points-to graph of qsort. Fixpoint reached.

6.2.2 Live region analysis

Live region analysis for split:

Execution paths:

- (1) (2) (3) out
- (4) (5) (6) (7) out
- (4) (8) (9) out

At the beginning, the sets of regions of split are:

- $inputR(split) = \{n_X, n_{\{L, Ls\}}, n_{Le}\}$
- $outputR(split) = \{n_{Le}, n_{\{L1, L11\}}, n_{\{L2, L21\}}\}$
- $deadR(split) = \{n_X, n_{\{L, Ls\}}\}$
- $bornR(split) = \{n_{\{L1, L11\}}, n_{\{L2, L21\}}\}$
- $localR(split) = \phi$

The results of live variable and live region analyses for each execution path.

pp	LV	LR
(1)	$\{L, X\}$	$\{n_L, n_{Le}, n_X\}$
(2)	$\{\}$	$\{\}$
(3)	$\{L1\}$	$\{n_{L1}, n_{Le}\}$
out	$\{L1, L2\}$	$\{n_{L1}, n_{Le}, n_{L2}\}$

pp	LV	LR
(4)	$\{L, X\}$	$\{n_{\{L, Ls\}}, n_{Le}, n_X\}$
(5)	$\{X, Le, Ls\}$	$\{n_{\{L, Ls\}}, n_{Le}, n_X\}$
(6)	$\{X, Le, Ls\}$	$\{n_{\{L, Ls\}}, n_{Le}, n_X\}$
(7)	$\{L11, Le, L2\}$	$\{n_{\{L1, L11\}}, n_{Le}, n_{\{L2, L21\}}\}$
out	$\{L1, L2\}$	$\{n_{\{L1, L11\}}, n_{Le}, n_{\{L2, L21\}}\}$

pp	LV	LR
(4)	$\{L, X\}$	$\{n_{\{L, Ls\}}, n_{Le}, n_X\}$
(8)	$\{X, Le, Ls\}$	$\{n_{\{L, Ls\}}, n_{Le}, n_X\}$
(9)	$\{L1, Le, L21\}$	$\{n_{\{L1, L11\}}, n_{Le}, n_{\{L2, L21\}}\}$
out	$\{L1, L2\}$	$\{n_{\{L1, L11\}}, n_{Le}, n_{\{L2, L21\}}\}$

At program point (6): no rules applied.

At program point (8): no rules applied.

The sets of dead and born regions are:

$$deadR(split) = \{n_X, n_{\{L, Ls\}}\},$$

$$bornR(split) = \{n_{\{L1, L11\}}, n_{\{L2, L21\}}\}.$$

Live region analysis for qsort:

Execution paths:

- (1) (2) out
- (3) (4) (5) (6) (7) out

At the beginning, the sets of regions of qsort are:

- $inputR(qsort) = \{n_{\{L, Ls\}}, n_{Le}, n_{\{S, A\}}\}$
- $outputR(qsort) = \{n_{\{S, A\}}, n_{Le}\}$
- $deadR(qsort) = \{n_{\{L, Ls\}}\}$
- $bornR(qsort) = \phi$
- $localR(qsort) = \{n_{L1}, n_{L2}, n_{\{A1, S2\}}\}$

The results for live variable and live region analyses for each execution path.

pp	LV	LR
(1)	$\{L, A\}$	$\{n_{\{L, Ls\}}, n_{Le}, n_{\{A, S\}}\}$
(2)	$\{A\}$	$\{n_{\{A, S\}}, n_{Le}\}$
out	$\{S\}$	$\{n_{\{A, S\}}, n_{Le}\}$

pp	LV	LR
(3)	$\{L, A\}$	$\{n_{\{L, Ls\}}, n_{Le}, n_{\{A, S\}}\}$
(4)	$\{A, Le, Ls\}$	$\{n_{\{A, S\}}, n_{Le}, n_{\{L, Ls\}}\}$
(5)	$\{A, Le, L1, L2\}$	$\{n_{\{A, S\}}, n_{Le}, n_{L1}, n_{L2}\}$
(6)	$\{Le, L1, S2\}$	$\{n_{Le}, n_{L1}, n_{\{A1, S2\}}\}$
(7)	$\{L1, A1\}$	$\{n_{L1}, n_{Le}, n_{\{A1, S2\}}\}$
out	$\{S\}$	$\{n_{\{A, S\}}, n_{Le}\}$

At program point (4): rule L1 applied,

$\alpha(n_X) = n_{Le}$, n_{Le} is live at (4) and at (5) also therefore $deadR(split) = deadR(split) - \{n_X\} = \{n_X, x_L\} - \{n_X\} = \{n_L\}$.

The sets of dead and born regions after this analysis are:

$$deadR(split) = \{n_{\{L, Ls\}}\},$$

$$bornR(split) = \{n_{\{L1, L11\}}, n_{\{L2, L21\}}\},$$

$$deadR(qsort) = \{n_{\{L, Ls\}}\},$$

$$bornR(qsort) = \phi.$$

6.2.3 Transformation

For split:

(1): add remove r_L after (1) (T4).

(2): add create r_{L1} before (2) (T2).

(3): add create r_{L2} before (3) (T2).

For qsort:

(1): add remove r_L after (1) (T4).

At program point (5): rename n_A to n_{S2} , i.e. the call to $qsort$ here puts S2 into the same region of A.

At program point (7): rename n_{A1} to n_S , i.e. the call to $qsort$ here puts S into the same region of A1

The transformed program is as follows:

split(X, L, L1, L2) :-

```
(
  (1) L => [],
  remove r_L,
  create r_L1,
  (2) L1 <= [],
  create r_L2,
  (3) L2 <= []
;
  (4) L => [Le | Ls],
  (
    (5) X >= Le
  - >
  (6) split(X, Ls, L11, L2),
  (7) L1 <= [Le | L11]
;
  (8) split(X, Ls, L1, L21),
  (9) L2 <= [Le | L21]
)
).
```

qsort(L, A, S) :-

```
(
  (1) L => [],
  remove r_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)
).

```

6.2.4 Comparison with the result of Henning and Kostis

The *qsort* program with region support generated by their prototype in [3] is shown below.

```

split(X, L, L01, L02) c(X0, R17)o(R0, R1) :-
(
  L = [],
  L01 = [],
  L02 = [],
  new R0,
  new R1
;
  L = R17.[X0.Le| R17.Ls],
  (X >= Le
  - >
  split(X, Ls, L011, L02) c(X0, R17)o(R0, R1),
  L01 = R0.[Le|L011]
;
  split(X,Ls,L01,L021) c(X0, R17)o(R0, R1),
  L02 = R1.[Le|L021]
)
).

```

```

qsort(L,A,S) c(X0, R15)i(R14) :-
(
  L = [],
  release R14,
  S = A
;
  L = R14.[X0.Le | R14.Ls],
  split(Le, Ls, L01, L02) c(X0, R14)o(R4, R6),
  release R14,
  qsort(L02, A, S02) c(X0, R15)i(R6),
  qsort(L01, R15.[Le | S02], S) c(X0, R15)i(R4)
).

```

In this example, again, the removal of the region of the input list backbone of *split* happens in the caller of *split*, while it is actually dead inside. By being able to removing it inside *split* before creating the

regions for the two sublists our transformed program will use no more memory than what needed to store the original list.

6.3 Game of Life

We assume a simplified, fake implementation of *nextgen* in the discussion below. The only essential point of *nextgen* here is that its behaviour causes output regions different from input ones. So if our analysis is precise enough the input regions of *nextgen*, as it is called in *life*, should be removed by itself. The built-in *is*, as any other built-ins, is treated like *deadR(is)* and *bornR(is)* are empty. Namely, it will only read from or write to regions, never create or remove a region. *nextgen* is declared with mode (in, out) and *life* with mode (in, in, out).

```

nextgen(G, G1) :-
(1) G => gen(A),
(2) A1 is A + 2,
(3) G1 <= gen(A1).

```

```

life(N, G, H) :-
(
  (1) N => 0
- >
  (2) H := G
;
  (3) N1 is N - 1,
  (4) nextgen(G, G1),
  (5) life(N1, G1, H)
).

```

6.3.1 Region points-to analysis

Analysis for nextgen:

Nodes: n_G, n_{G1}, n_A, n_{A1} .

1. Intraprocedural analysis:

(1): $edge(n_G, (gen/1, 1), n_A)$.

(4): $edge(n_{G1}, (gen/1, 1), n_{A1})$.

No interprocedural analysis happens for *nextgen*.

Fixpoint reached and the graph is shown in Figure 13.

Analysis for life:

Nodes: $n_N, n_G, n_H, n_{N1}, n_{G1}$.

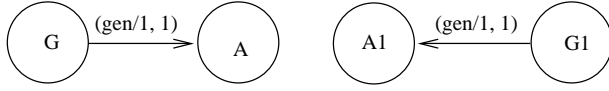


Figure 13: The points-to graph of nextgen.

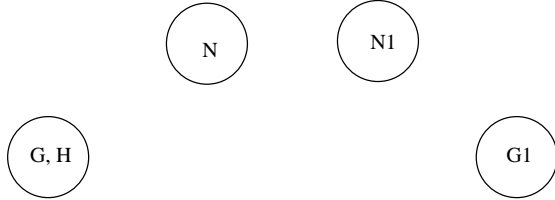


Figure 14: The points-to graph of life after intraprocedural analysis.

1. Intraprocedural analysis:

(1): do nothing.

(2): $unify(n_H, n_G)$.

The graph after this step is shown in Figure 14.

2. Interprocedural analysis:

(4): $nextgen(G, G1)$ $\alpha(n_G) = n_G, \alpha(n_{G1}) = n_{G1}$.R7: new node $m1$, $edge(n_G, (gen/1, 1), m1)$, $\alpha(n_A) = m1$.R7: new node $m2$, $edge(n_{G1}, (gen/2, 1), m2)$, $\alpha(n_{A1}) = m2$.

The graph after this step is shown in Figure 15.

(5): $life(N1, G1, H)$ $\alpha(n_N) = n_{N1}, \alpha(n_G) = n_{G1}, \alpha(n_H) = n_H$.R8: $renaming(n_{G1}, n_H, G_{life})$, causing rename n_{G1} to n_H , $m2$ to $m1$.

At this point if we had chosen to unify n_{G1} and n_H in the caller's graph instead of renaming, the temporary values in *life* would have been put into the same region of the output and not been able to be removed. This is because n_H has been unified with n_G already, so after this merging H, G and $G1$ are in the same region. This will prevent the call to $nextgen(G, G1)$ from removing the region of G . Fixpoint reached and the graph is shown in Figure 16.

6.3.2 Live region analysis**Live region analysis for nextgen:**

Execution paths: (1) (2) (3) (4) out.

At the beginning, the sets of regions of nextgen are:

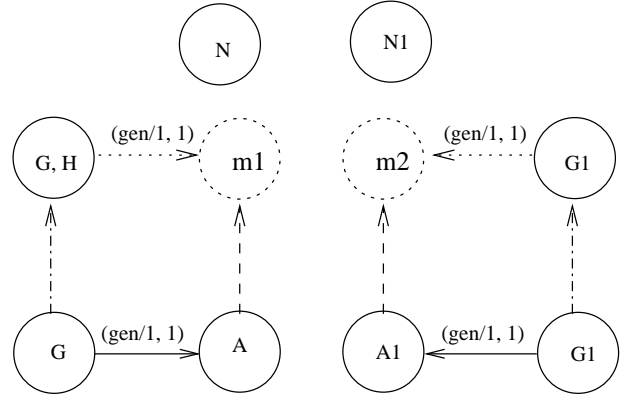
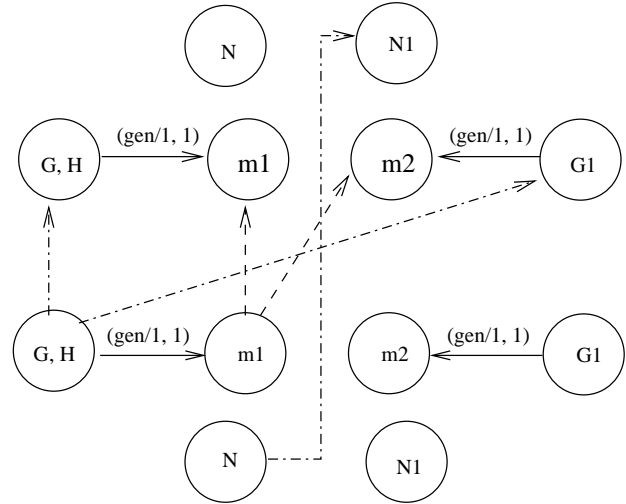
Figure 15: The points-to graph of life after interprocedural analysis at program point (4). The new nodes, edges, and α mappings are due to rule R7.

Figure 16: The points-to graph of life after interprocedural analysis at program point (5).

- $inputR(nextgen) = \{n_G, n_A\}$
- $outputR(nextgen) = \{n_{G1}, n_{A1}\}$
- $deadR(nextgen) = \{n_G, n_A\}$
- $bornR(nextgen) = \{n_{G1}, n_{A1}\}$
- $localR(nextgen) = \phi$

The results for live variable and live region analyses:

pp	LV	LR
(1)	{G}	{n _G , n _A }
(2)	{A}	{n _A }
(3)	{A1}	{n _{A1} }
out	{G1}	{n _{G1} , n _{A1} }

No rules applied and the sets of dead and born regions are unchanged.

Live region analysis for life:

Execution paths:

- (1) (2) out
- (3) (4) (5) out.

At the beginning, the sets of regions of life are:

- $inputR(life) = \{n_N, n_G, m1\}$
- $outputR(life) = \{n_G, m1\}$
- $deadR(life) = \{n_N\}$
- $bornR(life) = \phi$
- $localR(life) = \{n_{N1}, n_{G1}, m2\}$

The results for live variable and live region analyses for each execution path:

pp	LV	LR
(1)	{N, G}	{n _N , n _G , m1}
(2)	{G}	{n _G , m1}
out	{H}	{n _G , m1}
pp	LV	LR
(3)	{N, G}	{n _N , n _G , m1}
(4)	{G, N1}	{n _G , m1, n _{N1} }
(5)	{N1, G1}	{n _{N1} , n _{G1} , m2}
out	{H}	{n _G , m1}

No rules applied.

6.3.3 Transformation

For nextgen:

- (1): add remove r_G after (1) (T4).
- (2): add create r_{A1} before (2) (T2), remove r_A after (2) (T4).
- (3): add create r_{G1} before (4) (T2).

For life:

- (1): add remove r_N after (1) (T4).
- (3): add create r_{N1} before (3) (T2), remove r_N after (3) (T4).
- (5): rename n_{G1} to n_H and $m2$ to $m1$, i.e. the call to *life* here puts H into the same regions of G1. The transformed program is as follows:

```
nextgen(G, G1) :-
(1) G => gen(A),
    remove rG,
    create rA1,
(2) A1 is A + 2,
    remove rA,
    create rG1,
(3) G1 <= gen(A1).
```

```
life(N, G, H) :-
(
(1) N => 0,
    remove rN
- >
(2) H := G
;
create rN1,
(3) N1 is N - 1,
    remove rN,
(4) nextgen(G, G1),
(5) life(N1, G1, H)
).
```

7 Future extensions of the region analysis

The extensions of the above algorithm to support non-determinism, module-based analysis, and combination with compile-time garbage collection (CTGC) [4] can be developed tentatively as follows.

7.1 Non-determinism

To support non-determinism we will need to prevent the regions that will be used when backtrack-

ing from being removed. The sets of live regions at program points are derived from the sets of live variables. Therefore if we can compute the set of variables that are live at a program point when the program backtracks to that program point we will be able to compute the set of regions needed when backtracking. At the first sight, that set of variables can be approximated by the backward use with 2^{nd} instantiation as defined in [4] and the other parts of the algorithm can be intact. A variable X is said to be in local backward use (lbu) w.r.t. a program point (i) within a procedure definition if that variable is instantiated at (i) and can be accessed by the literals of the procedure after backtracking has reentered the code prior to (i). The new set of live variables at a program point will be defined as follows.

$$LV_2(i) = LV(i) \cup lbu(i).$$

7.2 Module-based region analysis

Mercury programs are composed of several modules. We assume that there are no circular calls among the modules so that when analysing a Mercury program each module can be analysed once. So when we are analysing a module all the procedures that are called from the module have been analysed already. If we allow a caller to have control on region-related behaviour of a callee then all the regions in *deadR* set of the callee can always be removed and all regions in *bornR* set can always be created by itself. We can change the analysis by dropping the rule *L1* and *L2* in live region analysis and enhancing the transformation to introduce “keep” and “use” statements with the following two additional rules.

T5: If a region, r , is supposed to be removed by the called procedure but it is still live after the call then the calling procedure will add “keep r ” before the call.

The effect of “keep r ” is that it will prevent the called procedure from really removing r .

T6: If a region, r , is supposed to be created by the called procedure but it has been created by the calling one then the calling procedure will add “use r ” before the call.

The effect of “use r ” is that the called procedure will use the region r instead of creating it.

This means that the region removal and creation

are now conditional. Having this conditional removal and creation makes the module-based region analysis rather simple. An analysed procedure will be identified by *deadR*, *bornR*, and the region points-to graph (nodes, edges, renaming, alpha mapping). In the region points-to analysis of a procedure (caller) that calls the analysed one (callee), the region points-to graph of the callee will be used to produce the caller’s points-to graph. The live region analysis takes place just to calculate the live region information at each program point of the caller. The transformation will use the two additional rules to introduce “keep” and “use” statements to make the behaviour of the callee suit the requirements at the calling context. With this organisation we will have to generate only one optimized version of a procedure with region support but any callers can control its real behaviour to meet their own needs. The drawback of this approach is that checking the conditions can be a burden at runtime.

7.3 Combination with CTGC

7.3.1 The approach

CTGC tries to reuse dead memory, while RBMM tries to remove the regions that contain them. So if CTGC decides that the cells of X are reused to construct Y then in the region points-to analysis we should merge the regions of X and Y , i.e. unify n_X and n_Y . In the definition of the current region points-to analysis, unifying two nodes can cause other nodes to be merged so we need to prevent those unwanted mergings of regions (which unnecessarily limit the chance of removal). More insights need to be developed to make this complete, but it is likely that by preventing any R rules to be triggered when “unifying two nodes because of reuse” in region points-to analysis and the current definition of *Reach(i)* using valid type selectors in live region analysis can support the combination.

7.3.2 A motivational example

A motivational example of combining RBMM and CTGC is shown below. The *convert* program is taken from Chapter 11 in [4] with some modifications just to make the explanation shorter. *convert* is declared with mode (in, out) and transforms a list

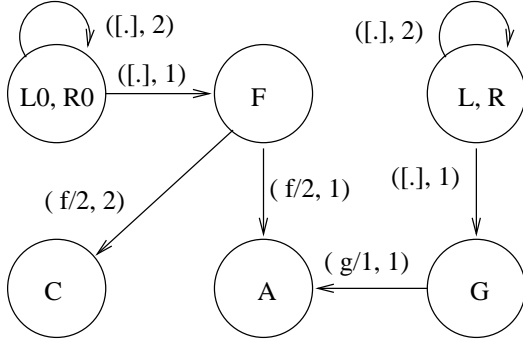


Figure 17: The points-to graph of *convert* after region points-to analysis.

of type *f* to a list of type *g*.
 $\text{:- type } f \text{ --- } > f(\text{int}, \text{int}).$
 $\text{:- type } g \text{ --- } > g(\text{int}).$

$\text{convert}(L0, L) \text{ :-}$

```
(
(1) L0 ==> [],
->
(2) L <= []
;
(3) L0 ==> [F | R0],
(4) F ==> f(A, C),
(5) G <= g(A),
(6) convert(R0, R),
(7) L <= [G | R]
).
```

After region points-to analysis, the graph of *convert* is shown in Figure 17.

The example will show the extended analysis with two different reuse decisions:

- reuse the cells of $L0$ to construct L and the cells of F to construct G ,
- only reuse the cells of $L0$ to construct L for the reason that either the backend does not allow reusing cells with different types or we subjectively prohibit the reuse of non-matching arity because as reported in [4] the setting with reuse of matching arity gives better result than the other ones do (assume the analysis prefers reusing $L0$ to construct L to reusing F).

1. Case 1: $L0$ for L and F for G :

We will need to unify n_{L0} and n_L , n_F and

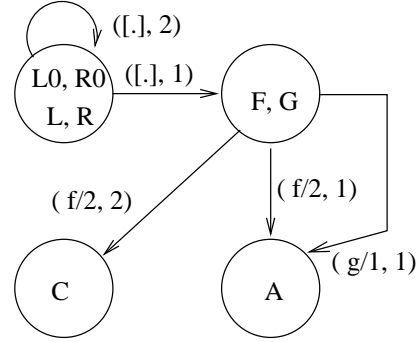


Figure 18: The points-to graph of *convert* after unifying $n_{\{L0,R0\}}$ and $n_{\{L,R\}}$, n_F and n_G .

pp	LV	LR
(1)	{L0}	{ $n_{\{L0,R0,L,R\}}$, $n_{\{F,G\}}$, n_A , n_C }
(2)	{}	{}
out	{L}	{ $n_{\{L0,R0,L,R\}}$, $n_{\{F,G\}}$, n_A }
pp	LV	LR
(3)	{L0}	{ $n_{\{L0,R0,L,R\}}$, $n_{\{F,G\}}$, n_A , n_C }
(4)	{F, R0}	{ $n_{\{L0,R0,L,R\}}$, $n_{\{F,G\}}$, n_A , n_C }
(5)	{R0, A, C}	{ $n_{\{L0,R0,L,R\}}$, $n_{\{F,G\}}$, n_A , n_C }
(6)	{R0, G}	{ $n_{\{L0,R0,L,R\}}$, $n_{\{F,G\}}$, n_A , n_C }
(7)	{G, R}	{ $n_{\{L0,R0,L,R\}}$, $n_{\{F,G\}}$, n_A }
out	{L}	{ $n_{\{L0,R0,L,R\}}$, $n_{\{F,G\}}$, n_A }

Figure 19: Live region analysis results of *convert* in Case 1.

n_G , no rules will be applied after these actions. The points-to graph after this is shown in Figure 18.

Live region analysis:

Execution paths: (1) (2) out,
 (3) (4) (5) (6) (7) out.

At the beginning, the sets of regions of *convert* are:

- $\text{input}R(\text{convert}) = \{n_{\{L0,R0,L,R\}}, n_{\{F,G\}}, n_A, n_C\}$
- $\text{output}R(\text{convert}) = \{n_{\{L0,R0,L,R\}}, n_{\{F,G\}}, n_A\}$
- $\text{dead}R(\text{convert}) = \{n_C\}$
- $\text{born}R(\text{convert}) = \phi$
- $\text{local}R(\text{convert}) = \phi$

The results for live variable and live region analyses is shown in Figure 19. No rules

applied and the sets of dead and born regions are unchanged. Note that at “out” the live variable set contains L of type $\text{list}(\text{g}(\text{int}))$ therefore n_C is not reachable from L because $([., 1]) \bullet (f/2, 2)$ is not a valid type selector of L . The same situation happens at (7), where the live variable set contains G of which $(f/2, 2)$ is not a valid type selector, and R of which $([., 1]) \bullet (f/2, 2)$ is not a valid type selector and the node n_C is not reachable.

Transformation:

At (1), rule T4 applied and the transformed program is as follows.

$\text{convert}(L0, L) :-$

```
(
  (1) L0 => [],
  remove r_C,
  - >
  (2) L <= []
;
  (3) L0 => [F | R0],
  (4) F => f(A, C),
  (5) G <= g(A),
  (6) convert(R0, R),
  (7) L <= [G | R]
).
```

We see that the transformed program can deallocate the memory for C, which is memory leak in CTGC. The memory cells of the list backbone and of part of the list’s elements are reused by CTGC.

2. Case 2: Only reuse $L0$ for L :

We will unify only n_{L0} and n_L , no rules will be applied after this action. The points-to graph after this is shown in Figure 20.

Live region analysis:

Execution paths: (1) (2) out,

(3) (4) (5) (6) (7) out.

At the beginning, the sets of regions of convert are:

$$\begin{aligned}
 \bullet \text{inputR}(\text{convert}) &= \{n_{\{L0, R0, L, R\}}, n_F, n_A, n_C\} \\
 \bullet \text{outputR}(\text{convert}) &= \{n_{\{L0, R0, L, R\}}, n_G, n_A\} \\
 \bullet \text{deadR}(\text{convert}) &= \{n_F, n_C\} \\
 \bullet \text{bornR}(\text{convert}) &= \{n_G\}
 \end{aligned}$$

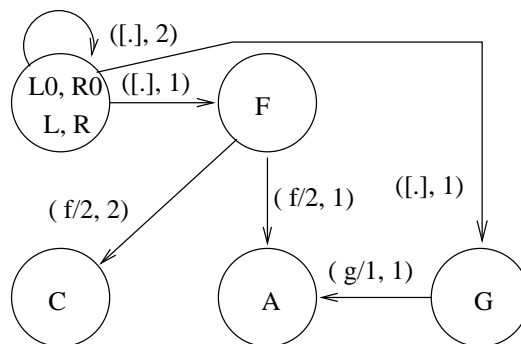


Figure 20: The points-to graph of convert after unifying $n_{\{L0, R0\}}$ and $n_{\{L, R\}}$.

pp	LV	LR
(1)	{L0}	{ $n_{\{L0, R0, L, R\}}$, n_F, n_A, n_C }
(2)	{}	{}
out	{L}	{ $n_{\{L0, R0, L, R\}}$, n_G, n_A }

pp	LV	LR
(3)	{L0}	{ $n_{\{L0, R0, L, R\}}$, n_F, n_A, n_C }
(4)	{F, R0}	{ $n_{\{L0, R0, L, R\}}$, n_F, n_A, n_C }
(5)	{R0, A, C}	{ $n_{\{L0, R0, L, R\}}$, n_F, n_A, n_C }
(6)	{R0, G}	{ $n_{\{L0, R0, L, R\}}$, n_F, n_A, n_C, n_G }
(7)	{G, R}	{ $n_{\{L0, R0, L, R\}}$, n_G, n_A }
out	{L}	{ $n_{\{L0, R0, L, R\}}$, n_G, n_A }

Figure 21: Live region analysis results of convert in Case 2.

$$\bullet \text{localR}(\text{convert}) = \phi$$

The results for live variable and live region analyses are shown in Figure 21.

At (6): rule L2 applied. So $\text{bornR}(\text{convert}) = \phi$, which means that the region n_G needs to be created by the procedure that calls convert . Look in detail at program point “out”, the live variable set contains L of type $\text{list}(\text{g}(\text{int}))$. n_C is not reachable from L because $([., 1]) \bullet (f/2, 2)$ is not a valid type selector of L . To make n_F not reachable from L one solution is to keep the two edges with the same label $([., 1])$ separated when unifying $n_{\{L0, R0\}}$ and $n_{\{L, R\}}$. That is, the points-to graph will contain two separated edges both starting from $n_{\{L0, R0, L, R\}}$: $(n_{\{L0, R0\}}, ([., 1]), n_F)$ and $(n_{\{L, R\}}, ([., 1]), n_G)$, but going to different nodes. From L we should only follow the later edge, not the former one (because $L \in \{L, R\}$).

Therefore n_G is reachable, but n_F is not. The same situation happens at (7), where the live variable set contains G of type $g(\text{int})$ and R of type $\text{list}(g(\text{int}))$ and the nodes n_C and n_F are not reachable.

Transformation:

At (1), rule T4 applied so r_F and r_C are removed.

Note that at the program point “out” in the execution path (1), no “create r_G ” is added, even according to the live region analysis it becomes live at this point. This is because at (2), L is constructed, not G . The live region analysis is not precise here but the transformation still ensures the correct solution. It is probably possible to enhance the live region analysis if we are able to distinguish the type constructors of R used in each execution path. In the execution path (1), L is constructed by \square , so n_G and n_A are not live at “out” in path (1).

The transformed program is as follows.

```

convert(L0, L) :-
(
  (1) L0 =>  $\square$ ,
    remove  $r_F$ ,
    remove  $r_C$ ,
  - >
  (2) L <=  $\square$ 
;
  (3) L0 => [F | R0],
  (4) F => f(A, C),
  (5) G <= g(A),
  (6) convert(R0, R),
  (7) L <= [G | R]
).

```

Here the transformed program will reuse the memory of the input list backbone to create the backbone of the output list and be able to deallocate the memory cells used for F and C , which otherwise are memory leaks.

7.3.3 Potential benefits of combining RBMM and CTGC

Without combination RBMM alone as described above can already be interesting for several programs, such as *qsort*, where RBMM helps the pro-

gram run with no more memory than what is required to store the whole original list. With the transformation based on region liveness information described above, the lifetime of regions in the output program is shorter when comparing with the results achieved by the analyses of Henning (in the context of functional programming - subset of SML and logic programming - XSB Prolog) and Sigmund (in the context of OOP - Java), which should cause less memory consumption. It is worth pointing out that our algorithm combines the good parts of [2] and [1]. The good points of [2] are the ability to create and remove regions across procedure borders. The support of renaming regions helps increase the precision of region points-to analysis. The advantages of [1] are the powerfulness but simplicity of the analysis and transformation, which should lead to simple correctness proof. This is not at all the case in [2] and [3], where not only is the correctness of the region type system hard to be proved but also the inference algorithm and its soundness proof are not trivial.

The possibility of benefits of the combination of RBMM and CTGC is when there are procedures contain cells that die but cannot be reused locally by CTGC. This is the case when some cells die in a procedure and the procedure has no construction allowed to reuse them (due to there is nothing to reuse for (Case 1), reuse decision or back-end constraints (Case 2),...). In [4] some of the cells that die unconditionally can be reused using cell cache (when the size fits). For the benchmark Ray Tracer, cell cache can increase the relative reduction from $\sim 25\%$ to $\sim 50\%$, which means that, at least for this medium program, there are quite many cells put into the cache and reused later on. Using cell cache has its own cost, such as maintaining the cache and checking the cache before any allocation, and may also harm locality. With the region analysis above the unconditionally died cells will be put into separate regions that can be reclaimed therefore we may gain over CTGC alone:

- The cost of maintaining and using the cache,
- The cells that are put into the cache but not reused. This information is not collected in [4] for the Ray Tracer program. But if this number is, on average, significant then it can be a definite advantage of RBMM.

- "Locality". This gain is not conclusive for RBMM. The experimental results from current RBMM systems do not always support good locality and faster code. This is reported as a more experimental than theoretical research problem [5].

References

- [1] S. Cherem and R. Rugina. Region analysis and Transformation for Java. In *Proceedings of the 4th international symposium on Memory management*, pages 85–96. ACM Press., October 2004.
- [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] Mazur N. *Compile-time garbage collection for the declarative language Mercury*. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, May 2004.
- [5] M. Tofte, L. Birkedal, M. Elsmann, and N. Haltenberg. A Retrospective on Region-Based Memory Management. In *Higher-Order and Symbolic Computation*, 17, pages 245–265. Kluwer Academic., 2004.