Region-based memory management for the logic programming language Mercury

Quan Phan

November 2009

Abstract

Region-based memory management (RBMM) is a form of compile-time memory management, well-known from the functional programming world. In this thesis we describe our work of investigating and developing RBMM for the logic programming language Mercury. Mercury is designed with strong type, mode, and determinism systems. These systems not only provide Mercury programmers with several direct software engineering benefits, such as self-documenting code and clear program logic, but also give the language developers useful information for optimizations.

The first challenge in realizing RBMM for a programming language is to divide program data into regions such that the regions can be reclaimed as soon as possible. In the thesis we have developed program analyses that determine the distribution of data over regions as well as their lifetimes in a Mercury program. The program then is transformed to a region-annotated program containing the necessary region constructs. We provide the correctness proofs of the related program analyses and transformation, which guarantee the safeness of memory accesses in annotated programs.

We have implemented runtime support that tackles the special challenge posed by backtracking. Backtracking can require regions removed during forward execution to be "resurrected", and any memory allocated during a computation that has been backtracked over must be recovered promptly, without waiting for the regions involved to come to the end of their life.

We study the effects of RBMM for a selection of benchmark programs including well-known difficult cases for RBMM. Our RBMM-enabled Mercury system obtains clearly faster runtime for two third of the benchmarks compared to the Mercury system using the Boehm runtime garbage collector, with an average speedup of 20%. In terms of memory usage, the region system achieves optimal memory

consumption in some programs. Our in-depth case study reveals the impact of sharing on memory reuse in programs in region-based systems.

Finally, we propose region reuse extension to our current region framework. Our RBMM system augmented with region reuse can automatically improve memory reuse for a popular class of programs that otherwise often require non-intuitive manual program-rewriting so that RBMM systems can obtain good memory reuse.

Contents

List of Tables

List of Figures

Acknowledgements

This dissertation is the final work that completes my doctorate education in the research group "Declarative Languages and Artificial Intelligence" (DTAI) at Katholieke Universiteit Leuven. It has been a long time, more than five years, since I was given a chance to try the domain of research, in particular research on programming languages, which I am always interested in. I think to be able to work with what we like is a lucky thing that does not happen to every of us. Despite working with the "hobby", it soon turned out to be a real challenge. I am very happy that I can finish the journey with a concrete achievement, namely being awarded the PhD degree in Computer Science.

At this special moment I would like to sincerely thank the professors in my research group for accepting me to DTAI. In particular I would like to thank Professor Maurice Bruynooghe, who has been leading DTAI since its first days. After around 20 years, the founding leader is still active than ever in every aspect of DTAI, making it a strong research group of more than 50 members. Maurice leads the group by examples with his dedication, consistency, and discipline in research, kindness, calmness and enjoyment in life. I believe that two things are needed for the durable existence of any organization: its good culture and its strong source of finance. I am very grateful to the professors for their great effort to ensure DTAI having both, making doing research in the group a privilege.

During my doctoral education, I worked under the supervision of my promoter, Professor Gerda Janssens. Supervision is actually not at all the right word for describing what she has done for me. I think it must be a good combination of help, guidance, caring, and protection. The very nice experience was that she always made me feel comfortably free with my work and my time. Gerda has always been available, showing the support and encouragement, when I needed such things the most. I am very grateful to her for everything she has brought to me. I also knew that helping me with a research topic that is not right in her expertise was not always comfortable for her. It required extra effort and determination. To this end I would like to express my great appreciation for her enduring interest, hard work and patience.

Apart from Gerda who has worked closely with me, I would like to thank the

other professors of my doctoral Examination Board for reading and commenting the dissertation's text: chairman Peter Van Houtte, two external members Kostis Sagonas and Zoltan Somogyi, Yolande Berbers from Distrinet research group, and two members from DTAI who are also my doctoral advisors Bart Demoen and Maurice Bruynooghe.

One of the decisive factors for the completion of my PhD is the implementation of my theory in the Mercury compiler. The implementation we have today would not exist without the help of the Mercury development team in the University of Melbourne, Australia. I had a very fruitful, fun and memorable 3-month stay there. My special thanks go to Professor Zoltan Somogyi who showed tremendous interest in my research. I enjoyed so much working with and learning from him during that time. I thank him for his enthusiasm for improving my first paper and for being my co-author of another one. I also thank Julien Fischer for his technical support during the development of my system. This was the time when I got to know the Mercury compiler, which is a complex system with many parts that are not necessarily compatible. Without their valuable help I would not have a system up and running during my time there. I also would like to extend my grateful thanks to the rest of the Mercury team at that time Rafe and Mark; the Mission Critical people Pete, Ian and Peter. It was much fun to share the office with you all, enjoyed the coffee culture in Melbourne every afternoon and the pubs every Friday. You made my stay in Melbourne an enjoyable and warm experience.

I thank my past and present colleagues in DTAI and in the Computer Science department in general. In particular I would like to thank Denise Brams and Karin Michiels for taking care of all the necessary routines and the system administrators including the DTAI representatives Jon Sneyers, Kurt De Grave, and Pieter Wuille for ensuring a smooth working environment. All my research activities are funded by DTAI, which, as far as I know, is then mainly funded by the Flemish government, I would like to thank them for the generous financial support.

During my time in Melbourne, I stayed at the Graduate House where I met several friendly graduate students from different parts of the world. I would like to thank them for creating such a warm, open, and fun atmosphere. Leslie also visited Melbourne and stayed in the House for a month during that time. It is fair to say that the experience there had caused certain mental changes to both of us. (Well, who don't at the Down Under? :-).) I thank him for spending time on discovering the spread out Melbourne with me.

I would like to thank Theofrastos for a pleasant time to ICLP 2009 in Pasadena. The trip would never be the same without you. Our two-people team was never competitive enough to challenge the first place in the Prolog contest but it was fun. That disappointment, if any, is easy to forget but the almost-12-hour shopping spree in that mall will never be forgotten ;-). The Prolog contest this year was very well organized by Tom Schrijvers and his wife. I am thankful for their arrangement. A warm thank goes to Dean, Hanne, Jon, Paolo, and Peter for a nice socializing

time during the same trip. I also very much appreciated Hanne for showing me that sushi can really be delicious.

Hanne is actually one of my two office mates for the last couple of years. She is energetic, passionate and open about many things. The other used to be a Danish girl, Mai, who joined us for almost a year then left prematurely due to personal reasons. I really felt for her and would wish her all the best in the future. Now that other is Abi from India, who has just started his PhD journey. It seems that the next one to leave the office will be me so I would like to thank them for the time together and wish them great success in work as well as in life.

I feel something missing if I do not mention my compatriot friend Nguyen Manh Thang who died very unluckily in a car accident just around a week after the preliminary defense of his PhD under the guidance of Professor Danny De Schreye. We had been together since the Master program in Artificial Intelligence in 2003-04. October 2004 we started our PhD education in DTAI, sitting next to each other in one side of our first office. I feel very sorry that now he has gone forever. All I can do is to keep the fond memories of him, a sincere friend, a dedicated husband and father, and a responsible Vietnamese citizen. Rest in peace, my dear friend! I would like to thank everyone who has shared this sadness with me. In particular, I am grateful to Jacques Riche and Paula Bruggen for their sympathy. They used to come infrequently to our office. Jacques came once a week while Paula only showed up in the evening. They both showed warm openness to our culture. I met Jacques more often. I thank him especially for his caring and willingness to share his vast knowledge and experience.

I would like to thank my friends for their support during this long period. Closest to me in Leuven are friends in VinaKUL, the group of Vietnamese in Leuven and in particular the footballers with whom I had so much enjoyment. Thank you very much, I am proud to be part of you.

The final thanks are to my family and relatives. The older I am the more I realize that I cannot thank my parents enough for their understanding and unconditional love. To my aunts and uncles, sisters and brothers, nieces and nephews, thank you for the warm moments we had together. I thank my parents-in-law for giving me their youngest daughter. Without her, I am sure not who I am today. To my beloved wife, Lan, thank you for the passionate love, patience and also the lack of it, which have opened up my eyes to many interesting perspectives.

Chapter 1

Introduction

1.1 Programming Languages and Memory Management

Programming languages are created to express computations that can be performed by a computer. The computations are often described by algorithms that manipulate data. When such algorithms are executed on a computer, the data need to be stored in the computer's memory. This leads to the necessity of *memory allocation*, namely assigning locations in memory to store some piece of data. In the very early days of programming languages, programmers had to manually take care of mapping their data onto physical addresses in the memory. Later on, automatic memory allocation has been implemented in almost all programming systems (Wilson, Johnstone, Neely, and Boles 1995). Also algorithms often have high memory complexity, producing a lot of temporary data that are necessary during the computation but not in their effective results. In general, new memory is allocated to store new data. But the computer memory is finite, hence at some point there will be a demand to reuse part of the allocated memory whose contents are no longer used. Without *memory reuse* not only would out-of-memory be an imminent problem, but also wasteful use of memory can lead to bad performance due to the burden of memory accesses. Having a convenient and efficient mechanism of memory management, i.e., allocation and reuse, thus plays a vital role in the popularity and success of a programming language system.

Manual memory management, such as the C-like `malloc/free`, leaves most of the responsibility to programmers. The programmers explicitly write code to ask for the exact amount of memory needed. More problematically, they themselves need to find the point where the allocated memory can safely be released. This decision is nontrivial considering the complexity of software systems nowadays, causing all sorts of problems. Most prominently are memory leaks when some

part of memory is reclaimed too late and mysterious runtime crashes when still-used data are released too soon. Moreover, no matter how good the programmers are, there cannot be guarantees for the soundness of memory accesses.

Overcoming the tedious and error-prone nature of manual deallocation became a requirement for the development of increasingly larger computer applications. This necessity along with the development of high-level programming languages led to the fruitful research on *automatic* garbage collection techniques (Wilson 1995). The term *garbage collection* dates back to the development of the high-level declarative programming language LISP (McCarthy 1960). A garbage collector is a computer program whose function is to, during the runtime of the main program, detect data that are no longer in use, called *garbage*, and make their storage available for reuse by the main program. Runtime garbage collection has gradually been adopted as an integral component of modern programming language systems. Garbage collection algorithms guarantee memory safety, which is essential for large-scale application development. The state-of-the-art implementations of garbage collectors can provide programmers with good memory use and competitive performance compared to the manual practice. In spite of all the proven advantages, one noticeable drawback of runtime garbage collection is that the task of detecting garbage is done completely at runtime, which can incur significant overheads. Runtime garbage collection is also known to have adverse effects on locality of reference, affecting the overall performance of programs. These problems can be alleviated using more sophisticated techniques such as incremental or generational garbage collection but in general to have a state-of-the-art implementation of a garbage collector requires many difficult decisions on design and strategy.

One idea to reduce the incurred overheads is to shift some of the collection overheads from runtime to compile-time. *Compile-time* garbage collection is a technique following this approach. The idea is that, when compiling a program, we use program analyses to decide when a data object becomes dead during the execution of the program. After the point the object is no longer accessed in the remaining execution and its storage can be reused for new allocation by in-place updating. Compile-time garbage collection has long been investigated with the potential of reducing the cost of runtime garbage collection because in the presence of in-place updating the collection process would need to be triggered less often during the run of a program. In the context of declarative programming languages, due to the absence of imperative constructs such as assignments, updating a field in a data structure requires reconstruction of the structure itself. Therefore the research on compile-time garbage collection here focuses on in-place updating in popular aggregate data structures such as list and array (Hudak and Bloss 1985; Sastry, Clinger, and Ariola 1993; Jones and Le Métayer 1989; Mazur 2004). For object-oriented languages, the emphasis is to reclaim individual objects (Guyer, McKinley, and Frampton 2006; Cherem and Rugina 2006). Programming systems

in which a runtime garbage collector is assisted by compile-time garbage collection often have a smaller memory footprint and improved runtime performance.

Region-based memory management (RBMM) (Tofte and Talpin 1997) is another compile-time technique. It is based on the idea of grouping data objects having the same lifetime into regions, the motive being that reclaiming entire regions at the end of their lifetime makes collection very fast. A typical use of regions is to store temporary data that hopefully can be reclaimed in time. So the main challenge in region-based memory management is to find an automatic way to divide program data into regions such that the regions have as short as possible lifetimes so that we can reuse memory as much as possible while the overheads of using regions need to be small enough so that the gain due to no runtime garbage collection can actually lead to execution speedups for programs. The program analysis that is charged with this responsibility is usually referred to as region inference or region analysis. There is an additional and unique challenge when applying RBMM to logic programming due to the existence of nondeterministic execution in terms of backtracking. Supporting backtracking in RBMM poses several issues to both the region analysis and the engineering of the runtime system.

In this thesis we investigate the application of region-based memory management to the modern logic programming language Mercury.

1.2 Motivational Example

We give a simple and intuitive example showing the idea and benefits of RBMM. Through the example we reveal the challenges and issues of realizing RBMM in a programming language system.

Take, for example, a drawing program in which we have an operation that produces a highlighted triangle from a plain one. An example of a plain triangle and a highlighted version of it is shown in Figure 1.1.

A plain triangle is represented by a triple of points, each in turn is encoded by horizontal and vertical coordinates. In the triangle ABC in Figure 1.1, A is at $(1, 1)$, B is at $(1, 5)$, and C is at $(4, 1)$. We use the following term to represent it,

```
plain_triangle(point(1,1), point(1,5), point(4,1)).
```

A highlighted triangle has some additional features, such as a thickness for its edges, a pattern for its surface, and two edges highlighted with their own colors. The highlighted triangle shown in Figure 1.1 has a thickness of two; the pattern on its surface is parallel lines raising to the right; and it has an edge in red and one in green. To represent highlighted triangles we use a data structure different from that of plain ones. For the highlighted ABC in Figure 1.1 we represent it by the following term:

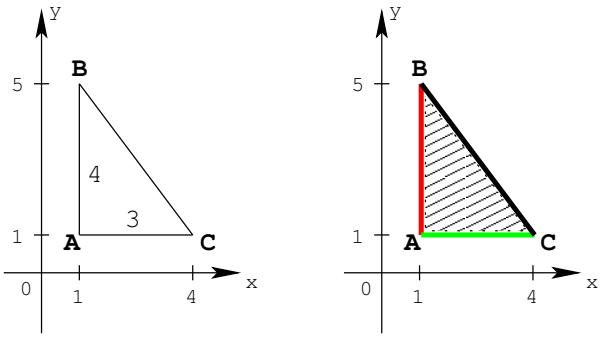


Figure 1.1: A plain triangle and one of its highlighted versions.

```
highlight_triangle(PlainTriangle, HighlightedTriangle) :-
  PlainTriangle = plain_triangle(A, B, C),
  % (*)  code to compute the Lengths and Angles of the two edges.
  ...,
  % (**) code to pick a Thickness and a Pattern for the triangle,
  % and two Colors for the two edges.
  ...,
  AB = edge(A, Length1, Angle1, Color1),
  AC = edge(A, Length2, Angle2, Color2),
  HighlightedTriangle = highlighted_triangle(AB, AC, Thickness, Pattern).
```

Figure 1.2: Code that highlights a plain triangle.

```
highlighted_triangle(
    edge(point(1,1), 4, 90, red),
    edge(point(1,1), 3, 0, green),
    2, 1).
```

The first two components represent the two highlighted edges. The position of an edge is defined by a point, the length of the edge, and an angle from the x-axis. For example, AB starts from A, has length of four, and is 90 degree away from the x-axis. The third component is the thickness and the last one encodes the pattern. Our highlighted ABC has a thickness of two and here we simply assume its pattern is encoded by one.

The desired operation can be implemented as in Figure 1.2. We ignore the code at (*) and (**) because it is not significant to the issue we are considering. The in-memory representations of these terms are in Figure 1.3. Each box visualizes a memory word, a basic storage unit. We call a block of contiguous words a memory

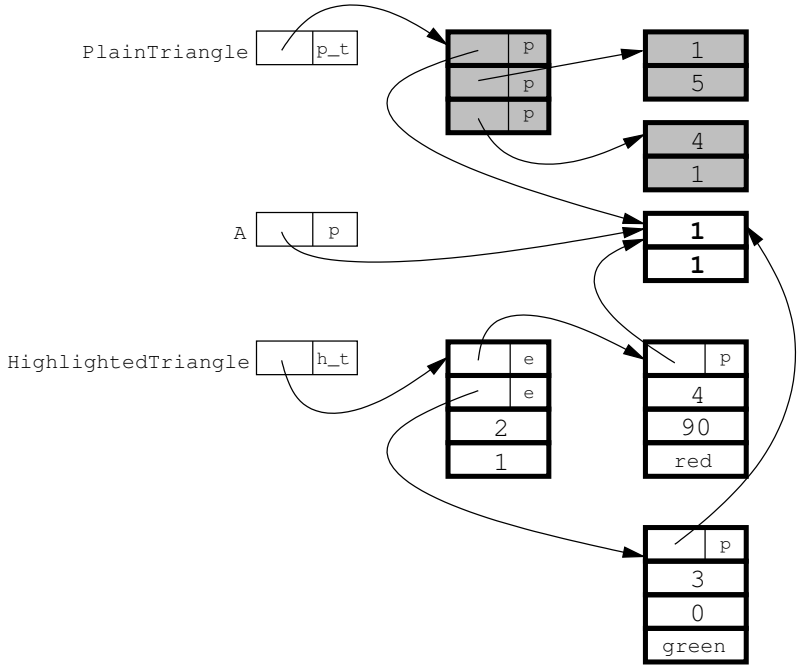


Figure 1.3: In-memory representations. A box containing a pointer, which is drawn as an arrow, is *tagged* to show the type of the data structure that the box points to.

cell. For example, `PlainTriangle` points to a memory cell of three words. We also use a tag, the right part of some box, to denote the type of data structure that the box points to, such as `p_t` for `plain_triangle`, `p` for `point` and so on. The boxes in bold mean the words on the heap, the other are words on stacks or are registers. The cells reached from `PlainTriangle` are in memory before the run of the triangle-highlighting operation in Figure 1.2. Out of these, ones with shaded background become *garbage* after the operation and the two-word cell pointed to by `A` is *shared* between the plain triangle and the resulting, highlighted one. We say that the cell is reused. The three other four-word heap cells reachable from `HighlightedTriangle` are newly allocated in the operation, at the last three lines.

If we can somehow arrange that the garbage cells are allocated into a *region* and the cells reachable from `HighlightedTriangle` are put into *different* regions, we will be able to reclaim the garbage cells by releasing their corresponding region, right after the highlighting operation. Note that, in this example, in-place updating as in compile-time garbage collection where a garbage cell is reused for a new allocation will not help because the sizes of the new cells are larger than those of the garbage ones.

To realize this idea, we have to accomplish several tasks.

- We can see, for example, that the term representing the plain triangle needs to be stored in two different regions, one for the garbage cells and the other for the two-word, reused cell. This poses the task of deciding which data are allocated into which regions. The usage context of data is a decisive factor in this decision. If we limit the scope to just the highlighting operation in our example, we would reasonably think that all the cells reached from `HighlightedTriangle` should be in one region. But if, at some other part of the program, only one of the two edges of the highlighted triangle is needed and the other becomes garbage then we may want them to be in separate regions so that the garbage can be reclaimed. A suited distribution of data over regions is needed to have satisfactory memory reuse.

- We need to locate the points in a program where the regions have to be created and can safely be destroyed. This means we have to estimate the lifetimes of regions. With the aim to reuse memory we would want to give regions as short as possible lifetimes. We want to have a fully automatic system. Therefore this and the above task have to be done at compile-time. The tools for that are program analyses that allow us to learn about properties of programs before running them.

- Support for regions and region operations needs to be implemented in the runtime system of the programming language. Efficient implementation plays a significant role in the overall runtime performance of programs in the region-based system.

- The existence of backtracking in logic programming requires taking backward executions into account, usually leading to more complicated program analyses and more responsibilities for the runtime support.

Grouping data objects into regions then releasing them all at once has several advantages.

- Reclaiming the whole region is very efficient because there is no overhead of detecting garbage at runtime.
- The latency between when a memory cell becomes dead and when it is actually reclaimed can often be small. This certainly depends on whether we can give regions short enough lifetimes.
- It may also offer good data locality because related data are put together.

We choose to study RBMM in the context of Mercury amid the limited research about RBMM for logic programming. Since the fundamental work about RBMM for functional programming (specifically SML) (Tofte and Talpin 1997), there have been several improvements and new developments in that context (Aiken, Fähndrich, and Levien 1995; Birkedal, Tofte, and Vejlstrup 1996; Henglein, Makholm, and Niss 2001), and the idea has also been brought to other programming paradigms, such as imperative programming (Gay and Aiken 1998; Grossman, Morrisett, Jim, Hicks, Wang, and Cheney 2002) and object-oriented programming (Cherem and Rugina 2004; Chin, Craciun, Qin, and Rinard 2004). A thorough review of the achievements of this research field can be found in (Tofte, Birkedal, Elsman, and Hallenberg 2004).

For logic programming languages, there have been only attempts to apply RBMM to Prolog in (Makholm 2000b; Makholm 2000a) and in (Makholm and Sagonas 2002). However, the algorithm for RBMM in (Makholm 2000b; Makholm 2000a) was developed for a non-standard implementation of Prolog which would require substantial changes before it could be applied in any standard implementation. In (Makholm and Sagonas 2002) the authors fixed the problem by implementing RBMM in the context of the standard technology for implementing Prolog, the Warren Abstract Machine (WAM) (Aït-Kaci 1999). Nevertheless, this work mainly concentrated on the runtime extensions needed by regions to run Prolog programs with RBMM. It adapted a type-based region analysis originally developed for the functional programming language SML (Henglein, Makholm, and Niss 2001) to work with Prolog. Since Prolog is natively untyped and, more importantly, unmoded, the region inference has to get the needed information from type and mode inferences, which often yields imprecise results. Moreover, that Prolog predicates inherently have no determinism requires them to be treated generally as nondeterministic. These limitations restrict the optimizations that would improve the performance of RBMM, making it hard to become a practical alternative

to native runtime collectors in Prolog systems. The logic programming language Mercury has none of these limitations. It has strong type and mode systems and also the determinism information. This fact, the pure nature of Mercury (the absence of side-effects), and the limited research on RBMM in logic programming motivated us to investigate whether region-based memory management could be developed and implemented efficiently in this context.

1.3 Goals and Contributions

We would like to have a region-based Mercury system in which the memory management task is fully automated. There has been research on explicit RBMM in which programmers manually augment programs with information about regions (Gay and Aiken 1998; Grossman, Morrisett, Jim, Hicks, Wang, and Cheney 2002). Since Mercury is a high-level, declarative language we consider this explicit approach inappropriate for Mercury. Instead, program analyses and transformation will be used to derive the necessary information automatically.

Mercury requires explicit declarations of types, modes, and determinism. This in turn gives us a rich body of information about a program at compile-time. We investigate how these pieces of information can help making the program analyses precise. The precision of these program analyses has a decisive role in obtaining good memory reuse. An important aspect of memory management is its runtime efficiency. We want to investigate whether it is possible to build a region-based system that is competitive with the Mercury system using the state-of-the-art implementation of runtime garbage collection (Boehm and Weiser 1988). One challenge is to support backtracking in the context of regions, without much adverse effects on deterministic programs.

In this thesis we describe the first automated RBMM system for the logic programming language Mercury. For a Mercury program as written by programmers, our system automatically detects the necessary regions and augments each allocation site with a region into which the allocation happens and also inserts instructions to create the regions when needed and to remove them to recover memory when it is safe to do so. The main contributions of the thesis are as follows.

1. We develop the static program analyses for generating the region-annotated programs. These include a region points-to analysis to divide Mercury terms into regions, a liveness analysis that assigns lifetimes for the regions, and a program transformation to annotate the original programs with the derived region information.

2. We prove the correctness of the program analyses in the sense that all memory accesses in the annotated programs are safe.

3. The runtime support for RBMM is designed and implemented to deal correctly with backtracking and without incurring significant overhead.

4. Our RBMM-enabled system achieves faster execution times for several benchmark programs than the Mercury system that uses the Boehm garbage collector (Boehm and Weiser 1988) for memory management. The region system also achieves optimal memory consumption in some benchmarks.

5. We make a detailed analysis of the RBMM behaviour of a selection of programs (including well-known difficult cases). This study reveals the impact of sharing on memory reuse in RBMM systems. We develop a solution to improve memory reuse for a certain class of programs in such difficult situations.

1.4 Structure of the Thesis

In Chapter 2 we introduce Mercury as a programming language and its intermediate form that our work is based on. We also describe intuitively how RBMM can be realized for Mercury and justify our decisions on how to support backtracking.

The model of the heap when it is divided into regions is presented in Chapter 3. We explain the modeling of the locations of terms based on their types by the use of type-based region graphs. The sharing among program variables then is modeled by region points-to graphs.

Based on this region model we develop the static analyses: region points-to analysis, region liveness analysis, and program transformation and their corresponding correctness proofs are given in Chapters 4, 5, and 6, respectively. All in all these analyses and transformation automatically convert a Mercury program written by programmers into a region-annotated program that contains enough region information such as the assignments of regions to allocation sites and when to create and remove regions.

Chapter 7 shows the basic extensions to the runtime system needed to support RBMM in deterministic programs. We show in detail how regions and basic region operations, such as region creation, region removal, allocation into regions are implemented. Chapter 8 describes our handling of backtracking. Backtracking in Mercury involves disjunctions, if-then-elses, and commit operations. Therefore we describe the detailed support for each of these constructs.

The evaluation of our RBMM systems is given in Chapter 9. We look at both memory usage and runtime performance for a selective set of benchmarks, including known difficult programs for RBMM. We discuss the relation between sharing and memory reuse in region-based systems through our detailed study of the runtime behaviour of several programs in our region-based systems. The above study results in the region reuse improvement that is described in Chapter 10.

We discuss the related research activities in Chapter 11 and finally conclude in Chapter 12.

1.5 Bibliographic Notes

Several parts of the technical content in this thesis have been published in proceedings of internationally reviewed conferences. The first version of the material in Chapters 4, 5, and 6 is first published in (Phan and Janssens 2007). Chapters 7 and 8 are based on our publication (Phan, Somogyi, and Janssens 2008). Parts of the experimental results in Chapter 9 were reported in (Phan and Janssens 2007) and (Phan, Somogyi, and Janssens 2008). Finally, the work in Chapter 10 is published as (Phan and Janssens 2009). The list of publications can be found at the end of the thesis.

Chapter 2

Preliminaries

In this chapter we briefly introduce the logic programming language Mercury. The language contains explicit declarations of types, modes and determinism. More important to the technical contents of this thesis is the language's intermediate form that is internal to the Mercury compiler because our program analyses and transformation are defined for it. We then describe intuitively how RBMM can be realized for Mercury and justify our design choices.

2.1 Mercury Language

Mercury is a pure logic programming language intended for the creation of large, fast, reliable programs (Somogyi, Henderson, and Conway 1996). 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.

2.1.1 Predicates

Predicates in Mercury are similar to Prolog predicates. A predicate definition contains one or more clauses. A clause contains a head atom and a body. A body is a goal, which can either be a conjunction of goals, a disjunction of goals, an if-then-else construct, a negated goal, or a literal. A literal is either a unification or an atom.

Example 2.1. The clauses of a predicate that produces the concatenation of two lists by appending one to another are as follows.

```
append(_, Y, Y).
append([Xe | Xs], Y, [Xe | Zs]) :- append(Xs, Y, Zs).
```

□

Mercury treats functions as predicates with the return value as an extra argument, so in the thesis we will only deal explicitly with predicates.

2.1.2 Types

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 as in Example 2.2.

Example 2.2. 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).

2.1.3 Modes

The mode system classifies each argument of each predicate as either input or output; there are exceptions, but we do not explicitly consider them in this thesis. 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.

2.1.4 Determinism

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, including zero.

Example 2.3. The *quicksort* program written in Mercury with declarations of types, modes, and determinisms for two essential predicates, `qsort` and `split`, is shown in Figure 2.1. Some specific elements in `main` are included for completeness but have no importance to the reader's understanding. These include `!IO`, which are state variables representing the states of the world for declarative input/output, and `io.write`, a predicate from the `io` library module, that writes out a term. □

```
main(!IO) :-
    qsort([2, 3, 1], [], S),
    io.write(S, !IO).

:- pred qsort(list_int, list_int, list_int).
:- mode qsort(in, in, out) is det.
qsort([], A, A).
qsort([Le | Ls], A, S) :-
    split(Le, Ls, L1, L2),
    qsort(L2, A, S2),
    qsort(L1, [Le | S2], S).

:- pred split(int, list_int, list_int,
    list_int).
:- mode split(in, in, in, out) is det.
split(_, [], [], []).
split(X, [Le | Ls], L1, L2) :-
( if X >= Le then
    split(X, Ls, L11, L2),
    L1 = [Le | L11]
  else
    split(X, Ls, L1, L21),
    L2 = [Le | L21]
).
```

Figure 2.1: The *quicksort* program in Mercury.

2.1.5 Other Features

In this thesis we deal with the core subset of the language as we describe above. We typically require a program to be written in a single module. Mercury has several other features that we have not yet explicitly supported or considered in this work. In Chapter 12 we discuss some of these features such as module system, library modules, and higher-order programming. A complete description of Mercury can be found in the Mercury language reference (Mercury 2009).

2.2 Mercury Code inside its 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:

$$\begin{array}{lcl} \text{predicate } P & : & p(x_1, \ldots, x_n) \leftarrow G \\ \text{goal } G & : & x = y \mid x = f(y_1, \ldots, y_n) \mid p(x_1, \ldots, x_n) \\ & & (G_1, \cdots, G_n) \mid (G_1; \ldots; G_n) \mid \mathit{not}\ G \mid \\ & & (\mathit{if}\ G_c\ \mathit{then}\ G_t\ \mathit{else}\ G_e) \mid \mathit{some}\ [x_1, \ldots, x_n]\ G \end{array}$$

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

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

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

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

Figure 2.2: *quicksort* program in superhomogeneous form.

mode information by writing `<=` for construction unifications, `=>` for deconstruction unifications, `==` for equality tests, and `:=` for assignments. Based on the mode information, the compiler reorders the subgoals in a conjunction to ensure that ones that consume the value of a variable always come later than the subgoal that produces its value. More details about this abstract syntax can be found in (Somogyi, Henderson, and Conway 1996).

We show the *quicksort* program in this abstract syntax in Figure 2.2. For readability, we have chosen meaningful names for some additional variables that are added automatically by the Mercury compiler. The reader may want to compare this with the program in Figure 2.1, which is the kind of code Mercury programmers would normally write. We also replace the sequence of unifications needed to construct a single ground term with a single goal. For example, the list construction at (1) in `main` in Figure 2.2, actually stands for

```
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]
```

The extra detail is of no interest in this thesis.

In the rest of the thesis, we will ignore negation, since `not` G can be implemented as `if` G `then fail else true`, where `fail` and `true` are two builtin goals, with `fail` always failing while `true` always succeeding. Note that in Mercury (unlike in Prolog), the condition of an if-then-else is allowed to succeed several times. Whether the condition of a particular if-then-else can do so will be recorded in its determinism annotation, and many parts of the compiler, including the RBMM implementation, 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.

2.3 Overview of Region-Based Memory Management in 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 plain 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. We will discuss the necessity of this support in detail in Chapter 8.

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. (We describe the implementation of regions in Chapter 7.) 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

```
main(!IO) :-
    create(R20), create(R21),
(1) L <= [2, 1, 3] in R20,
    create(R22),
(2) A <= [] in R22,
(3) qsort(L@R20, A@R22, S@R22),
(4) io.write(S, !IO),
    remove(R21), remove(R22).

qsort(L@R6, A@R8, S@R8) :-
(
(1) L => [],
    remove(R6),
(2) S := A
;
(3) L => [Le | Ls],
(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)
).
```

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

Figure 2.3: Region-annotated *quicksort* program.

as extra arguments to procedure calls. Figure 2.3 shows the region-annotated *quicksort* program after our region transformation.

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`. 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` (note that the two output lists of `split` start to be constructed in the base case), 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`.

Chapter 3

Region Modelling

Data in logic programs are typically represented by so-called *terms*. Terms often have different lifetimes. In RBMM we want to allocate these terms to different regions for the purpose of reclaiming the terms in one region at once. So the first and perhaps most important task in implementing RBMM is to arrange terms into regions. In this chapter we develop the model of the storage of terms when the heap is composed of regions.

3.1 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.

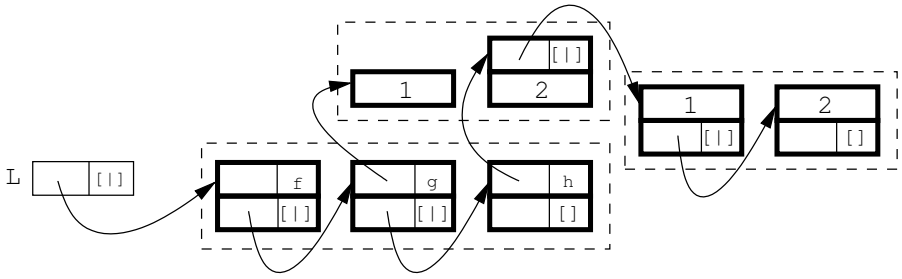


Figure 3.1: Term representation of L=[f, g(1), h([1, 2],2)].

Example 3.1. Consider the following types.
`:- type elem ---> f; g(int); h(list_int, int).`
`:- type list_elem ---> []; [elem | list_elem].`
Figure 3.1 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. □

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 throughout. 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 section.

In Figure 3.1 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 `list_elem`. We also put *all* the elements, which have the type `elem`, 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 2.3, 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.

3.2 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 thesis 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 modelled 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 source node of the edge contains references into (points-to) the region bound to by the region variable represented by the target node of the edge. 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 3.2. The type-based region graph for the type `list_elem` in Example 3.1 is shown in Figure 3.2. The `[|]` principal functor is stored in $R^{list\text{-}elem}$. 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 $R^{list\text{-}elem}$, 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 3.1. □

Example 3.3. 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 3.3.

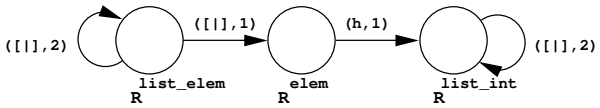


Figure 3.2: The type-based region graph of the type `list_elem`.

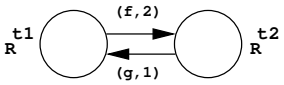


Figure 3.3: Type-based region graph of mutually recursive types.

3.3 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 <= 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 => 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**, $G(N, E)$, for a set of variables V, 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 repre-

sents 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 3.1 (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 which 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 4 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 3.4. Consider the following sequence of code to construct the term that L in Example 3.1 is bound to. The type of K is of no importance.

```
...,
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 3.4. 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. □

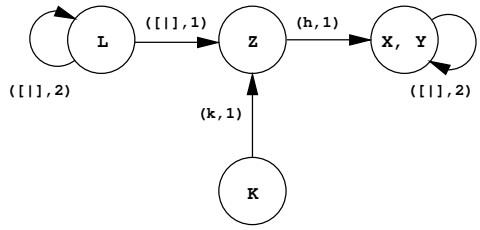


Figure 3.4: Modelling of sharing information.

3.4 Conclusion

We have described the memory models of terms in a region-based system. Terms, program variables, types, and in-memory storage of terms are closely related. We have relied on types, which are statically available to the Mercury compiler, to assign regions to terms and program variables. The region-based memory model of a procedure is then defined by a region points-to graph for its variables and that of a program is by a set of region points-to graphs of its procedures. Region points-to graphs not only contain information about locations of program terms but also represent sharing among them at the level of regions. We formulate this sharing as region-sharing. Our next task now is to develop a program analysis that computes these memory models for Mercury programs, which will be dealt with in the next chapter.

Chapter 4

Region Points-To Analysis

In this chapter we develop a region points-to analysis that 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. As we described in Chapter 3 the memory model of the whole program consists of the region points-to graphs of its procedures.

4.1 Overview

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 out-edge with a specific label from one node to another, the operation is recursive, i.e., unifying two nodes may cause more nodes to be unified.

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 ours inside the MMC (Chapters 7 and 8), if certain terms do not need heap storage, their corresponding regions can just be ignored.

4.2 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.

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 \ {m}
vars(n) = vars(n) ∪ vars(m)
for all (m, (f, i), k) ∈ E do
    E = E \ {(m, (f, i), k)}
    if (n, (f, i), k) ∉ E then
        E = E ∪ {(n, (f, i), k)}
    end if
end for
for all (k, (f, i), m) ∈ E do
    E = E \ {(k, (f, i), m)}
    if (k, (f, i), n) ∉ E then
        E = E ∪ {(k, (f, i), n)}
    end if
end for
for all l, l' ∈ N do
    if (n, (g, j), l) ∈ E ∧ (n, (g, j), l') ∈ E ∧ l ≠ l' then
        unify(l, l')
    end if
end for
```

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

Require: p is in superhomogeneous form.
Ensure: Sharing created by explicit unifications is represented in G_p.

```
G_p = (∅, ∅)
for all X ∈ p do
    G_p = G_p ⊎ init_rptg(X)
end for
for all unif ∈ p do
    if unif ≡ (X := Y) then
        unify(n_X, n_Y)
    else if unif ≡ (X => f(Y_1, ..., Y_n) or X <= f(Y_1, ..., Y_n)) then
        for i = 1 to n do
            unify(node(n_X, (f, i)), n_{Y_i})
        end for
    end if
end for
```

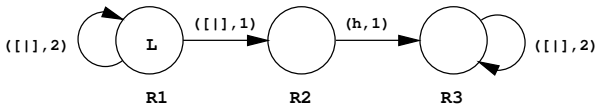


Figure 4.1: The resulting region points-to graph of *init_rptg*(L).

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 region points-to graph returned by *init_rptg*(L), with `L` of the type `list_elem` (Figures 3.1 and 3.2) is shown in Figure 4.1 with the assumption that the region variables are fresh.

The intraprocedural analysis adds all the sharing created by the unifications in the procedure to G_p. For assignment, construction and deconstruction unifications we unify the nodes corresponding with the sharing created by them. We ignore test unifications because they do not create any sharing.

4.3 Interprocedural Analysis

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

For a call $q(Y_1, \ldots, Y_n)$, the head of the defining procedure is assumed to be $q(X_1, \ldots, X_n)$. The 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 $\alpha : N_q \rightarrow N_p$ that maps the nodes of the formal arguments (X_i's) to the nodes of the corresponding actual arguments (Y_i's). Then these nodes are the starting points for the integration of the remaining region-sharing. This is done by following the relevant edges in G_q to extend the α function to all the relevant nodes in G_q (rule P2) and to unify the relevant nodes in G_p (rule P1).

For a whole program, we can first do the intraprocedural analysis for every procedure. Then given the fact that in the interprocedural analysis the analysis information is only propagated from graphs of callees to those of callers, we can do the interprocedural analysis for a program efficiently by decomposing the call-dependency graph into a tree of strongly connected components, and analysing the components in bottom-up order. Algorithm 4 illustrates this approach.

The points-to graphs of the `split` and `qsort` procedures in the *quicksort* pro-

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 α relation.
    for $k = 1$ to n **do**
      $\alpha(n_{X_k}) = n_{Y_k}$
    end for

    % Ensure α 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_j})
      end if
    end for

    % Integrate sharing in G_q into G_p.
    In the graph G_q, start from each n_{X_i}, follow each edge once and apply the rules P1 and P2 in Figure 4.2 when applicable.
  end for
until There is neither change in G_p nor in any of the α functions.

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

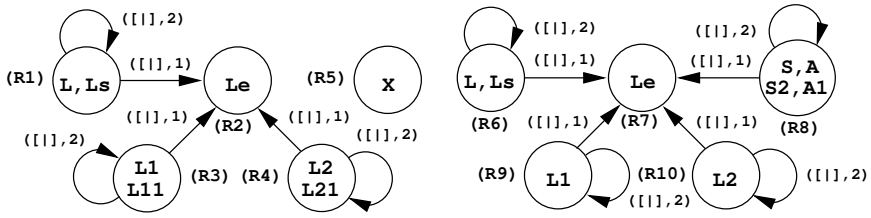
$$\frac{(n_q, (f,i), m_q) \in E_q \quad \alpha(n_q) = n_p \quad (n_p, (f,i), m_p) \in E_p \quad \alpha(m_q)\ undefined}{\alpha(m_q) = m_p} \quad \text{(P2)}$$

Figure 4.2: Interprocedural analysis rules.

Algorithm 4 Region points-to analysis of a program

Require: A Mercury program P with its procedures in superhomogeneous form.
Ensure: Region points-to graphs for all procedures.

```
for all procedure p in P do
    intraproc(p)
end for
Decompose P's call-dependency graph into a tree of strongly connected components (SCCs).
for all SCC (bottom-up order) do
    repeat
        for all p in SCC do
            interproc(p)
        end for
    until reach a fixpoint
end for
```



(a) `split` (b) `qsort`

Figure 4.3: The region points-to graphs of `split` and `qsort`.

gram in Example 2.3 are shown in Figure 4.3. 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.

Illustration of Interprocedural Analysis

In Figures 4.4 and 4.5, we illustrate the interprocedural region points-to analysis of the `qsort` procedure at two call sites. Each figure contains two region points-to graphs: one of the caller (namely `qsort`) at the top and the other of the involved callee at the bottom. The directed dash-dot lines represent the typical α mappings at each call site.

Figure 4.4 shows the processing at the call to `split` at the program point (4) in `qsort`. At this point the region points-to graph of `split`, G_{split}, is already in

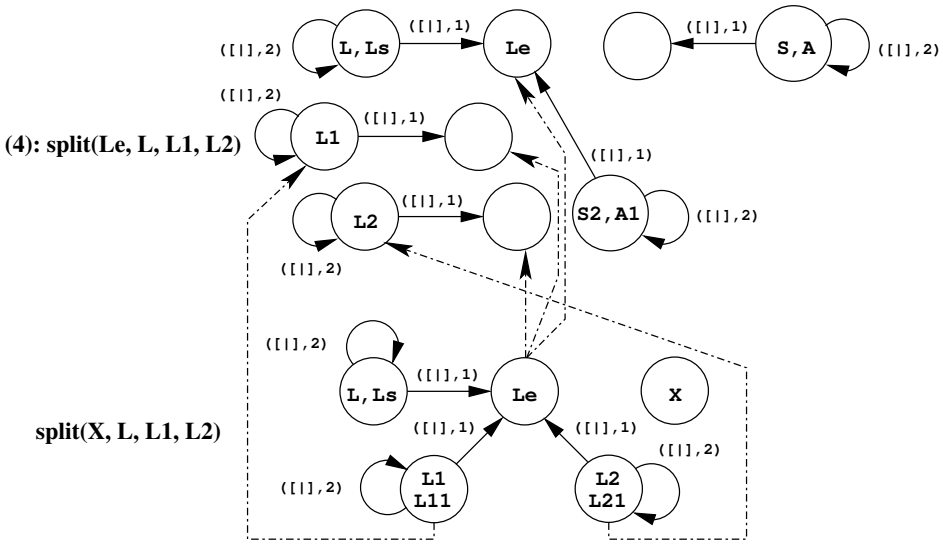


Figure 4.4: Capturing region-sharing in a call to another procedure.

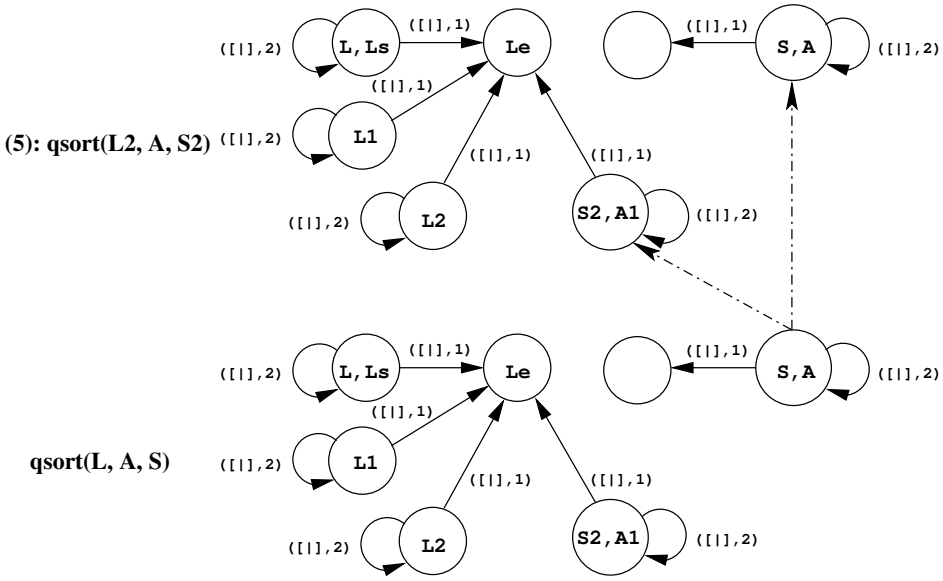


Figure 4.5: Capturing region-sharing in a recursive call.

its final state, which is the same as the graph in Figure 4.3(a). At this point, the points-to graph of `qsort`, G_{qsort}, which is obtained by the intraprocedural analysis, does not yet represent the sharing of the elements between the input list `L` and the temporary lists `L1` and `L2`. The interprocedural analysis makes the sharing available from the sharing of the elements between `L`, `L1`, and `L2` in `split`. Starting from $\alpha(n_{L1}) = n_{L1}$, $\alpha(n_{L2}) = n_{L2}$, and $\alpha(n_L) = n_L$ (not drawn in Figure 4.4), the application of the rule P2 will capture the mappings from n_{Le} in G_{split} to three different nodes in G_{qsort}. This will cause the three nodes to be unified due to the rule P1. And the region points-to graph of `qsort` after this step is the same as the ones shown in Figure 4.5, which are the graphs used in the analysis of the recursive call to `qsort` at (5).

For the analysis of the recursive call at (5), notice that the sharing among the arguments of `qsort` has not been fully represented in the callee's graph yet. For example, the sharing of the elements between the input list `L` and the output list `S` is not yet available. This sharing, represented as the region-sharing between n_L and n_S, is captured after, for example, the rule P1 is applied to unify n_A and n_{S2} due to $\alpha(n_A) = n_A$ and $\alpha(n_S) = n_{S2}$. After this step the region points-to graph of `qsort` is the same as the final one in Figure 4.3(b). (The analysis of the second recursive call at (7) does not add any more information to the graph. In reality, the analysis would also need to go one more iteration to confirm that the fixpoint of G_{qsort} has been reached.)

4.4 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 4.1 *The region points-to analysis of a program terminates.*

Proof. An α 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 α 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 4.2 *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 4.2, 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 4.3 (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 3.1, 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 4.4 *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 α relation that represents the connection between G_q^{sub} and G_p^{sub}. This initial α relation connects the nodes of the formal arguments with the nodes of the corresponding actual arguments. In this α 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 α 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 4.5 *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.

Lemma 4.5 essentially shows that the α 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 4.6 (Sharing created by procedure calls) *All the region-sharing in* G_q^{sub} *is also in* G_p^{sub}.

Proof. The proof of Theorem 4.6 follows from Lemma 4.4 and Lemma 4.5.

Now we can continue with the proof of the sharing-among-terms part of Theorem 4.2.

Proof. (Sharing among terms) The proof of the second part of Theorem 4.2 follows from Lemma 4.3 and Theorem 4.6, 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. (An example for this can be found in Figure 4.5.) And we already showed by Lemma 4.3 that this sharing is represented in the region points-to graph of the procedure containing the unification. By Theorem 4.6, we showed that the sharing will also be represented in the region points-to graphs of any procedures that invoke the procedure. Therefore even when the procedure is recursive, the sharing created by the recursive calls will finally be represented in its region points-to graph. This concludes the proof of Theorem 4.2.

In the future chapters when we mention region points-to graphs we mean the ones obtained by the region points-to analysis. We will only be specific if otherwise.

4.5 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 during the execution of the procedure. A construction unification is the only construct in Mercury that allocates memory. When processing a construction unification $X <= f(\ldots)$ 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 2.2 and Figure 4.3, $allocation(split) = \{R3, R4\}$, $allocation(qsort) = \{R8, R9, R10\}$,

4.6 Conclusion

In this chapter we have presented a program analysis that computes a region-based memory model for a Mercury program. The analysis is unification-based: the

region points-to graphs of program variables are unified to reflect the sharing the program possibly creates. We prove that the resulting memory model adequately represents locations and sharing of the terms in a program. The sharing relation is important for the estimation of lifetimes of the region variables as we will see in Chapter 5.

Chapter 5

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. In this chapter we construct a region liveness analysis that approximates lifetimes of the region variables, i.e., their liveness, to decide when a region needs to be created and when it can safely be reclaimed. We make a distinction between local liveness and global liveness. Local liveness concerns the lifetime of the region variable inside the procedure itself, namely when we consider the procedure alone. Global liveness concerns the liveness with respect to the whole program, namely when we take into account the call sites of the procedure. Local liveness is computed in Section 5.2 and global liveness is described in Section 5.3.

5.1 Technical Background

A region variable being live means that it should be bound to a region and is possibly used in future (forward) execution. 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. The goal of the region liveness analysis is therefore to decide which region variables are live at each program point and which region arguments become live or stop being 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

Table 5.1: Input and output arguments of unifications.

	in_args	*out_args*
$X <= f(X_1, \ldots, X_n)$	$\{X_1, \ldots, X_n\}$	$\{X\}$
$X => f(X_1, \ldots, X_n)$	$\{X\}$	$\{X_1, \ldots, X_n\}$
$X == Y$	$\{X, Y\}$	$\emptyset$
$X := Y$	$\{Y\}$	$\{X\}$

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 arguments to *atom*. For specialized unifications they are defined in Table 5.1. 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.

5.2 Live Region Variables at a Program Point

In this section we specify the analysis that computes the local liveness of region variables in a procedure. We express local liveness by the sets of region variables that are live before and after every program point in the 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 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.

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

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) = ∅
end for
for all ep ≡ ⟨l_1, ..., l_n⟩ in p do
    for j = n downto 1 do
        i = pp(l_j)
        if j = n then
            LV_after(i) = out_args(p)
        else
            LV_after(i) = LV_after(i) ∪ 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) \ out_args(l_j)) ∪ in_args(l_j)
        end if
    end for
end for
```

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

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

Then the live region variable analysis of a procedure is specified in Algorithm 6 in which the sets of live region variables before (LR_{before}) and after (LR_{after}) a program point are straightforwardly computed as the unions of the *Reach* sets of all live variables in LV_{before} and in LV_{after} of the program point, respectively.

5.3 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 section introduces an analysis that, by taking into account the calling contexts of a procedure in the whole program, decides which region variables become live or cease to be live inside the procedure. With this global liveness 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:

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) = ∅
    for all X ∈ LV_before(i) do
        LR_before(i) = LR_before(i) ∪ Reach(X)
    end for
    for all X ∈ LV_after(i) do
        LR_after(i) = LR_after(i) ∪ Reach(X)
    end for
end for
```

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

The motivation is that, in the 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 creates those mapped to by ones in $bornR(q)$ just before and removes 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 $outlivedR(p)$.

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

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

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

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

The literal l is a call to $q(\dots)$ at a program point.

Figure 5.1: Region liveness analysis rules.

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 5.1 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 of region alias 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). Note that the rules L2 and L4 are there to deal with region alias. The aim of handling region alias is to make sure that any region variables that are involved in region alias do not belong to either *bornR* or *deadR* sets. It is probably easier to read the two rules as if we are dealing with r'. We can also apply the rules to r'', with r'' in the role of r' in the rules. If r'' belongs to either set, it is eliminated from the set. Otherwise nothing happens.

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 Figure 2.3, `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 $\langle(3),(4),(5),(6),(7)\rangle$. [1] Note that the third execution path of `split` does not contain the test at (5) because of the semantics of if-then-else. The LV and LR

[1] For convenience, we use program points to describe execution paths.

Table 5.2: Live variable and live region variable sets in *quicksort* program.

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

(a) *split*

pp	LV	LR
(1_b)	$\{L, A\}$	$\{R6, R7, R8\}$
$(1_a, 2_b)$	$\{A\}$	$\{R8, R7\}$
(2_a)	$\{S\}$	$\{R8, R7\}$
(3_b)	$\{L, A\}$	$\{R6, R7, R8\}$
$(3_a, 4_b)$	$\{A, Le, Ls\}$	$\{R8, R7, R6\}$
$(4_a, 5_b)$	$\{A, Le, L1, L2\}$	$\{R8, R7, R9, R10\}$
$(5_a, 6_b)$	$\{Le, L1, S2\}$	$\{R9, R7, R8\}$
$(6_a, 7_b)$	$\{L1, A1\}$	$\{R9, R7, R8\}$
(7_a)	$\{S\}$	$\{R8, R7\}$

(b) *qsort*

Table 5.3: Division of the set of region variables.

	localR	*bornR*	*deadR*	*outlivedR*
split	$\emptyset$	$\{R3, R4\}$	$\{R1\}$	$\{R2, R5\}$
qsort	$\{R9, R10\}$	$\emptyset$	$\{R6\}$	$\{R7, R8\}$

sets of `split` are in Table 5.2(a), of `qsort` in Table 5.2(b) (see also Figure 2.2 and Figure 4.3). 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 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, *R5* is 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 5.3.

5.4 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 (Nielson, Nielson, and Hankin 1999). Theorem 4.2 guarantees the locations of variables and their possible 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 points-to graphs.

The analysis in Section 5.3 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 5.1 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.

5.5 Conclusion

In this chapter we have developed a program analysis to estimate lifetimes of the region variables. The idea is to know the places where a region needs to be created and where it can safely be removed. Recall that we have chosen to deal with lifetimes with respect to forward execution only. Backward executions when happen will be handled by runtime support, which will be described in Chapter 8.

In our framework region variables can live across procedure boundaries and can have arbitrarily overlapping lifetimes. Regions can be passed among procedures by using region arguments. We classify the set of region variables of a procedure into local ones, which only exist in the scope of the procedure, and nonlocal ones, which may exist outside the scope. With local ones only local liveness, namely the liveness before and after the program points in the procedure, is needed to decide where to create and remove them. For nonlocal ones, other than local liveness we also compute their global liveness by taking into account the calling contexts of the procedure.

After this phase we have enough information about regions and the places where they need to be created and removed in a program. The next task is to annotate the program with these pieces of information, which is the main content of the next chapter.

Chapter 6

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 left-hand 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 sections.

6.1 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 during the execution of the procedure, called *allocR*. This set of region variables can be computed by $allocR = (inputR \cup outputR) \cap allocation$ (Section 4.5). Note that *allocR* is not necessarily disjoint with *bornR*, *deadR* and *outlivedR*.

$$\frac{\begin{array}{c} l \equiv q(\ldots) \\ r \in LR_{after}(pp(l)) \setminus LR_{before}(pp(l)) \\ r \in localR(p) \cup bornR(p) \cup deadR(p) \\ r = \alpha(r') \rightarrow r' \notin bornR(q) \end{array}}{add\ \text{“}create(r)\text{”}\ before\ l} \quad \text{(T1)}$$

$$\frac{\begin{array}{c} l \equiv X <= f(\ldots) \\ r \in LR_{after}(pp(l)) \setminus LR_{before}(pp(l)) \\ r \in localR(p) \cup bornR(p) \cup deadR(p) \end{array}}{add\ \text{“}create(r)\text{”}\ before\ l} \quad \text{(T2)}$$

$$\frac{\begin{array}{c} l \equiv q(\ldots) \\ r \in LR_{before}(pp(l)) \setminus LR_{after}(pp(l)) \\ r \in localR(p) \cup deadR(p) \cup bornR(p) \\ r = \alpha(r') \rightarrow r' \notin deadR(q) \end{array}}{add\ \text{“}remove(r)\text{”}\ after\ l} \quad \text{(T3)}$$

$$\frac{\begin{array}{c} l \equiv unif \\ r \in LR_{before}(pp(l)) \setminus LR_{after}(pp(l)) \\ r \in localR(p) \cup deadR(p) \cup bornR(p) \end{array}}{add\ \text{“}remove(r)\text{”}\ after\ l} \quad \text{(T4)}$$

$$\frac{\begin{array}{c} l'\ is\ next\ to\ l\ in\ an\ execution\ path \\ r \in LR_{after}(pp(l)) \setminus LR_{before}(pp(l')) \\ r \in localR(p) \cup deadR(p) \cup bornR(p) \end{array}}{add\ \text{“}remove(r)\text{”}\ before\ l'} \quad \text{(T5)}$$

$$\frac{\begin{array}{c} r \in VR(pp(l)) \setminus LR_{after}(pp(l)) \\ r \in localR(p) \cup deadR(p) \cup bornR(p) \end{array}}{add\ \text{“}remove(r)\text{”}\ after\ l} \quad \text{(T6)}$$

Figure 6.1: Transformation rules.

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 α function at the call site.

6.2 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.

6.2.1 Transformation Rules

The transformation rules in Figure 6.1 make use of the local and global liveness of region variables to introduce *create* and *remove* instructions when necessary.

Transformation Rules T1 and T2

To justify the transformation rules consider a literal l at a program point i in a procedure p. When a region variable *locally* becomes live at i, namely when it is

not live before i but is live after i, we need to ensure that the region variable is bound to a region after i. It will never be the case that a region variable locally becomes live because it is not live after a program point but it is live before the next program point in an execution path (Proposition 6.2, Section 6.3). So, we consider when the region, which is bound to by the region variable that locally become live, is created. If l creates the region, 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 in the latter case we need to add a *create* instruction before l and this occurs when the region variable belongs to either $localR(p)$, $bornR(p)$, or $deadR(p)$. This is reflected by the transformation rules T1 and T2. The first condition on r in these two rules means the region variable locally becomes live at i. The second condition on r means according to the global liveness p is allowed to create the region. It is intuitively clear that p needs to a region to which a region variable that belongs to $bornR(p)$ or $localR(p)$ is bound. The reason that T1 and T2 also have $deadR$ in the second condition on r is that 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 because R is in $deadR(p)$. The third condition in T1 is make sure that l, which is a procedure call, does not create the region. The rule T2 is for specialized unifications. Recall that unifications never create regions. This rule is restricted to only construction unifications because for the other kinds of specialized unifications the first condition on r always fails (see Proposition 6.3, Section 6.3).

Transformation Rules T3, T4, and T5

When a region variable locally ceases to be live, we need to distinguish two cases. The first case is when the region variable is live before i but not live after i. We now consider when the region bound to by the region variable is removed. If the global liveness of the region variable does not allow p to remove the region, the region is removed by a caller of p and no annotation is introduced at i. Otherwise, the region is removed inside p. This means the region variable is in one of the $deadR$, $localR$, or $bornR$ sets of p. There are two subcases: if l 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. The first condition on r in these two rules says that the region variable locally become dead. The second one expresses that p is allowed to remove the region. Similar to the case of 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. Again, the third condition on r in the rule T3 is to ensure that the procedure call does not remove the region.

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.

Example of Re-Creation

We illustrate the effects of the re-creation of regions by two procedures in Figure 6.2 and their region-annotated counterparts in Figure 6.3. We include the definition

```
% p(in, out).
p(A, B) :-
(1) C <= [1],
    ( if
(2)     A == 1
      then
(3)     B := C
      else
(4)     B <= [2]
    ).
```

```
% q(in, out).
q(X, Y) :-
(1) Z := length(X),
    ( if
(2)     Z == 1
      then
(3)     V := X
      else
(4)     V <= [1]
    ),
(5) Y := Z + length(V).
```

```
length(L) = N :-
(   L == [], N := 0
;   L => [_ | T],
    N := length(T) + 1
).
```

Figure 6.2: Effect of re-creation of regions.

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

```
p(A, B@R1) :-
    create(R1),
(1) C <= [1] in R1,
    ( if
(2)     A == 1
      then
(3)     B := C
      else
        remove(R1),
        create(R1),
(4)     B <= [2] in R1
    ).
```

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

Figure 6.3: Effect of re-creation of regions: region-annotated version.

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 2.2). 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.

Assume that `p` can create `R1`, i.e., no calling context forces it otherwise. So `R1` is in $bornR(p)$. In Figure 6.3, 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: Rule T6

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 2.1) to explicitly tell the compiler that we do not care about its value. A singleton variable is often a mistake of the programmer. The 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 are not used in the future. Being instantiated means that we need regions to store their terms, and we do want to remove the regions. However, that such variables are not used in the future makes them *never* live according to our concept of live variables (Section 5.2). Therefore we may not always rely on the change of liveness, from live to dead, to remove related regions (as in the rules T3-T5) because some of the regions may never be live. That is why we have the rule T6 that tries to remove a region reachable from a void variable right after the point where the void variable gets instantiated. 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.

6.2.2 Insertion Algorithm

The insertion of the instructions is specified by Algorithm 7 in which the transformation rules in Figure 6.1 are applied to the literal at each program point.

Algorithm 7 Insertion of region instructions in a procedure p.

Require: p in superhomogeneous form; the information about region points-to graphs and region liveness is ready.

```
for all program point i in p do
    l = literal(i)
    apply rule T6 to l
    if l ≡ unif then
        apply rule T4 to l
        if l ≡ X <= f(...) then
            apply rule T2 to l
        end if
    else
        apply rules T1 and T3 to l
    end if
end for
for all ep ≡ ⟨l_1, ..., l_n⟩ in p do
    for j = 1 to n − 1 do
        apply rule T5 to l_j, l' ≡ l_{j+1}
    end for
end for
```

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 6.3.

The first loop in Algorithm 7 applies all the transformation rules, except T5, to the literals at all the program points in a procedure. We use the function $literal(i)$ to refer to the literal at the program point i. While the rule T6 can be applied to any literal, T4 needs to be tried only when the literal at a program point is a unification, T2 only when the literal is a construction unification, and T1 and T3 only when the literal is a procedure call. The second loop follows every execution path to try rule T5, which needs to consult the information at two consecutive program points at the same time.

The result of the program transformation of the *quicksort* program in Example 2.3 has been shown in Figure 2.3. The addition of the *remove* instructions after the first program points in both *qsort* and *split* procedures results from the application of T4. Two *create* instructions inserted in the *split* procedure are effects of T2.

6.3 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 6.1 *Consider a procedure p in a program P. We call P' the region-annotated program that is produced by applying the analyses and transformation in Chapters 4, 5, and 6 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 6.1, we formulate several propositions.

Proposition 6.2 *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.

Proposition 6.3 *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 $X <= f(X_1, \ldots, X_n)$. By definition (Algorithm 5) $LV_{before}(i) = LV_{after}(i) \setminus \{X\} \cup \{X_1, \ldots, X_n\}$. So we can compute $LR_{before}(i) = (\bigcup Reach(V) \; \forall \; V \in LV_{after}(i) \wedge V \neq X) \cup (\bigcup Reach(X_i) \; i = 1..n)$. We can also write $LR_{after}(i) = (\bigcup Reach(V) \; \forall \; V \in LV_{after}(i) \wedge V \neq X) \cup Reach(X)$. 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 $LR_{before}(i) = (\bigcup Reach(V) \ \forall \ V \in LV_{after}(i) \wedge V \neq X) \cup Reach(Y)$. We can write $LR_{after}(i) = (\bigcup Reach(V) \ \forall \ V \in LV_{after}(i) \wedge V \neq X) \cup Reach(X)$ 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 6.4 *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 6.3).

If the unification is a construction then $LR_{before}(i) \subseteq LR_{after}(i)$ (Proposition 6.3). 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 6.1.

Proof. (Theorem 6.1)

Hypothesis: Assume that Theorem 6.1 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 the region variable of the caller 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 6.2). 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 6.4 we have that $LR_{before}(i) \subset LR_{after}(i)$ and then from Proposition 6.3 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, that l is a construction unification, which does not remove any regions, means 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 6.5 *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 4.2 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 6.1 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.

Theorem 6.6 *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 4.2 we know the region variables where the terms that the variable is possibly bound to are 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 X are 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.

6.4 Conclusion

We have described the transformation in which region points-to graphs and region liveness information are used to annotate a Mercury program with region information. In a region-annotated program a region variable is allowed to change its bindings if it is safe to do so. This, in certain cases, leads to shorter lifetimes for regions, thus better memory reuse.

We proved the correctness of the annotated programs by showing that memory accesses in them are safe in the sense that when a term is allocated, its region has been created and a region is removed only when no more memory accesses happen to it.

Region-annotated programs contain enough region information to be run by a specialized runtime system, which will be explained in Chapters 7 and 8.

Chapter 7

Runtime Support for Regions in Deterministic Programs

In this and Chapter 8, we describe the runtime support needed to execute region-annotated programs. The liveness of the regions in the programs is determined with respect to forward execution. Let us have a look at the lifespan of a region in a deterministic program (which has only forward execution). A region comes into existence when a `create(R)` instruction is executed which assigns memory to the region and binds the region variable `R` to a so-called *region handle*, which refers to the assigned memory. From then on, terms are *allocated* into the region by construction unifications annotated with `R`. The lifetime of a region is ended by a `remove(R)` instruction when the memory referred to by the region handle bound to `R` is no longer needed and is *reclaimed*. The runtime support for this is described in the current chapter.

All the above works well in the case of simple forward execution in deterministic programs but not in the case of backtracking. It is no longer safe for a `remove` instruction to reclaim the region when the data in the region is still needed after backtracking. So in the next chapter we extend the runtime support to take into account backtracking. The runtime support for backtracking has to prevent the reclamation of backward live regions and it also has to reclaim the memory used by backtracked-over computations.

7.1 Region Data Structure

In our system, a region is a singly-linked list of fixed-size region pages. Each region page has a *data area*, an array of words that can be used to store program data, and a pointer to the next region page to form the singly-linked list. (In the newest region page of a region, this pointer is of course null.) The *handle* of the region (the way the rest of the system refers to it) is the address of the *region header*. Besides some other fields that we will introduce later, the header structure includes a *region size record*: a pointer to the newest region page, and a pointer to the next available word in the newest region page. Since region pages have a fixed size, these two values implicitly also specify the amount of free space in the newest region page. To avoid fragmentation, we store each region header at the start of the data area of its region's *first* region page. Figure 7.1 shows a region with two region pages; the shaded areas represent memory allocated to user data.

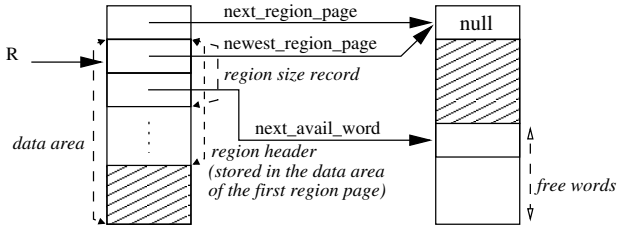


Figure 7.1: The data structure of a region `R`.

7.2 Region Operations

There is no bound on region size. When a *region is created* it will contain only one region page, but it can be extended by adding more region pages when necessary. The program maintains a global list of free region pages. If the free list runs out, the program requests a big chunk of memory from the operating system, divides it into region pages, and adds them to the free list. When a region needs to be extended, we take a region page from the free list and add it to the region as its new last region page, and then update the region's size record. When a *region is reclaimed*, we return all its region pages to the free list. An *allocation into a region* always happens in its newest region page simply by increasing the pointer to the next available word. When the free amount in this region page is not enough for an allocation, we extend the region before allocating.

This aspect of our implementation is generally similar to the "standard" RBMM implementations for SML and Prolog, which are described in detail in (Makholm

2000b; Makholm 2000a). The advantage of this implementation is that the basic region management actions are bounded in time. Disadvantages are that there is no natural size for the region pages (Tofte, Birkedal, Elsman, and Hallenberg 2004), and that if the remaining space of a region page is not enough for an allocation, that space will be wasted when a new region page is added.

Chapter 8

Runtime Support for Backtracking

8.1 Introduction

Backtracking introduces two issues that need to be handled: reclaiming the memory allocated by the computations backtracked over, and ensuring that regions are reclaimed only when they are dead with respect to both forward and backward execution. The first issue obviously has to be handled at runtime. For our initial implementation, we have chosen to deal with the second issue, backward liveness, in the runtime system too. We expect this to give us the insights we will need in the case we want to extend the program analysis to handle backward liveness both safely and precisely. Moreover, our current system can serve as a reference for that work.

In Mercury, disjunctions are the main source of backtracking because they provide alternatives. But an if-then-else is just a special kind of disjunction: (*if* C *then* T *else* E) is semantically equivalent to (C, T; *not some* $[\cdots]$ C, E). Operationally, Mercury will try C. If C succeeds, Mercury executes T; if C fails, it executes E as if C had never been tried. The handling of commit (Section 2.2) is related to the handling of backtracking because committing to a solution may prune some alternatives of relevant disjunctions. Therefore, we need to provide runtime support for backtracking in the context of these three language constructs.

The region-annotated program in Figure 8.1 illustrates our two tasks.

We constructed this program, which, unfortunately, has no intuitive meaning, to illustrate the interaction between regions and backtracking; we will use it as our running example when describing the runtime support. (For completeness, we include the definitions of `member` and `length` whose behaviour, as we will see, is of no importance in this work.) Regarding the lifetime of the regions, in `main`,

```
main(!IO) :-
    create(R1),
(1) X <= [1, 3, -1, 3] in R1,
    create(R2),
(2) A <= [-2] in R2,
    ( if
        create(R3),
(3)     p(X@R1, A@R2, B@R2, Y@R3)
      then
(4)     io.write(B, !IO),
        remove(R2),
(5)     io.write(Y, !IO),
        remove(R3)
      else
(6)     io.write(X, !IO),
        remove(R1),
(7)     io.write(A, !IO),
        remove(R2)
    ).

% mode(in, in), semidet
member(X, L) :-
    L => [H | T],
    (H == X ; member(X, T)).
```

```
:- pred p(list_int, list_int, list_int, list_int).
:- mode p(in, in, out, out) is semidet.
p(X@R4, U@R5, V@R5, Y@R6) :-
(1) X => [H | T],
    ( if
        remove(R4),
(2)     H < 0
      then
(3)     Y  <= [H] in R6,
(4)     ( if    member(H, U)
(5)       then  V := U
(6)       else  V <= [H | U] in R5
        )
      else
(7)     p(T@R4, U@R5, V@R5, Y1@R6),
(8)     ( if    length(V) > length(Y1)
(9)       then  fail

        )
    ).

% mode(in) = out, det.
length(L) = N :-
    (   L == [], N := 0
    ;   L => [_ | T],
        N := length(T) + 1
    ).
```

Figure 8.1: Illustrating the interaction of regions and backtracking.

`R1` and `R2` are created before the construction of the two lists `X` and `A`. The `main` procedure creates `R3` before the call to `p` at (3), and `p` will use this region to store the skeleton of `Y`. All the `remove` instructions for regions are added after the last *forward* uses of the terms stored in them. Two procedures `member` and `length` only read their input variables, so no region argument is needed. For p, $deadR(p) = \{R4\}$, $bornR(p) = \emptyset$, $outlivedR(p) = \{R5, R6\}$, and $allocation(p) = \{R5, R6\}$.

- Task 1: Preventing the reclamation of backward live regions. The condition of the if-then-else in the `main` predicate is the call to the semidet procedure `p`. The RBMM transformation marks the region `R1` for removal in the call because it is forward dead (it is not used in the then part) even though it is backward live (it *is* used in the else part). We must make sure that `R1` is not actually removed while it is backward live. In this case, that means we need to delay the reclamation of `R1` until we reach the then part; it is not safe to actually destroy `R1` if the condition fails. We therefore distinguish *reclaiming* a region, which makes the memory of the region available for other uses and thus potentially destroys its contents, from the operation of *removing* a region, which causes the region to be reclaimed only when it is safe to do so. Basically, a region is removed when it is forward dead, and it

is reclaimed when it is both forward and backward dead.

- Task 2: Reclaiming the memory used by backtracked-over computations. The call to `p` has two output arguments, `B` and `Y`. `main` tells `p` to put any cells for `B` in `R2`. The condition is extended with a `create` instruction to create `R3` so that `p` can put `Y` into it. If the condition succeeds, we must leave both regions alone. If the condition fails, we should restore `R2` to its size before the condition, and we should reclaim `R3` in its entirety.

8.2 Runtime Concepts

We now define several runtime concepts that we will use in the description of the runtime support for backtracking.

Old vs new regions. A region is *old* with respect to a point during the execution of a program if it was created before that point, otherwise it is *new* with respect to that point. We also refer to old regions as the *existing regions*. To allow efficient checks whether a region is old or new, we maintain a global *region sequence number* counter (starting at one) and include a `sequence_number` field in region headers. When we create a region, we timestamp it by setting its `sequence_number` from the global counter, and increment the counter. If at a point during the execution of the program (such as a resumption point where the program resumes when backtracking) we save the current sequence number, then all the regions which are created before that point, i.e., the old regions with respect to the point, will have their sequence numbers smaller than the saved value; ones which are created after that point, i.e., the new regions with respect to the point, will have their sequence numbers greater than or equal to the saved value. When the program backtracks to the save point, we can use the saved value to check whether a region has been created before or after the point. In the context of RBMM, the memory that we want to reclaim at a resumption point will be new allocations into existing regions, and new regions in their entirety (since they have been created by the computation we have just backtracked over).

Region list. To do instant reclaiming of new regions, knowing the sequence numbers of the new regions is not enough; we also need to *reach* them. We therefore link all the live regions into a doubly-linked *region list* (using two additional pointers in the region header). We maintain a global pointer to the head of the list, which will be the newest live region. When a region is created we add it to the head of the region list; when a region is reclaimed we remove it from the list. We maintain the invariant that the region list is ordered by regions' creation time, newest first. To reclaim new regions, we can traverse the region list from its head and reclaim each region until we meet an old one.

Region size snapshots. To do instant reclaiming of new allocations into an existing region, we need the old size of the region. When we need to remember the size of a region at a point, we can save its region size record (Section 7.1) at that point.

Protection. We will prevent the destruction of backward live regions by *protecting* them so that when a removal happens to the region during forward execution, the removal will be ignored.

8.2.1 Changes to Live Regions by a Goal

When providing support for backtracking, sometimes we want to know about the changes which may be caused by a goal to the set of regions the goal may refer to. This means we need to know about any new regions the goal creates, any live regions the goal removes, and any live regions in which the goal performs allocations. We refer to these sets of regions as the goal's *created*, *removed*, and *allocated* sets, respectively. We have computed several region variable sets of procedures, such as *inputR*, *bornR*, *deadR*, and *allocation*. The *created*, *removed*, and *allocated* sets of goals can be computed from these in a fairly straightforward manner.

Changes to live regions by a goal: creation. Only `create` instructions and procedure calls may create regions. A `create` instruction always creates the region in its argument. A procedure call will create the regions that are the actual region arguments corresponding to the formal arguments in the *bornR* set of the called procedure. For a compound goal, its created set is the set of all regions created inside it, *including* those that are also removed by it.

Changes to live regions by a goal: removal. We can similarly use `remove` instructions and the *deadR* sets of procedures to compute the removed set of each goal. Since we only care about the old regions which are removed inside a goal, we exclude regions created inside the goal (i.e., the goal's created set) from its removed set.

Changes to live regions by a goal: allocation. A region is allocated into in construction unifications and procedure calls. A construction unification will allocate into the region with which it is annotated. A procedure call possibly allocates into the region variables that are mapped to by those in the procedure's *allocation* set. Because we are only interested in the old regions, we restrict the allocated set to the ones mapped to by those in $\mathit{inputR} \cap \mathit{allocation}$.

Changes to live regions by a goal: an example. Take the condition of the if-then-else in the procedure `p` in Figure 8.1 as an example goal. We say that the region `R4` is removed in the condition because `R4` is live before the condition and `remove(R4)` has been added to the condition. Or take the condition of the if-then-else in `main`. We say region `R3` is created in the condition because `create(R3)` has been inserted into the condition, while region `R1` is removed in the condition

because it is live before the condition and is removed in the call to `p`. We have $allocation(p) = \{R5, R6\}$, but if consider only input region variables then the call to `p` at (3) in `main` may only allocate into `R2`, which is mapped to by `R5`. So during the condition in `main` `R2` may be allocated into.

We provide the runtime support for backtracking for a program by generating extra supporting code at the right places to achieve our goals. In the next three sections we will describe in detail the support for disjunctions, if-then-elses, and commits.

8.3 Support for disjunction

Although the Mercury language does not specify the language's search strategy, the Mercury compiler supports only one search strategy: depth-first search with chronological backtracking, so that the disjuncts of each disjunction are tried in order. Given a disjunction `(g1; ...; gi; ...; gn)`, we refer to `g1` as the first disjunct, to the `gi`'s for all $1 < i < n$ as middle disjuncts, and to `gn` as the last disjunct of the disjunction. We will also use "later disjunct" to refer to any `gi` for $i > 1$.

A disjunction can have any determinism. The most general determinism is of course nondet, but if one of the disjuncts always has at least one solution, then the disjunction as a whole does too, so a disjunction can also be multi. And if the disjunction has no outputs (which happens frequently for disjunctions in the conditions of if-then-elses), then the disjunction as a whole cannot have more than one solution, which means that it will be either det or semidet, depending on whether it has an always-succeeding disjunct. (Typical programs do not contain det disjunctions, since they are equivalent to `true`.)

For our purposes, the important distinction is between nondet and multi disjunctions on the one hand, in which backtracking may reach a later disjunct from code executed outside the disjunction, *after* the success of a previous disjunct, and semidet and det disjunctions on the other hand, in which backtracking to a later disjunct is possible only from code *within* an earlier disjunct. Since we do not care about the minimum number of solutions of each disjunction, our support treats multi disjunctions the same as nondet ones and det disjunctions the same as semidet ones. In the following, we will therefore talk only about nondet and semidet disjunctions. We consider nondet disjunctions first, since they are more general.

Figure 8.2 shows in pseudo-code form the supporting code we added to a nondet disjunction. We insert code at the following points: (d1) which is the start of the first disjunct, (d2) which represents the start of every middle disjunct, and (d3) which is the start of the last disjunct. These code fragments communicate using shared data in what we call a *disj frame*. Each entry to a disjunction creates a new disj frame. Since multiple nested disjunctions can be active at the same

time, we link these frames together to form the *disj stack* (this is possible due to chronological backtracking). The disj stack is not a separate stack; we reserve space for its frames in the usual stacks used by the Mercury language implementation. We maintain a global pointer to the top disj frame on the disj stack.

```
...,
( (d1): start of the disjunction and also of the first disjunct
        (a) push a disj frame
        (b) save the global region sequence number
        (c) save region size records and their number
    g1
; ...
; (d2): start of a middle disjunct
        (a) do instant reclaiming of new regions
        (b) do instant reclaiming of allocations in old regions
    gi
; ...
; (d3): start of the last disjunct
        (a) do instant reclaiming of new regions
        (b) do instant reclaiming of allocations in old regions
        (c) pop the disj frame
    gn
), ...
```

Figure 8.2: RBMM runtime support for nondet disjunctions.

A disj frame has a fixed part and a nonfixed part. In Figure 8.3, the fixed part is the 4-slot box separated by a thick line from the nonfixed part. The four slots in the fixed part are:

- The `prev_disj_frame` slot holds the pointer to the previous disj frame, (or null if there is none).
- The `saved_seq_num` slot holds the value of the global region sequence number at the time when the disjunction was entered.
- The `num_prot_region` field gives the number of regions which are protected by a semidet disjunction (which we will discuss later). For a nondet disjunction, this slot will contain zero.
- The `num_size_rec` field gives the number of region size records saved in the nonfixed part.

8.3.1 Disj-Protecting Backward Live Regions

Consider a region which was created before the execution of a disjunction. Assume that this region is removed during forward execution, either by the code of a disjunct, or after the success of that disjunct by code following and outside

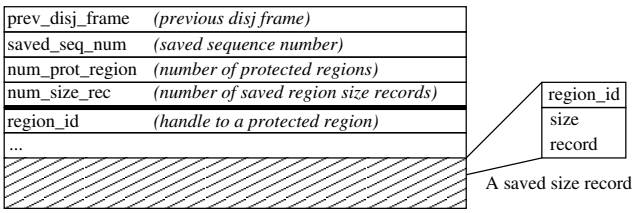


Figure 8.3: The structure of a disj frame.

the disjunction, but that this region is backward live with respect to a later disjunct of the disjunction. In this case, we need to make sure that if the region is removed during forward execution, it will not be actually reclaimed. Of course, the instruction that removes the region may not be reached because forward execution may fail before it gets there. But in general, we have to assume that the `remove` instruction *will* be executed, and that if the region may be needed after backtracking, we will need to prevent it from being reclaimed during the forward execution. We achieve this by *disj-protecting* such regions as follows. At the start of the disjunction, i.e., at (d1), we push a disj frame on the disj stack and save the current global sequence number into the `saved_seq_num` slot of the disj frame. A region is disj-protected by a disj frame if its sequence number is smaller than the sequence number saved at the disj frame. The `remove` instruction will only reclaim a region if the region is not disj-protected. There is an invariant that if a region is protected by a disj frame, it is also protected by all the later frames on the disj stack. This means that to check if a region is disj-protected or not, we only need to check if it is protected by the top disj frame.

The program will no longer backtrack into a disjunction after starting the execution of its last disjunct. This means that no regions need to be protected any more by this disjunction. Therefore, at the start of the last disjunct, i.e., at (d3), we disj-unprotect them by popping the disj frame. The regions which had previously been protected only by this disj frame will be reclaimed when execution reaches their `remove` instructions.

8.3.2 Instant Reclaiming of New Regions

When the program backtracks to a later disjunct, we want to reclaim all the regions that have been created during the computation that has just been backtracked over, i.e., all the regions that were created after entry to the disjunction. At (d1), we saved the global sequence number in the disj frame. Therefore at the start of a later disjunct of the disjunction, i.e., at (d2) and (d3), we just need to traverse the region list, and reclaim all the regions we see until we encounter a region whose sequence number indicates that it was created before the disj frame.

8.3.3 Instant Reclaiming of New Allocations in Old Regions

When arriving at a later disjunct, we want to restore all the regions that existed before the disjunction to the sizes they had when entering the disjunction, recovering any memory that has been allocated in them. To restore the size of a region, we need to save the region's size record in the nonfixed part of the disjunction's disj frame at (d1) so that we can restore the region's size at (d2) and (d3). We need three slots for each region: one for the region handle so that we know to which region the saved record belongs, and the other two for the record itself (see Figure 8.3). To be able to loop through the saved records and restore the regions at (d2) and (d3), we store the number of saved records in the fixed `num_size_rec` slot. The first saved record can be located by taking the address of the frame, and adding both the size of the fixed part and the number of slots for protected regions (which is zero in this case of nondet disjunctions).

The set of regions that existed before the disjunction and that may be allocated into by code following the disjunction is not available to the compiler. In theory, we could implement a global analysis to make it available, but such an analysis would be very complicated, especially for multi-module programs. Even if such an analysis existed, we would still have a big problem, which is that the number of regions in this set is not bounded, and in many cases the set would contain tens, hundreds or even thousands of regions. Saving and then restoring the sizes of that many regions can take a significant amount of both memory and time. We do not want this overhead to outweigh the benefits of instant reclaiming.

In our implementation, we have chosen to save and restore the sizes of only the regions that are *locally* forward live before the disjunction. (This information is readily available inside the Mercury compiler.) This may lead to some missed opportunities for recovering memory, but since nondet disjunctions are quite rare in most Mercury programs, we do not expect this to be too much of a problem. (We will see below that we do not miss memory recovery opportunities for semidet disjunctions.)

We save and restore the sizes of *all* regions that are locally forward live at the start of the disjunction (the number of these regions governs how much space we reserve for the nonfixed part of the disj frame). We save and restore the sizes even of regions that are never allocated into before backtracking, since, in the absence of the analysis mentioned above, we do not know which ones of those are. This may lead to some unnecessary saving and restoring, but since in typical programs the number of regions whose size we save and restore at a disjunction is usually relatively small, we do not expect the memory or runtime cost of these unnecessary saves and restores to be all that significant.

8.3.4 Specialized Treatment of Semidet Disjunctions

Because at most one disjunct of a semidet disjunction may succeed, when one of its disjuncts is reached, it means that all the previous disjuncts have failed and more importantly that the execution has not passed outside the disjunction's scope. Therefore, we only need to provide runtime support for a semidet disjunction if in its scope there is some change with respect to the set of existing regions. This basically means that the runtime support for nondet disjunctions described above will only be applied to semidet disjunctions whose created, removed and allocated sets are not all empty. In our practical experience with Mercury, most semidet disjunctions contain only tests, and rarely make changes to the heap. Therefore the support we describe below is needed only by a relatively small fraction of semidet disjunctions.

For a semidet disjunction, the Mercury compiler generates code such that when one of its non-last disjunct succeeds, the execution will commit to it and not go back to try any later disjuncts. This means the code we add at (d3) may not be reached after the success of a non-last disjunct, causing two problems. First, the disj frame will not be popped. Second, the regions which are protected by this disjunction will not be reclaimed at the start of the execution of the last disjunct as in the case of nondet disjunctions. Our solution is to do these two tasks at the end of any non-last disjuncts, i.e., after their success at (e1) and (e2) as in Figure 8.4.

To solve the first problem, we pop the frame at (e1.b) and (e2.b). To solve the second problem, at (d1) we loop through the regions in the disjunction's removed set. If a region is already protected, we do not want it to be reclaimed in the disjunction and its `remove` instructions inside the disjunction will be ineffective anyway, so we do not need to do anything. If a region is not already protected, we save its handle in the nonfixed part of the disj frame. At the end, we store the number of region handles we saved in the frame's `num_prot_region` slot. The code at (e1.a) and (e2.a) will loop through the saved handles, and reclaim all the saved regions (they were logically removed during the disjunct but their `remove` instructions were thwarted by the protection of this disjunction.)

At (d1.c), we save the sizes of only the regions in the disjunction's allocated set. Since execution cannot leave a semidet disjunction, we do not miss any memory recovery opportunities by restricting ourselves to these regions.

8.4 Support for if-then-else

The condition of an if-then-else (ite) can be either semidet or nondet. In most Mercury programs, the overwhelming majority are semidet, and this is the case we will look at first. Such if-then-elses share some properties with semidet disjunctions. If the condition succeeds, the execution will never enter the else part, and

```
...,
( (d1): start of the disjunction and of the first disjunct
        (a) push a disj frame
        (b) save the global region sequence number
        (c) save region size records and their number
        (d) save protected regions and their number
    g1
  (e1): end of the first disjunct
        (a) reclaim protected regions
        (b) pop the disj frame
; ...
; (d2): start of a middle disjunct
        (a) do instant reclaiming of new regions
        (b) do instant reclaiming of allocations in old regions
    gi
  (e2): end of a middle disjunct
        (a) reclaim protected regions
        (b) pop the disj frame
; ...
; (d3): start of the last disjunct
        (a) do instant reclaiming of new regions
        (b) do instant reclaiming of allocations in old regions
        (c) pop the disj frame
    gn
), ...
```

Figure 8.4: RBMM runtime support for semidet disjunction.

if the condition fails, the failure must have occurred in the scope of the condition.

Like disjunctions, if-then-elses need to protect regions from being reclaimed while backward live. But in the case of if-then-elses, we can restrict out attention to regions removed in the condition (i.e., in the condition's removed set), since this is the only part of the code in which the if-then-else itself can make a region backward live. When execution reaches the start of the then part, backtracking to the else part is no longer possible, which means that any regions that have been marked for removal in the condition have to be reclaimed for real, unless they are protected by a surrounding scope.

Also, if-then-elses, like disjunctions, should do instant reclaiming of memory allocated by backtracked-over computations. In the case of if-then-elses, this means that at the start of the else part, we should recover any memory allocated by the condition.

In general, we only need to provide support for changes to regions which occur inside the condition. This is good, because the conditions of if-then-elses are often very simple, containing only one or a few tests. Conditions whose created, removed and allocated sets are all empty are therefore fairly common. For such if-then-elses, the mechanisms we describe below are unnecessary, and so we optimize them away. If at least one of these three sets is not empty, we add code at the starts of the condition, the then part, and the else part, i.e., at points (i1), (i2), and (i3) in Figure 8.5.

```
( if
    (i1): start of the condition
        (a) push an ite frame
        (b) save the protected regions and their number
        (c) save size records and their number
    ...
  then
    (i2): start of the then part
        (a) reclaim the ite-protected regions
        (b) pop the ite frame
    ...
  else
    (i3): start of the else part
        (a) unprotect the ite-protected regions
        (b) do instant reclaiming of new regions
        (c) do instant reclaiming of allocations in old regions
        (d) pop the ite frame
    ...
)
```

Figure 8.5: RBMM runtime support for if-then-else with semidet condition.

For each if-then-else, we use a data structure called an *ite frame* to store the information used for its runtime support. As with disj frames, we embed ite frames in the ordinary stacks used by the Mercury implementation, and link them together into the *ite stack*, with a global variable pointing to its top. The structure of an ite frame is exactly analogous to that of a disj frame, the only difference being that the first slot of the fixed part, `prev_ite_frame`, holds a pointer to the previous ite frame (or null if there is none).

8.4.1 Ite-Protecting Backward Live Regions

Since the compiler knows the regions in the removed set of the condition (in our example in Figure8.1, `R1` is such a region), we will stop them from being reclaimed by *ite-protecting* them at the entry to the if-then-else. To allow us to ite-protect regions, we add to the region header a pointer field, `ite_protected`, which is set to null when a region is created. A region is ite-protected if its `ite_protected` field is not null. The `remove` instruction will now only reclaim a region if its `ite_protected` field is null and it is not disj-protected. (We do not use the same protection mechanism as in the case of disjunction. The reason for this will be explained when we describe how we handle if-then-elses with nondet conditions.) Before entering the condition, i.e., at (i1), we push an ite frame, and then iterate over the to-be-protected regions. If one of these regions is already protected for a surrounding disjunction or if-then-else, we ignore it. Otherwise, we protect it by setting its `ite_protected` field, which is currently null, to point to the ite frame. For such a protected region, we add its handle to a `region_id` slot in the nonfixed part of the ite frame. Then we also put the final number of regions we protect

in this way into the frame's `num_prot_region` slot. We do this so that we can loop over all the regions protected by this ite frame in two places: at the start of the then part (i2.a), where we reclaim all these regions (giving delayed effect to the `remove` instructions in the condition), and at the start of the else part (i3.a), where we undo their protection by resetting their `ite_protected` fields to null.

8.4.2 Instant Reclaiming

When the condition fails, we want to reclaim both the new regions created inside it and any new allocations into old regions. In our example in Figure 8.1 we want to reclaim all of `R3` and some of `R2`.

To reclaim new regions, at (i1.a) we save the current sequence number into the new frame's `saved_seq_num` slot, and at (i3.b), we add code that traverses the region list and reclaims all the regions until meeting an old region.

To reclaim new allocations into an old region, at (i1.c) we save its size record into the nonfixed part of the ite frame. Although it is reasonable to do this for the regions in the allocated set of the condition, it would be wasteful to reclaim new allocations into the regions which will be reclaimed right at the start of the else part. Unfortunately, while the compiler knows which old regions have `remove` instructions at the start of the else part, it does not know which of these will actually reclaim their regions, since it does not know which regions are protected by surrounding code. We handle this uncertainty as follows. We generate code at (i1.c) for every old region which is live at that point. For those that are not removed at the start of the else branch, this code always saves their size records unconditionally. For those that are removed at the start of the else branch, this code checks whether they are protected *before* this if-then-else, and saves their size records only if they are. This is an optimization because the test to see if a region is protected takes less time than saving its size record, and restoring it if the condition fails. We record the number of size records we saved in the `num_size_record` slot, so that code at (i3.c) can restore them all.

The final action of an if-then-else with a semidet condition is to pop the ite frame at either (i2.b) or (i3.d).

8.4.3 Handling if-then-elses with Nondet Conditions

Unlike Prolog, Mercury allows the condition of an if-then-else to have more than one solution. If the condition is nondet, then execution can backtrack into the condition from the then part or later code. This poses two problems we need to solve.

First, since the condition can succeed more than once, the code we add at the start of the then part (i2) can also be executed more than once. Because we need the ite frame every one of these times, we cannot let the code pop it at (i2.b); we must keep it until after the last time it may be used, i.e., after the last execution

of the condition. We arrange for this to happen by modifying the way the code generator handles the failure of the condition.

Normally, the code generator arranges for failures of the condition *before* the condition succeeds for the first time to cause a branch to the start of the else part, while a failure of the condition *after* it has succeeded represents a failure of the if-then-else as a whole, and will be handled accordingly, in whatever way the surrounding context demands. For example, if the if-then-else is one disjunct of a disjunction, its failure will cause execution to resume at the start of the next disjunct. We call the place to branch to on failure of the whole if-then-else the *failure continuation*.

We modified the code generator so that if the nondet condition needs support for region operations, i.e., it has a nonempty created set, removed set or allocated set, we branch to the failure continuation only after we execute code to pop the ite frame, the same code that for semidet conditions we would execute at (i2.b).

Second, the condition being nondet means that it must include, directly or indirectly, a nondet disjunction (since this is the only Mercury construct that can introduce nondeterminism). Therefore we must ensure that the supporting code fragments we generate for the if-then-else and the disjunction inside it do not step on each other's toes.

Our support for if-then-elses with semidet conditions provides ite-protection for regions in the condition's removed set that are not yet protected before the if-then-else. For such a region in a nondet condition, there are two cases. The first case is when the region is removed before the first nondet disjunction inside the condition. That means that when the `remove` instruction is executed, the region is ite-protected but not disj-protected. The `remove` instruction will (correctly) not reclaim it. Later on, the region will be reclaimed when the condition succeeds for the first time by the supporting code added at (i2). Because the program may backtrack into the condition and may reach the then part again, when the region is reclaimed at (i2.a), we need to nullify its entry in the ite frame so that it will not be wrongly reclaimed again the next time execution reaches (i2.a). This explains our saving of the pointer to the ite frame in the `ite_protected` field in the region header of a protected region.

In the second case, the region is removed after the start of the first disjunction in the condition, either in the disjunction itself or at some point after it. In an execution containing a non-last disjunct, when the `remove` instruction is encountered the region is not reclaimed because it is both ite- and disj-protected. We need to ensure that if the condition succeeds and execution reaches the then part, the region should not be reclaimed at (i2) because it may be needed when the execution goes back into the condition. We therefore put different code at (i2.a) if the condition is nondet; this code will reclaim a region only if it is not currently disj-protected (Figure 8.6). The region will remain both ite- and disj-protected until the execution enters the last disjunct, at that time it will lose its disj-protection

(Section 8.3). When the `remove` instruction in the condition is executed after this, it will not reclaim the region because it is still ite-protected, but the code at (i2.a) will reclaim it.

```
for each saved region_id
    if region_id != null && !is_disj_protected(region_id)
            reclaim the region;
            region_id  = null
```

Figure 8.6: Code at (i2.a) for if-then-else with nondet condition.

When the nondet condition fails, in both cases above, the region is only ite-protected, not disj-protected. It is because in the first case, the region is never disj-protected and in the second case, the failure happens only after all the disjuncts of the nondet code have been tried and failed, and the region has been disj-unprotected at the start of the last disjunct. This situation is exactly the same as when a semidet condition fails. Therefore the code at (i3) is exactly the same for nondet conditions as for semidet conditions.

8.5 Support for commit

When the goal inside a commit succeeds for the first time, we commit to that solution by discarding the inner goal's outstanding alternatives. We call the point in the code where this happens the *commit point*. If the inner goal is nondet (rather than multi), it may also fail. When it fails, the compiler's failure-handling mechanism causes execution to pass through a *failure point* before the program resumes forward execution at the resumption point of the next surrounding goal. The failure point is there to allow the execution of some cleanup code. We add code to support region operations at two or three points in Figure 8.7: the entry point of the commit (c1), the commit point (c2), and the failure point (c3); if the inside goal has determinism multi, we do not modify (c3) as execution will never reach there.

Consider a region that is in the removed set of a commit goal. If it is already protected by a disjunction or if-then-else when execution arrives at (c1), then the region should not be reclaimed by any code inside the commit, and the mechanisms we have described so far are sufficient to ensure this. If the region is not already protected at (c1), then the region should be reclaimed before execution reaches (c2). Ensuring this needs a new mechanism because the goal inside a commit will contain, directly or indirectly, at least one disjunction that can succeed more than once (if it did not, it would have at most one solution, and there would be no commit operation), and the runtime support for this disjunction will protect the region from being reclaimed during the execution of its non-last disjuncts. On the

```
some [...]
  (c1): entry to the commit
    (a) push a commit frame
    (b) save the sequence number
    (c) save the pointer to the top disj frame
    (d) save the to-be-reclaimed old regions and their number
  ( the inner goal )

  (c2): commit point
    (a) reclaim the saved old regions
    (b) reclaim the new regions
    (c) restore the state of the disj stack
    (d) pop the commit frame

  (c3): failure point
    (a) restore status of the saved regions
    (b) pop the commit frame
```

Figure 8.7: RBMM runtime support for commit.

other hand, we cannot simply insert code at (c2) to reclaim the region, since it *can* already be reclaimed by its `remove` instruction in the execution of the last disjunct before reaching (c2). We do not need to worry about the case when regions are protected only by semidet disjunctions or by if-then-elses inside a commit, since these constructs, if any, protect regions only temporarily, and ensure that any regions that are removed inside them and are not protected when the execution enters them will be reclaimed before the execution exits them.

As before, our solution involves a new embedded stack, the *commit stack*. We push a new commit frame at (c1), and fill in its fixed fields, which will be discussed later. Following this will be the code that, for each region in the removed set of the commit goal, checks whether the region is already protected. If it is, that region is left alone. If it is not, we add the handle of the region to the commit frame's nonfixed part, and record the address where this handle is stored in the commit frame in the region's own header, in a new field called `commit_slot`. This way, when a region that should be reclaimed inside the commit actually survives to (c2) due to the protection of an inner disjunction, code at (c2) can iterate through all the region handles in the commit frame and reclaim those regions. However, we cannot do this for regions that are actually reclaimed inside the commit (whose remove instructions were executed in the last disjuncts). That is why, when we reclaim a region, we check whether its header's `commit_slot` field is null. If not, then it will contain the address of a pointer to the region header from a commit frame and the reclaim operation will replace that pointer in the commit frame with a null. Making the loop at (c2.a) ignore such nulled-out region handle pointers ensures that each region recorded in the commit frame's list is reclaimed exactly once, and that this will happen as soon as possible.

If the goal inside the commit fails, we need to undo the update of the saved

regions' `commit_slot` fields, so at (c3.a) we reset them all to their original values. To make this possible, we save each original value in the commit frame next to the pointer to the region header from which it is taken. This effectively chains together all the entries referring to a given region in the commit stack. The reclaim operation will set to null not just the first slot in this chain, but all of them.

This mechanism is sufficient to correctly handle any old regions that are in the commit goal's removed set. To handle any new regions (regions created inside the commit) that are also removed inside the commit, we record the current region sequence number in the commit frame at (c1). When a new region is removed in the commit, if it is not protected, it is reclaimed. If it is protected, we mark it so that at the commit point we can reclaim it. We add a field `destroy_at_commit` to the region header, and we augment the `remove` instruction again so that when a protected, new region is removed in a commit, the `remove` instruction will set the region's `destroy_at_commit` field to true. At the (c2.b) part of the commit point, we traverse the region list until meeting an old region, and reclaim the new regions whose `destroy_at_commit` field is true.

We do not need to worry about instant reclaiming of new regions in the created set and of new allocations into regions in the allocated set of the commit, since that will be done by the construct(s) surrounding the commit.

At the commit point, the Mercury execution algorithm throws away all the remaining alternatives of the goal inside the commit. To reflect this, at (c2) we need to restore the embedded disj stack to the state it had at (c1). This is why at (c1), we save the current disj stack pointer in a fixed slot in the new commit frame, and at (c2), we restore the disj stack pointer from there. The regions protected by the disj frames thrown away by this action will be exactly the ones removed by the code at (c2.b).

The layout of commit frames is shown in Figure 8.8, with the fixed and nonfixed parts are separated by a thick line. The meaning of the first two fields should be

prev_commit_frame	*(previous commit frame)*
saved_seq_num	*(saved sequence number)*
saved_disj_sp	*(disj stack pointer)*
num_saved_region	*(number of saved regions)*
region_id	*(handle to a saved region)*
prev_commit_slot	*(original commit slot of the saved region)*
...	

Figure 8.8: The structure of a commit frame.

clear. The third field contains the value of the disj stack pointer at the time when the commit was entered. The last field gives the number of region handles and saved `commit_slot` fields actually stored by the code at (c1.d) in the nonfixed part.

8.6 Conclusion

In this chapter we have specified the design and implementation of the support needed for backtracking in the runtime system of Mercury. In general the support has two tasks. The first is to protect backward live regions and only reclaim them when they are *both* forward and backward dead. The second is instant reclaiming, namely reclaim new regions and new allocations into existing regions that have happened in the backtracked-over computation.

One challenge for the design is that we do not want to pay for the backtracking support when we do not actually need it. This is significant to the overall performance of the region-based system because only a small fraction of Mercury program code is actually nondeterministic. By making use of determinism information we have been able to minimize this cost for semidet disjunctions and if-then-elses. Our experimental results on runtime performance shown in Chapter 9 confirm this claim.

Chapter 9

Experimental Evaluation

We have implemented the region analysis and transformation, Chapters 4, 5, and 6, as well as the runtime support, Chapters 7 and 8 in the Melbourne Mercury compiler. The runtime support algorithms are implemented in the backend that generates low-level C code. In this chapter we first show the results of evaluating our region-based systems. We then provide an in-depth study of the relation between sharing and RBMM, which reveals opportunities for improvements.

9.1 The Experimental Systems

In our experiments we use three RBMM systems. The first one, *rbmm1*, is similar to the RBMM system in (Phan, Somogyi, and Janssens 2008) in which we do not track the regions that are allocated into. In rbmm1, while the region operations (Section 7) are implemented as C functions the runtime support for backtracking (Section 8) is implemented using C macros. The functionality of the second system, *rbmm2*, is exactly the same as that of the first one, however we consistently implement the whole runtime support in functions. The runtime support in the third system, *rbmm3*, is also coded in functions. Moreover it contains the tracing (Section 4.5) and use of allocated regions as presented in Chapter 8. Having these differences, the three systems still have the same usage of the regions they create and remove. However, they differ in other aspects such as compilation time, size of object files, the runtime support needed, and hence runtime performance. The idea of experimenting with rbmm1 and rbmm2 is to study the effects of using macros and functions. Comparing the experimental results of rbmm2 and rbmm3 can reveal the impact of tracing allocated regions. For all three RBMM systems, we use region pages of size 2,048 words, in which 2,047 are available to store program data. When needed, we request blocks of 100 region pages from the OS.

We also compare the RBMM systems with a Mercury compiler that is the

same as the RBMM systems in all aspects except that it uses the Boehm garbage collector (Boehm and Weiser 1988) instead of RBMM. We call this system *heap*.

The experiments were performed on a PC with two 2.8 GHz Pentium 4 CPUs, 1.5 GB of RAM, running Linux version 2.6.24-24-generic Ubuntu 4.2.4-1ubuntu4 SMP. The programs are compiled with the Mercury compilers described above and with the gcc 3.4.4.

Next, we will present the benchmarks and then we will give the results of our experiments. Finally we discuss the RBMM behaviour of the benchmarks in more detail as we are interested in their memory (i.e., the use of the regions), their runtimes and the kind of runtime support they need.

9.2 The Benchmark Programs

In the experiments, we used a set of relatively small benchmark programs. We made a careful selection of the benchmarks, which are actually more like a collection of case studies that we use to illustrate the strong and weak points of RBMM. When we do not give averages, it is because our benchmark collection contains several cases known to be difficult for RBMM in terms of memory reuse. While we would have liked to test our system with bigger, more realistic programs, we are currently not able to do so because the region analysis and transformation do not yet support higher order code and multi-module programs.

The benchmark programs in Table 9.1 are divided into three groups. The first group contains benchmarks that do not need any run-time support for backtracking. The ones of the second group do need backtracking support. The third group consists of manually modified versions of benchmarks that illustrate how programs can be made more region-friendly (hence the r as prefix of the names).

In the first group, we have included some deterministic programs that use some temporary data: **dna** computes subsequent similarities between sequences, **isort** implements an insertion sort algorithm and works on a list of 10000 integers, **nrev** reverses a list of 5000 integers, **primes** finds all the primes less than 20000, **qsort** sorts a list of 100000 integers. The program **dna** is a difficult case for RBMM (Makholm and Sagonas 2002; Phan and Janssens 2007). The benchmark **isort** was mentioned in (Aspinall, Hofmann, and Konečný 2008). It creates sharing between the input list and the sorted list. Therefore it is also difficult for RBMM.

The programs in the second group need runtime support for if-then-elses and/or disjunctions. **bigcatch** and **filrev** are Mercury versions of programs used in (Aspinall, Hofmann, and Konečný 2008). They manipulate lists of lists of integers and introduce sharing between the input, the temporary data and the output and as such they present difficult cases for RBMM. **bsolver** is a simple solver for systems of binary linear equations and inequations over integers; **boyer** is a toy theorem prover; **crypt** finds the unique answer to a cryptoarithmetic puzzle; **healthy** is a nondeterministic variant of **life** that searches for a generation that after a certain

Table 9.1: Information about the benchmarks.

	# Predicates	# LOC	if-then-else	disjunction	
				semidet	nondet
dna	16	168	x		
isort	6	84	x		
nrev	5	41	x		
primes	8	57	x		
qsort	6	54	x		
bigcatch	12	108	x		
boyer	17	304	x		
bsolver	41	531	x	x	
crypt	15	147	x		x
filrev	12	100	x		
healthy	24	335	x		x
life	18	232	x		
queens	9	90	x		x
sudoku	22	274	x		x
rdna	17	177	x		
risort	7	68	x		
rlife	19	300	x		
rqueens	10	96	x		x

number of reproductions (8) still has a number of live cells that is higher than a threshold (80); **life** implements the Game of Life (known to be a difficult case for RBMM); **queens** solves the 12-queen problem by first generating a permutation and then checking; **sudoku** finds the solution for a sudoku puzzle by doing propagation on finite domains.

The programs **rlife** and **rdna** are the versions of **life** and **dna** that have been manually made region-friendly as will be explained later by adding a copying predicate. **rqueens** is modified from **queens** by changing the `delete` predicate (called by `permute`) to copy the remaining list after deletion. Also **risort** copies the remaining list when inserting an element into a sorted list. We come back to this group of programs when discussing the benchmarks in detail.

9.3 Experimental Results

9.3.1 Compilation Times and Object File Sizes

We first compare the three RBMM systems and the heap system with respect to their compilation times and the sizes of their object files. The compilation time results in seconds are given in Table 9.2. There are always slowdowns during compilation with the RBMM systems compared to the heap system. The cost of

Table 9.2: Compilation time.

	heap	**rbmm1**	**rbmm2**	**rbmm3**
dna	0.86	0.99	0.97	0.99
isort	0.59	0.60	0.59	0.59
nrev	0.56	0.59	0.60	0.59
primes	0.57	0.58	0.59	0.60
qsort	0.59	0.64	0.64	0.66
bigcatch	0.66	0.72	0.72	0.76
boyer	1.25	2.26	2.10	2.04
bsolver	1.77	2.68	2.54	2.31
crypt	0.88	1.07	0.98	0.99
filrev	0.66	0.71	0.72	0.74
healthy	1.04	1.54	1.37	1.32
life	0.90	1.18	1.11	1.08
queens	0.61	0.75	0.72	0.73
sudoku	1.39	1.92	1.89	1.84
rdna	0.88	1.00	1.00	0.99
risort	0.60	0.67	0.66	0.65
rlife	0.92	1.21	1.14	1.10
rqueens	0.67	0.76	0.74	0.75

including RBMM into the programs is reasonable: on average the slowdown for rbmm1 is 23% and for the other two 20%. Compilation with the function-based system rbmm2 is most of the time faster than with the partly macro-based system rbmm1. (We subjectively consider only the differences large than 20 ms reliably slower or faster.) It is likely because the runtime support functions in rbmm2 have been compiled when the RBMM-enabled compiler was compiled while in rbmm1 the code from the macros is expanded and compiled when a benchmark program is compiled. Compared to rbmm2, tracing and making use of the allocated regions in rbmm3 sometimes help to reduce compilation time, but the effect is quite small. This can be explained by the fact that the overhead of tracking is rather small and the information about allocated regions often can lead to speedups because fewer region arguments are passed and less runtime support code is added.

The sizes in bytes of the object files (their text sections) are given in Table 9.3. The object files of the RBMM systems are, as expected, larger than those of the heap system. The use of macros in rbmm1 can double the size as for **bigcatch** and **healthy**. For programs in the first group as well as **rdna** and **risort**, those that do not need runtime support for backtracking there is no difference in object file sizes between rbmm1 and rbmm2. For the others rbmm2 produces objects with smaller sizes as macros are no longer used, notably in **healthy** the reduction with respect to rbmm1 is up to 25%. The function-based system with allocation tracking, rbmm3, gives rise to a further reduction as by keeping trace of the allocated

Table 9.3: Size of the object files.

	heap	**rbmm1**	**rbmm2**	**rbmm3**
dna	4,703	6,159	6,159	5,935
isort	920	1,160	1,160	1,160
nrev	880	1,168	1,168	1,168
primes	929	1,168	1,168	1,168
qsort	1,065	1,466	1,466	1,466
bigcatch	1,561	3,049	2,425	2,025
boyer	13,670	20,965	17,460	15,908
bsolver	15,800	26,424	23,338	19,434
crypt	5,560	9,296	6,960	6,912
filrev	1,490	2,465	2,209	2,049
healthy	7,837	16,109	11,759	10,269
life	5,492	9,203	7,843	6,867
queens	1,769	3,235	2,515	2,483
sudoku	7,586	12,226	11,586	10,722
rdna	4,752	6,304	6,304	6,080
risort	1,065	1,400	1,400	1,400
rlife	5,653	9,572	8,228	7,172
rqueens	2,058	3,573	2,869	2,837

regions we can reduce both the amount of supporting code for backtracking and the number of region arguments of procedures: **bsolver** and **filrev** benefit the most, while on average we gain 6% compared to rbmm2. The average increase for rbmm3 with respect to heap is 30% which is acceptable. This shows that for larger programs the systems with the functions are likely preferable.

9.3.2 Memory Usage

We measured the memory consumption of the regions for the RBMM systems. Note also that the runtime support consumes some memory as will be discussed later. Here we focus on the storage of program data. The results in Table 9.4 are the same in all three RBMM systems. For each benchmark, we give the total number of regions created during its execution, and the maximum number of regions coexisting during its run. We also include the total number of words allocated and the maximum number of words that coexist. *SLR* is the Size of the Largest Region and *S (%)* is the saving, calculated by 1 - Max words/Total words.

RBMM achieves optimum memory management in **nrev** (which reverses a list of 5000 integers), in **primes** (which finds all primes less than 20000), and in **qsort** (which sorts a list of 100000 integers). For nondeterministic programs, memory savings can still be high, such as in **crypt**, **healthy**, **queens**, and **sudoku**. The

Table 9.4: Memory use in the region-based systems.

	Regions		Words used		SLR	S (%)
	Total	Max	Total	Max		
dna	2,082,005	8	18,926,797	4,590,797	4,096,000	75.74
isort	2	1	67,029,000	67,009,222	67,009,222	0.03
nrev	5,002	2	25,015,000	10,000	10,000	99.96
primes	2,264	1	5,221,386	39,998	39,998	99.23
qsort	200,002	21	5,865,744	200,000	200,000	96.59
bigcatch	2	2	25,015,000	25,015,000	25,005,000	0.00
boyer	12	3	430,683	143,561	143,505	66.67
bsolver	95	8	2,914,534	2,911,528	2,908,442	0.10
crypt	416	3	3,442	94	64	97.27
filrev	5	3	25,023,004	25,019,000	25,009,000	0.02
healthy	3,917,123	82	62,639,310	2,794	2,054	99.99
life	50,303	102	894,336	8,208	6,486	99.08
queens	4,545,702	2	121,453,230	114	90	99.99
sudoku	6,650	88	84,080	16,678	10,916	80.16
rdna	2,083,005	9	18,930,797	501,752	428,733	97.35
risort	373,213	1	289,968,666	2,000	2,000	99.99
rlife	50,354	102	894,394	1,856	1,722	99.79
rqueens	23,080,415	13	142,047,288	156	24	99.99

impact of instant reclaiming on memory reuse is different for these nondeterministic programs (Table 9.5). In **crypt** and **queens**, instant reclaiming helps to collect most of the words while in **healthy** it contributes just a small part and it reclaims none in **sudoku**.

For cases such as **isort**, **bigcatch**, **bsolver** and **filrev** we see that most of the memory goes to the biggest region. Typically, this biggest region contains garbage data but, as it also holds some live data, it cannot be reclaimed.

The heap system uses the Boehm garbage collector (Boehm and Weiser 1988) for memory management. In our experiments, we just use the default configuration of this collector as it is in the Mercury compiler distribution. It is a stop-the-world, non-parallel mark-and-sweep collector that uses 1024-word pages. It starts with a heap of 64k words and heuristically carries out collections of garbage or expands the heap on demand. Data about memory use in the heap system is shown in Table 9.6. In the first column, the benchmarks in which the Boehm collector behaves the same are grouped into the same row. The second column (# gc) shows the numbers of times the collector is run while the third column (# expansions) tells the numbers of expansions of the heap. The maximal sizes of the heap in kB and words are shown in the last two columns, respectively.

The numbers show that, in almost all of the benchmarks, the RBMM systems can work within spaces that are smaller than requested by the collector. RBMM

Table 9.5: Words reclaimed thanks to runtime support. The other words are reclaimed by `remove` instructions. Only programs with some non-zero numbers are shown.

	New allocations		New regions		Start of then		Commit point	
	Words	%	Words	%	Words	%	Words	%
bigcatch	0	0.00	0	0.00	10,000	0.04	0	0.00
crypt	0	0.00	3,270	95.00	0	0.00	6	0.17
queens	12,356,378	10.17	109,096,776	89.83	52	0.00	0	0.00
rqueens	0	0.00	133,809,696	94.20	0	0.00	132	0.00
healthy	81,862	0.13	3,314	0.01	0	0.00	0	0.00
sudoku	0	0.00	0	0.00	0	0.00	6,480	7.71

Table 9.6: Memory use in the heap system.

	# gc	# expansions	max size (kB)	max size (words)
dna	6	4	30,524	7,814,144
isort	19	4	30,524	7,824,144
nrev	8	4	30,524	7,824,144
primes, qsort	2	4	30,524	7,824,144
bigcatch, filrev	4	10	119,804	30,669,824
boyer, life, rlife	1	3	22,892	5,860,352
bsolver	1	4	30,524	7,824,144
crypt, sudoku	0	2	17,168	4,395,008
healthy	18	4	30,524	7,824,144
queens	35	4	30,524	7,824,144
rdna	6	4	30,524	7,824,144
risort	82	4	30,524	7,824,144
rqueens	41	4	30,524	7,824,144

Table 9.7: Runtime performance.

	# Iter	heap			RBMM runtime			Saving
		runtime	gc time	# gc's	rbmm1	rbmm2	rbmm3	
dna	40	16.70	5.32	219	17.29	17.22	17.12	-2.51%
isort	20	27.52	8.82	380	12.65	12.54	12.55	54.40%
nrev	40	20.28	6.56	284	8.61	8.63	8.67	57.25%
primes	100	21.13	3.61	149	14.29	14.32	14.31	32.28%
qsort	80	13.80	4.08	140	7.89	7.94	7.95	42.39%
bigcatch	8	16.76	3.58	8	11.91	11.94	11.89	29.06%
boyer	2,000	14.86	2.07	89	14.57	19.83	19.85	-33.58%
bsolver	200	11.74	3.97	166	4.25	4.30	4.30	63.37%
crypt	100,000	16.70	2.31	98	14.66	15.97	16.11	3.53%
filrev	10	16.87	3.86	13	12.13	12.12	12.08	28.39%
healthy	4	7.97	1.64	72	10.73	16.30	6.46	18.95%
life	200	11.58	1.14	51	12.13	15.21	10.54	8.98%
queens	5	19.47	3.96	172	15.94	19.70	19.92	-2.31%
sudoku	5,000	9.68	2.36	103	8.28	8.23	7.93	18.08%
rdna	40	16.82	5.26	219	16.84	17.05	16.92	-0.59%
risort	8	47.60	15.10	656	16.01	16.79	16.03	66.32%
rlife	200	11.73	1.22	51	12.23	16.12	10.52	10.32%
rqueens	5	22.38	4.62	201	26.16	30.34	29.96	-33.87%

systems often need to request only 2048 x 100 words. The worst case for RBMM is **isort** where a large amount of garbage is produced and RBMM is not able to reuse memory efficiently. The heap system can work with ten times less memory in this case.

9.3.3 Runtime Performance

We also studied the runtime performance of our benchmark programs in the heap system and in the three RBMM systems because it is probably the most important criterion for the practicality of RBMM. To eliminate the uncertainty involved in measuring small times, we ran each program many times in a loop. Each benchmark has a row in Table 9.7 that gives the number of iterations, the actual execution times with the heap and with the three RBMM systems, the heap system's gc time (all in seconds, all for user mode only), the number of collections executed by the Boehm collector, and the savings achieved by using rbmm3 instead of the heap system (given by 1 - rbmm3 runtime / heap runtime). We present the savings only for the rbmm3 system because firstly it is fair to compare the heap system with an RBMM system that is also implemented using functions. Secondly, between the two function-based RBMM systems, rbmm3 should always be the faster because it contains more optimizations.

The rbmm3 system gets clearly better runtimes for 12 out of 18 benchmark programs including both deterministic and nondeterministic programs. The speedups range from around 10% to more than 60% compared to the heap system. This promising result can be explained by the fact that with RBMM we avoid the burden of runtime garbage collection and that the overhead of supporting regions is

Table 9.8: Runtime support information in rbmm1 and rbmm2 systems.

	Disj frames					Ite frames					
	Total	M	# Words	Mw	Sr	Total	M	# Words	Mw	Sr	P
bigcatch	0	0	0	0	0	5,045	1	35,499	11	5,091	46
boyer	0	0	0	0	0	115,884	2	814,392	14	116,952	0
bsolver	90	1	1,170	13	270	243	1	4,026	19	1,018	0
crypt	55	4	220	16	0	1	1	4	4	0	0
filrev	0	0	0	0	0	5,000	1	50,000	10	10,000	0
healthy	2,431	9	24,304	84	4,860	17,449,109	2	174,491,084	14	34,898,216	0
life	0	0	0	0	0	177,788	1	1,777,880	10	355,576	0
queens	12,356,498	12	86,495,486	84	12,356,498	1	1	5	5	0	1
sudoku	81	81	810	810	162	1	1	16	16	4	0
rlife	0	0	0	0	0	177,788	1	1,777,880	10	355,576	0
rqueens	12,356,498	12	86,495,486	84	12,356,498	1	1	5	5	0	1

Table 9.9: Runtime support information in rbmm3 system.

	Disj frames					Ite frames					
	Total	M	# Words	Mw	Sr	Total	M	# Words	Mw	Sr	P
bigcatch	0	0	0	0	0	46	1	230	5	0	46
boyer	0	0	0	0	0	114,813	1	803,682	7	114,810	0
bsolver	0	0	0	0	0	17	1	170	10	34	0
crypt	55	4	220	16	0	1	1	4	4	0	0
filrev	0	0	0	0	0	0	0	0	0	0	0
healthy	2,431	9	24,304	84	4,860	1	1	4	4	0	0
life	0	0	0	0	0	0	0	0	0	0	0
queens	12,356,498	12	86,495,486	84	12,356,498	1	1	5	5	0	1
sudoku	81	81	324	324	0	1	1	10	10	2	0
rlife	0	0	0	0	0	0	0	0	0	0	0
rqueens	12,356,498	12	86,495,486	84	12,356,498	1	1	5	5	0	1

reasonably modest. Moreover, the runtimes of eight out of these 12 programs are still smaller than the corresponding runtimes in the heap system excluding collection times, suggesting that RBMM may also provide better data locality. In **bigcatch** and **filrev**, two difficult cases for RBMM, their memory-use pattern actually has adverse effects on the operation of the runtime collector. They all build very large lists that are live data before producing any garbage. The collector tries to collect the heap when some allocations happen. This behaviour, in the situation where there is little garbage, causes the marking phase to take significant time. These facts explain the large speedups from around 30% to 60% in several programs.

Before discussing the results of the other programs we show the detailed information about the disj frames and the ite frames that are used in the benchmark programs in *one iteration* in Table 9.8 for rbmm1 and rbmm2 and in Table 9.9 for rbmm3.

The information about commit frames is left out of the tables because each nondeterministic program uses just one commit frame of four words and no dynamic information is saved. Only programs that do use runtime support for backtracking are included. The five columns related to disj frames are as follows. Total is the

total number of disj frames used in the whole run; M is the maximal number of disj frames coexisting at some point; the third one, # Words, is the total number of words used for all the disj frames; the fourth, Mw, is the maximal number of words used at some point; the last, Sr, is the total number of size records saved. No regions are protected at semidet disjunctions in these benchmarks. For ite frames, the first five columns have similar meanings to those for disj frames. The last is the total number of regions that are protected by the ite frames (i.e., their handles are saved to the frames). The Mw columns show that the memory used by these embedded frames is negligible in all benchmarks. All the *total* numbers here must be multiplied with the corresponding numbers of iterations in Table 9.7 to have the total numbers in the iterated programs used in the runtime experiment.

The rbmm3 system is only a little faster than the heap one in **crypt**. Despite being a nondeterministic program, the runtime support for backtracking it needs is rather cheap (see Table 9.9). However, we see that the program handles a rather large number of small regions, more than 41M regions. The largest region of these contains 64 words and on average it is only eight words per region. (The total number of regions can be computed by multiplying the number of iterations in Table 9.7 with the total number of regions in one iteration in Table 9.4). So the gain due to no runtime collection is largely damaged by the overhead of handling many regions, resulting in just a small overall speedup.

This adverse impact of handling many regions on performance shows as well in other programs where the numbers of regions are substantial, more than ten million. The overhead can be seen in the programs such as **qsort**, **healthy**, **life**, **sudoku**, and **rlife**, where we still have clear speedups but they are not as substantial as in those with much smaller numbers of regions. This should also be the reason for the small slowdowns in **dna** and **rdna**. These two programs do not suffer at all from any supporting code for backtracking, but all the gain is cancelled out. The better performance of **rdna** compared to **dna** is because it can reuse memory better with the help of the copying predicate. In this program, the overhead of copying is rather low because it is just copying blocks of four words one thousand times. The largest region in **rdna** does not need to be expanded as much as the one in **dna**, thus the gain in speed. Compared to **crypt**, **queens** is also nondeterministic but the runtime support in rbmm3 for it is rather costly (again see Table 9.9). So besides the cost of handling many regions (22.5M) it also has to pay for the runtime support for nondet disjunctions. The combined cost results in the decrease in performance.

The two worst cases for rbmm3 are **boyer** and **rqueens**. For **rqueens** the number of regions is excessively large, five time more than that in **queens**. This can be seen as the negative side-effect of the copying predicate whose purpose is to help distribute terms better. It does fulfill the purpose as we can see in Table 9.5 compared to **queens** almost all the words are now reused due to the reclamation of new regions. We would have seen a better runtime if we had been able to

disable the saving of size records at disj frames, which becomes useless while is very expensive. The slowdown in **boyer** is mainly due to the cost of saving size records (229.62M) at ite frames, which is also in vain. A closer look at **boyer** reveals that it contains some semidet procedures that allocate into their input regions. They are called in the conditions of if-then-elses. So the saving of size records is needed if we want to have instant reclaiming. However for the specific input used in our benchmark, the calls to such semidet predicates all succeed so no words are (need to be) reclaimed.

Comparing the runtime results in rbmm2 and in rbmm3 gives us an idea about the usefulness of tracking allocated regions. While the reduction in the number of region arguments does not have a strong impact in these benchmarks, having less supporting code for backtracking shows marked speedups in **healthy** and **life** (**rlife** also) in rbmm3. This enhanced performance corresponds with the reductions in Table 9.9 compared to Table 9.8. We can see that the main impact is on the ite frames. For **filrev** and **life** we can get rid of them completely. For some others we no longer have to save any size records to ite frames. This is very important because while nondet code is the exception (because in practice almost all Mercury predicates (98+%) have at most one solution), if-then-elses are very popular in Mercury programs. Ensuring the efficiency of this specific construct is vital to the efficiency of the Mercury language in general. However, tracking of allocated regions cannot help in all cases, such as in the case of **boyer**. In that case the trade-off between reclaiming the words allocated in conditions of if-then-elses and efficiency would be worth considering.

We see that in the programs where the runtime support for backtracking is needed using macros to implement the support may improve performance. Table 9.7 shows that **boyer**, **crypt**, **healthy**, **life**, and **queens** (also **rlife**, **rqueens**) are faster in rbmm1 than they are in rbmm2. It is because using macros can avoid the cost of calling functions. More importantly, because these programs are small using macros, which increases code size, likely does not adversely affect instruction cache behaviour. However, for larger programs we conjecture that this will be the case.

9.4 When is It Hard to Reuse Memory in RBMM?

It is arguable that sharing is the most basic and natural form of memory reuse. However, sharing seems to conflict with RBMM because in RBMM we want terms to be separated in different regions. In this section we study in detail some benchmark programs, which are selected on purpose, to have a clear view on the impact of sharing on RBMM. Some of them are known difficult cases for RBMM such as **dna** and **life**. Some others create sharing that make it hard for in-place updating such as **isort**, **bigcatch**, and **filrev** (Aspinall, Hofmann, and Konečný 2008).

In our region points-to analysis we essentially put two program variables into

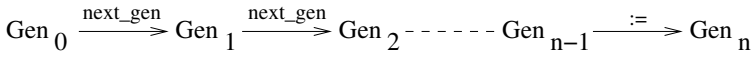


Figure 9.1: The computation of generations in **life**.

the same region in two cases: when there is an assignment between them or when they are bound to a term and its same-type subterm in a recursive data structure (Section 4). If the variables in a region have different lifetimes we will have a memory leak because we cannot reclaim in time the memory of the ones with shorter lifetimes but we will have to wait until the longest lived one is dead to destroy the whole region.

One solution for this is to, at a certain point, copy the live data to a different region so that the rest can be reclaimed. We experiment with this approach in **rdna**, **rlife**, **risort**, and **rqueens**.

The **life** benchmark encodes the Game of Life in which a new generation is generated from a previous one based on a set of production rules. From an initial generation it uses a loop (in the `life` predicate) to produce several intermediate ones before reaching the final generation, which is the wanted output. We represent a generation by a list of *live* cells, each cell is represented by its row and column in a 20x20 board. To store a generation we need two regions, one for the skeleton and the other for the cells. In the program the list skeletons of two successive generations are independent while their cells may share. In the recursive case of the predicate `life`, we first call `next_gen` to compute a new intermediate generation, whose skeleton could be in a different region, and then we call `life` recursively with the new intermediate generation as input. While in its base case, we assign the last intermediate generation to the output generation. The computation is summarized in Figure 9.1. Due to the assignment in the base case, which *only* creates the sharing between the last intermediate generation and the output generation, our region points-to analysis decides that the skeletons of the input and output generations in the `life` predicate are in the same region and then enforces this for *all* the (recursive) calls to `life`. This eventually means that the skeletons of *all* the generations are placed in one big region with a size of 6,486 words. In **rlife** we replace the assignment in the base case with a copying predicate that does not create the sharing, thus helping to separate the skeleton of each generation in a separate region, which then can be reclaimed in time. We see in Table 9.4 that, compared to **life**, the maximal number of words used in **rlife**, is reduced by 77% to 1,856 words. This is because the garbage consisting of the skeletons of the old generations and of temporary data created during the process of generating a new one is timely reclaimed.

The program **dna** simulates the matching of a given DNA sequence to each of the DNA sequences in a predefined set. The matching degree of two sequences is represented by a similarity, which is computed based on the similarities of their

elements with respect to the spatial relation among them. The resulting similarities (between two sequences) are calculated one by one and put in an ordered tree, which is a recursive data structure. To store a tree we need two regions, one for its left and right skeletons and the other for its nodes where all of the similarities are stored. Other than that, in this program, there are assignments in several predicates that establish sharing among the similarities in a way such that all the ever-computed similarities end up in the same large region of 4M words. (This is also the region of the nodes of the tree.) The maximal number of words in use during a run of the program is about 4.6M. In the so-called region-friendly version **rdna**, a similarity between two sequences is copied, just before it is added to the tree. By doing this the region analysis can decide that the region to which the similarity is copied is the region of the nodes of the tree and it can reclaim its previous region containing all the temporary similarities involving in its computation. The size of the largest region now drops to 0.43M words. The biggest region of **rdna** is the region of the skeleton of the tree.

In Chapter 10, we propose a more desirable solution, a more refined region analysis that, by taking into account different execution paths, can keep apart the regions of the variables in an assignment. An implementation of the improved analysis should achieve the same effect as in **rlife** and **rdna** while it neither requires manually rewriting programs nor incurs the cost of copying. The evidence supporting this can be found in Section 10.6.1.

The problems with memory reuse in RBMM in **isort** and **queens** are typical for programs that use recursive data structures and continuously update them, such as adding/deleting an element to/from a list or tree. Because an updated structure normally shares some part(s) with the original one they are stored in the same regions. Therefore we cannot reclaim the obsolete parts of the original structure. In **risort** and **rqueens**, we try to have better memory reuse by adding a predicate to copy the modified structure so that the original region can be reclaimed after the copying. In **risort** the copying happens after an integer is inserted and in **rqueens** after a queen is deleted. With this manual modification, in **risort** where a list of 1,000 integers is sorted, we obtain optimal memory management (see Table 9.4). In **rqueens**, compared to **queens**, the maximal number of words used is larger. We guess that this is due to region protection when there are some disj-protected regions that are removed but not reclaimed and instant reclaiming has not yet occurred. However, the size of largest region drops to 24 words, which is the storage needed to represent a list of 12 queens. While memory reuse can be improved by this copying approach the runtime overhead incurred is very expensive. We see a 50% increase of runtime for **rqueens** in Table 9.7 and for **risort** we have to reduce the input size ten times (from 10,000 integers in **isort**) so that the program can finish in reasonable time. Similar problems with memory reuse in the presence of recursive data structures can also be seen in **dna** and **rdna** where similarities are inserted to trees; and in **bsolver** where the domains

of the integral variables, which are lists of integers, are reduced.

The problem in **bigcatch** and **filrev** is also related to recursive data structures. In this case the structures are not updated but only part of them is used, i.e., is live data. Because of that we have to keep the whole region alive. We believe that this should also be overcome by the copying approach. We have no automatic solution for the problems related to the use of recursive data structures yet.

9.5 Conclusion

The experimental results show that the cost of having region-based memory management in Mercury programs is bearable. The region analyses, which have not been tuned, increase compilation time about 20%. Including the extra code for region-based memory management also increases code size of object files of around 30%.

Region-based systems often achieve good memory reuse, in some cases we even witness optimal memory use. Our in-depth study shows that programs in which recursive data structures are updated make it difficult for region-based memory management to reuse memory. In these cases combining region-based memory management with compile-time or runtime garbage collection would be a good idea.

The most practical experimental system, rbmm3, shows 20% speedup on average, which is very promising. The runtime results for programs containing nondet code give evidence that the overhead to support backtracking is rather modest. The huge speedups, up to 65%, in some programs seem to suggest that RBMM may offer good data locality. In some benchmark programs, such as **bigcatch** and **bsolver**, we experience that, in region-based systems, even when RBMM cannot help with reusing memory at all, we still have better runtime performance compared to using runtime garbage collection.

Chapter 10

Region Reuse

In Section 9.4 we mentioned the imprecision caused by the handling of assignments in the region points-to analysis. In the view of assigning variables to regions, we can say that the region points-to analysis (Chapter 4) sometimes too eagerly groups variables into regions without taking into account different execution paths, reducing reuse opportunities. In this chapter we describe an improvement of the region analysis and transformation that have been described in Chapters 4, 5, and 6. By distinguishing execution paths we obtain a more precise region allocation that ultimately leads to better memory reuse. To model the memory effects of different execution paths we need to keep the regions apart, which is achieved by extending region points-to graphs with same-edges.

10.1 Motivation

As the imprecision that we try to remove is related to assignments, we will look at their relation with regions in particular. Consider the assignment `X := Y`, which binds the free variable `X` to the value of `Y`, which, for simplicity, is assumably stored in one region. As after the assignment `X` and `Y` are bound to the same value in a region, we can say that the two variables are in the same region.

The code in Figure 10.1 is a part of a Mercury program that manipulates lists of integers. As normal, we associate a program point with every literal in the body of a procedure. The procedure `q` has two execution paths: $\langle(1),(2)\rangle$ and $\langle(3),(4)\rangle$ in which different assignments for `Z` occur. Note that the program point (1) does not belong to the second execution path because for an if-then-else *if* C *then* T *else* E Mercury will first try C but if C fails, it will ensure that E is executed as if C had never been tried. In the second execution path `q` invokes `produce` that can assumably put its output argument `Y` in a region different from that of `X`. The code of `produce` in Figure 10.1 follows this pattern. If after a call

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

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

Figure 10.1: The running example for region reuse.

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

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

Figure 10.2: Region-annotated version with the eager approach.

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

The region points-to analysis in Chapter 4 follows this "eager" approach. The resulting annotated code for `q` and `produce` is in Figure 10.2. So if the program follows the second execution path of `q`, the eager approach prohibits the reclamation of the memory of `X`, even when it is dead after the call to `produce` (recall that we assume that `X` and `Y` are independent). The memory for `X` could have been reclaimed if it had been kept apart in a different region from that of `Y` and `Z`.

Let us explore the idea of *keeping apart* the regions for `X`, `Y` and `Z` in order to be able to reclaim memory better. Thus we would like to keep `X`, `Y` and `Z` in different regions. An assignment like `Z := X` could then involve the copying of the value

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

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

Figure 10.3: Region-annotated version with region reuse.

of `X` into the region of `Z`. Although this allows us to reclaim the memory of `X` in the else branch it incurs the overhead due to the copying, which is not desirable as it is linear to the size of the value. We can do better by *reusing* the region of `X` as the region of `Z` at (2) and the region of `Y` as the region of `Z` at (4), while the memory of `X` is reclaimed in `produce` as shown in the region-annotated code in Figure 10.3. We can view this as the result of distinguishing execution paths of a procedure when dividing its variables into regions. In the first execution path, `X` and `Z` are in the same region while in the second one, `X` is in a region different from that of `Y` and `Z`.

In the region-annotated version, other than the region annotations that we have seen so far, the special region-annotated assignment notation `X@R1 :== Y@R2` means that `R1` is bound to the region of `R2`, then `Y` is assigned to `X`, after that `R2` is unbound. Its effect is that we reuse the region of `R2` for `R1`. Then the behaviour of the code is as follows. A caller of `q` prepares `X` in the region bound to by `R1` and expects `Z` in some region bound to by `R3`. If the first execution path of `q` is taken `R3` will be bound to the region `R1`. If the second path is taken, the call to `produce` removes `R1` and creates `R2` in its base case, then `R3` is bound to `R2`.

10.2 Region Points-to Graph with Same-Edges

We have been using a region points-to graph to model the memory used by a Mercury procedure. The graph represents both locations of the related terms and the sharing of among them. One of the contributions of this chapter is a refinement of the modelling of this sharing.

In Mercury, the instantiation of variables, therefore the sharing among them, is caused by unifications. For the purpose of representing sharing more precisely, we divide the sharing into two groups. First, a construction unification

`X <= f(..., Y, ...)` allocates new memory for storing the functor `f` (actually the block of memory words corresponding to `f`) and creates sharing between `X` and `Y`. Also in a deconstruction unification `X => f(..., Y, ...)` `Y` is instantiated and `Y` shares with a subterm of `X`. The regions needed to store the values of `X` are given by its type-based region graph in which the edges point into the regions of subterms. This kind of sharing is modelled in the same way as before. That is the sharing between `X` and `Y`, with `Y` the i^{th} argument of `X`, is represented by having a node n_X with $X \in vars(n_X)$, an edge $(n_X, (f, i), n_Y)$ and another node n_Y with $Y \in vars(n_Y)$. Recall that the *vars* set of a node can contain either zero, one or more variables. In the case where constructions and deconstructions involve variables of *recursive* types such as lists, e.g., `L <= [E | T]` or `L => [E | T]`, `L` and `T` are forced to end up in the same *vars* set. All the program variables in the *vars* set of a node may be allocated in the same region.

Second, an assignment unification `X := Y` binds `X` to `Y` and creates sharing between `X` and `Y`. Previously (i.e., in Chapter 4), we did put the two variables in the same region with the justification being that they actually point to the same term after the assignment. In the new approach in this chapter we keep their regions apart and just remember that they are candidates for reusing one for the other after this point. We represent this in a region points-to graph by a new kind of edges, called *same-edges*. The sharing created by `X := Y` is modelled by a *directed* edge $s(n_Y, i, n_X)$ with i the program point where `X := Y` is found. Note that the same-edge between n_X and n_Y should propagate to the regions of their corresponding subterms.

A **region points-to graph** $G(N, E, S)$ for a set of variables V consists of a set of nodes, N, representing region variables, a set of directed edges, E, representing references between the regions bound to by these region variables and a set of directed same-edges, S, to model candidates for reuse. Whether the reuse can safely be done depends on the liveness of the involved regions. This will become clear later.

As before, the nodes are annotated with *vars* sets: we have $V = \bigcup_{n \in N} vars(n)$. The *vars* sets are disjoint. The function $node(n_X, (f, i))$ returns the node m if $(n_X, (f, i), m) \in E$ otherwise it is undefined.

The region points-to graphs of the procedures `q` and `produce` in our running example are in Figure 10.4 in which same-edges are the "dashed" ones. With same-edges we can keep the regions of `X`, `Y`, and `Z` separate, in contrast to the eager approach where we would have only one node with these variables in its *vars* set. Now we are still able to capture the fact that at some program points (i.e., at (2) and (4)) they are the same. Note again that we assume no regions for the integer elements of the list because they are stored right in the first word of a *cons* cell.

All in all, *sharing* is represented in a region points-to graph in three ways. Firstly, the directed edges in E represent sharing of subterms. Secondly, that a

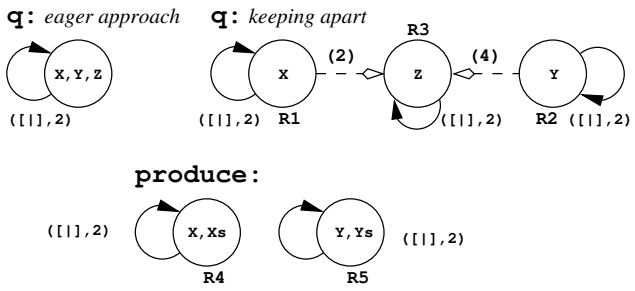


Figure 10.4: Region points-to graphs of `q` and `produce`.

vars set of a node may contain more than one variable represents the fact that these variables may be bound to the same term or to different subterms of a recursive term. Finally, sharing due to assignments is represented by the same-edges in S.

10.3 Region Points-to Analysis

The region points-to analysis computes a region model for a procedure and the whole program by capturing the locations and the sharing among variables. All the algorithms are modified to work with same-edges. Other than that they are rather similar to the ones in Chapter 4. We give the complete algorithms here and mark the notable additions for the ease of reading. To capture sharing now we use two operations: *unify* and *same_edge*. The modified operation *unify*, Algorithm 8, is extended to merge also the same-edges. Unifying two nodes n and m implies that the variables in $vars(n)$ and those in $vars(m)$ are stored in the same region. To ensure that there is only one out-edge with a specific label from one node to another the operation is again recursive, i.e., unifying two nodes may cause more nodes to be unified.

The novelty here (compared to Chapter 4) is the recording of sharing by using the *same_edge* operation that is defined by Algorithm 9. When we record a same-edge between two nodes we also need to recursively record same-edges for the corresponding nodes reached from them through corresponding edges. Recall that by $s(k, i, k')$ we mean to reuse the region bound to by the region variable represented by k for the region variable represented by k' at the program point i.

10.3.1 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 10. The function $pp(l)$ returns the program point associated to the literal l.

Algorithm 8 *unify*(n, m)

Require: $G(N, E, S)$, $n, m \in N$.
Ensure: $G(N, E, S)$ with n representing the unified node.

```
N = N \ {m}
vars(n) = vars(n) ∪ vars(m)
for all (m, (f, i), k) ∈ E do
    E = E \ {(m, (f, i), k)}
    if (n, (f, i), k) ∉ E then
        E = E ∪ {(n, (f, i), k)}
    end if
end for
for all (k, (f, i), m) ∈ E do
    E = E \ {(k, (f, i), m)}
    if (k, (f, i), n) ∉ E then
        E = E ∪ {(k, (f, i), n)}
    end if
end for

% Merging the related same-edges.
for all s(m, i, k) ∈ S do
    S = S \ {s(m, i, k)}
    if s(n, i, k) ∉ S then
        S = S ∪ {s(n, i, k)}
    end if
end for
for all s(k, i, m) ∈ S do
    S = S \ {s(k, i, m)}
    if s(k, i, n) ∉ S then
        S = S ∪ {s(k, i, n)}
    end if
end for
for all l, l' ∈ N do
    if (n, (g, j), l) ∈ E ∧ (n, (g, j), l') ∈ E ∧ l ≠ l' then
        unify(l, l')
    end if
end for
```

Algorithm 9 *same_edge*(n, m, i)

Require: $G(N, E, S)$, $n, m \in N$.
Ensure: $G(N, E, S)$ with same-edges between n and m and between any two corresponding nodes reachable from them.

```
S = S ∪ {s(n, i, m)}
for all (n, (f, i), k) ∈ E ∧ (m, (f, i), k') ∈ E do
    if k ≠ k' ∧ s(k, i, k') ∉ S then
        same_edge(k, k', i)
    end if
end for
```

Algorithm 10 *intraproc*(p): intraprocedural analysis of a procedure p

Require: p is in superhomogeneous form.
Ensure: Sharing created by explicit unifications is represented in G_p.

```
G_p = (∅, ∅, ∅)
for all X ∈ p do
    G_p = G_p ⊎ init_rptg(X)
end for
for all unif ∈ p do
    if unif ≡ (X := Y) then
        % Recording same-edge at an assignment.
        same_edge(n_Y, n_X, pp(unif))
    else if unif ≡ (X => f(Y_1, ..., Y_n) or X <= f(Y_1, ..., Y_n)) then
        for i = 1 to n do
            unify(node(n_X, (f, i)), n_{Y_i})
        end for
    end if
end for
```

Here the *init_rptg*(X) function, which generates a region points-to graph for X from the type-based region graph of the type of X, does one simple extra thing compared to the one in Algorithm 2, namely it also initializes the set of same-edges to empty. The intraprocedural analysis adds all the sharing created by the unifications in the procedure to G_p. We ignore test unifications because they do not create any sharing. For construction and deconstruction unifications we unify the nodes corresponding with the sharing created by them. For an assignment we say that at its program point the two variables are bound to the same term by adding a same-edge from the node of the right-hand side variable to the node of the left-hand side one. This leaves the possibility that in other execution paths they are not necessarily bound to the same term, therefore they do not necessarily have to be in the same region either.

10.3.2 Interprocedural Analysis

The interprocedural analysis, Algorithm 11, is almost the same as Algorithm 3. It just builds the α functions and then integrates the *relevant* sharing information from the region points-to graphs of the called procedures into G_p. Moreover, Algorithm 11 has to export the same-edges by relying on the α function at the call site. The program point of the same-edges is the program point of the call.

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

The result of the region points-to analysis for the example in Figure 10.1 has been shown in Figure 10.4. In the graph for `produce`, that `X` and `Xs` are in *vars*(*R4*) and that `Y` and `Ys` are in *vars*(*R5*) are due to the deconstruction at (3) and the construction at (5), respectively. Analysing the recursive call at (4) records $\alpha(R4) = R4$ and $\alpha(R5) = R5$ but makes no change to the shape of the graph. In the graph for `q`, the two same-edges are due to the assignments at (2) and (4). Processing the call to `produce` at (3) creates $\alpha(R4) = R1$ and $\alpha(R5) = R2$.

10.4 Good Same-Edges

The same-edges in our new region points-to graphs indicate regions that are candidates for reuse. For a procedure `q`, we keep a same-edge $s(R1, i, R2)$ with the intention that at the program point i we should *reuse* the region bound to by `R1` for `R2`, i.e., we make `R2` bound to the region currently bound to by `R1` and `R1` is considered unbound after that. However, it is only *safe* to do that if `R2` is not yet bound before i, if `R1` is not live after i, and if `q` is allowed to manipulate them.

Algorithm 11 *interproc*(p): interprocedural analysis of a procedure p

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

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

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

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

        % Integrate sharing in G_q into G_p.
        In the graph G_q, start from each n_{X_i}, follow each edge once and apply the rules P1 and P2 in Figure 4.2 when applicable.
        for all s(n_q, _, m_q) ∈ S_q do
            if α(n_q) = n_p ∧ α(m_q) = m_p ∧ n_p ≠ m_p then
                % Exporting same-edge relation at this call site.
                same_edge(n_p, m_p, i_p)
            end if
        end for
    end for
until There is neither change in G_p nor in any of the α functions.
```

So to decide if the region reuse implied by a same-edge is safe or not we need the liveness information of the involved region variables.

10.4.1 Region Liveness Information

In Section 5.2 we compute the region variables that are live before and after the program points in a procedure. This can be considered the local liveness of the region variables. Then in Section 5.3 we decide on the global liveness of the region arguments of a procedure, which is expressed by the *bornR*, *deadR*, and *outlivedR* sets. These computations are independent of same-edges therefore we still can use the algorithms in Sections 5.2 and 5.3 to compute the region liveness information.

For our example in Figure 10.1, based on the region points-to graph in Figure 10.4, we can derive the region liveness information for `produce` as follows. The local information is trivial in this example: in the first execution path $\langle(1),(2)\rangle$, `R4` is live before (1) and dead after that; `R5` is not live before (2) and is life after that. In the second execution path $\langle(3),(4),(5)\rangle$, `R4` is live from before (3) to before (4) and dead after that; `R5` becomes live at (4) and stays live until the end. Globally, `produce` is called in two calling contexts, We will look at how its *bornR* and *deadR* sets are computed. Initially, $bornR(produce) = \{R5\}$ because `R5` is reached from the output argument `Y` and $deadR(produce) = \{R4\}$ because `R4` is reached from the input argument `X`. The first calling context is at (4) in `produce`, where $\alpha(R4) = R4$ and $\alpha(R5) = R5$. It should be straightforward to see that `R4`, which is reached by `X` and `Xs`, is live before the call and is no longer live after it. So it is safe that `R4` ceases to be live in this call. For `R5`, which is reached from `Y` and `Ys`, it is not yet live before (4) and is live after. So `R5` can become live in this call. The other calling context of `produce` is at (3) in `q`. Here we have $\alpha(R4) = R1$ and $\alpha(R5) = R2$. Again we can see that `R1` is live before and not live after (3), meaning that it can become dead in the call. `R2` is not live before and is live after (3) so it can become live in this call. Therefore finally, we have $bornR(produce) = \{R5\}$ and $deadR(produce) = \{R4\}$.

10.4.2 Safeness Conditions for Same-Edges

We define the *safeness conditions* for a same-edge $s(R1, i, R2)$ in terms of the liveness information of the region variables as follows.

Definition 10.1 (Safeness conditions) *A same-edge $s(R1, i, R2)$ in the region points-to graph of a procedure* `q` *is good if the following conditions are all satisfied.*

1. $R1 \in LR_{before}(i) \setminus LR_{after}(i)$,
2. $R2 \in LR_{after}(i) \setminus LR_{before}(i)$,
3. $R1 \in deadR(q) \cup localR(q)$,

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

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

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

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

Figure 10.5: The running example extended with calling contexts of `q`.

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

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

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

Figure 10.6: Reuse region-annotated version for calling context 1.

4. $R2 \in bornR(q) \cup localR(q)$. □

The first two conditions can be seen as the local liveness requirements, while the global liveness requirements are in the last two. We call the same-edges that satisfy the above conditions the *good* ones. Otherwise there is no point keeping them and their regions can be unified after all.

In Figure 10.5 we extend the running example with two calling contexts for `q` to show the effect of good and bad same-edges.

In the first calling context, `X` is no longer used after the call to `q`. So we have $deadR(q) = \{R1\}$, $bornR(q) = \{R3\}$, and $localR(q) = \{R2\}$ (the reader may want to consult Figure 10.4 for the region variables). For `produce`, $deadR(produce) = \{R4\}$ and $bornR(q) = \{R5\}$. This program can be annotated as in Figure 10.6. Its behaviour is that the region of `X`, namely `R6`, is removed in the call to `q`. This is safe because `X` is no longer live after the call. `R7` is bound either to the region `R6` or to a new region created by the call to `produce` in `q` depending on which execution path is taken in `q`. But this does not matter because from outside of `q` we can assume that `R6` is dead and `R7` becomes live. And we only need to reclaim `R7` afterwards.

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

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

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

Figure 10.7: Wrong reuse region-annotated version for calling context 2.

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

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

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

Figure 10.8: Region-annotated version with the eager approach (with `main`).

In the second calling context, because `X` is still live after the call to `q`, the call is no longer allowed to remove `R6`. This means that `R1` is eliminated from $deadR(q)$ (i.e., `R1` is now in $outlivedR(q)$) and therefore `R4` is excluded from $deadR(produce)$. The annotated version now is in Figure 10.7. In `produce` there is no `remove(R4)` after (1) and in `main` `remove(R6)` is added after (4). If the program follows the first execution path in `q`, `R6` and `R7` are bound to the same region. Therefore that region will be wrongly removed twice in `main`. In general, we cannot guarantee which execution path is taken at runtime. Therefore in this case it is not safe to make use of the (bad) same-edges.

A safe way to handle this situation is to force `X`, `Y`, and `Z` in `q` into the same region as in the eager approach in Chapter 4. The annotated program is in Figure 10.8. (The annotated code for `q` and `produce` has been shown in Figure 10.2, we repeat it here with the annotated version of `main` for the ease of reading.) If we used the eager version for the first calling context, we would miss the chance to reuse the memory of `X` whenever the second execution path of `q` is taken.

So after the region liveness analysis we will re-examine the region points-to graphs to eliminate the bad same-edges using the safeness conditions of Definition 10.1. If some same-edges are excluded from the region points-to graph of a procedure q, it means that the region points-to graph of q changes. The change will affect the region points-to graphs of the callers of q and the α mappings at

the sites where q is called. Therefore we need to re-run the interprocedural region points-to analysis for the callers of q, i.e., all procedures in the same SCC as q and the ones in the SCCs that depend on q's SCC. We again can reach the fixpoint for each SCC in bottom-up order. The live variable analysis does not need to be run again. However, the derivation of live region variables at the program points in q is based on its region points-to graph and will need to be done again. The change of the local region liveness in q likely causes changes to the global region liveness of the procedures it calls, i.e., all procedures in the same SCC as q and the ones in the SCCs on which the SCC of q depends. So we also need to recompute the *bornR* and *deadR* sets of them.

We repeat the above process until no more same-edge is removed. At that time, all the remaining same-edges imply that reuses can safely happen. The whole algorithm is specified in Algorithm 12 in which $safe(se)$ checks whether the same-edge se satisfies the safeness conditions or not. $dep(q)$ computes the strongly connected component containing q and also the components that depend on the SCC of q. $invert\_dep(q)$ computes the strongly connected component containing q and also the components on which the SCC of q depends. $local\_region\_liveness(q)$ computes the sets of live region variables before and after program points in q (Section 5.2). $global\_region\_liveness(q)$ applies the analysis in Section 5.3 to q to re-calculate the *bornR*, *deadR*, and *outlivedR* of the procedures that q invokes.

10.5 Transformation

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

The transformations for adding extra arguments for region arguments, for annotating construction unifications with region variables, and for introducing the `create` and `remove` instructions are exactly the same as presented in Chapter 6.

A procedure can safely manipulate the region variables in its *bornR*, *deadR*, or *localR*. Ones in *localR* are local to the procedure. For the non-local ones in *bornR* and *deadR* we are sure that the global context of the procedure does not interfere with them, as the liveness and region aliasing have been taken into account when computing *bornR* and *deadR*. Consider a region variable `R` that the procedure can manipulate. In general a `create(R)` is inserted before the program point where `R` becomes live, i.e., it is not live before but live after the point and the literal at the point does not create it. For removal, there are two cases. If `R` becomes dead because it is live before but not live after the point, we add `remove(R)` after the point if the literal at the point does not remove `R`. If `R` becomes dead since it is live after the point but is not live before the next point in an execution path, we add `remove(R)` before the next point.

For our new analysis we need to introduce the region-annotated assignment.

Algorithm 12 Eliminating bad same-edges

Require: Happen after region liveness analysis: all region points-to graphs and region liveness information are available.
Ensure: All remaining same-edges are good.

```
repeat
    for all procedure q: G_q = (_, _, S_q) do
        for all se = s(k, i, k') ∈ S_q do
            if !safe(se) then
                unify(k, k')
                mark that G_q has changed.
            end if
        end for
    end for

    % Recompute related region points-to graphs and α mappings
    % of the callers of q.
    for all q whose G_q has changed do
        re_interproc_SCCs = dep(q)
        for all SCC ∈ re_interproc_SCCs do
            repeat
                for all q' ∈ SCC do
                    interproc(q')
                end for
            until nothing changed
        end for
    end for

    % Recompute local region liveness in q.
    for all q whose G_q has changed do
        local_region_liveness(q)
    end for

    % Recompute global region liveness of the procedures q calls.
    for all q whose G_q has changed do
        re_global_liveness_SCCs = invert_dep(q)
        for all SCC ∈ re_global_liveness_SCCs do
            repeat
                for all q' ∈ SCC do
                    global_region_liveness(q')
                end for
            until nothing changed
        end for
    end for
until No more same-edge is removed.
```

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

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

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

Figure 10.9: No explicit "reuse" after procedure calls.

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

We can see the effect of same-edges at procedure calls when changing the code of `q` by replacing the assignment at (4) with a recursive call as in Figure 10.9. Nothing other than region arguments is added at (4).

10.6 Discussion

10.6.1 Preliminary Experimental Results

So far we have not implemented region reuse in the Mercury compiler yet. The problem for which region reuse offers a solution is revealed in our study of the memory behaviour of the two benchmarks, **dna** and **life** (Section 9.4). To study the memory behaviour of these programs with region reuse we mimic their memory behaviour as if region reuse had been used by changing their source. Recall that in Section 10.1 we introduced region reuse to avoid copying terms at the moment of an assignment unification. Therefore we modify the benchmarks by adding an explicit copying at the place where our new approach would introduce a region reuse assignment. After that we can use the existing RBMM system, in particular the rbmm3 system in Chapter 9, to execute them. By doing so we can study the effects of region reuse on memory consumption. Note that for the discussion here, we manually introduce copying procedures that make the *desired* copies. The existing region analysis (in rbmm3) takes into account the copying procedure and then can decide whether the original and the copied ones can be kept apart or not. For example, for a list we copy both the list skeleton and the elements, the region analysis decides for the program as a whole whether the old and the new skeletons as well as the old elements and the new elements are in the same regions

or not. In summary, we can check whether region reuse can be done and whether it is beneficial for **life** and **dna** (and any other program) by adding explicitly the copying and by using the existing RBMM system. The profiling information about memory consumption of these programs allows us to measure the effects of region reuse.

For **life**, the region reuse-mimicking program is exactly the same as **rlife**, which we already experimented with in Chapter 9 and studied specifically in Section 9.4. The effect of region reuse is that the maximal number of words used is reduced by 77% to 1856 words.

Interestingly for **dna**, we also can achieve the same effect as in **rdna** with region reuse. We manually replace the assignments that caused the harmful region-sharing in the existing RBMM system with calls to a copying predicate. This region reuse-mimicking program then confirms that in a system with region reuse, all these "harmful" assignments would be replaced by our region-annotated assignments. This would result in a significant reduction of nearly 90% in the size of the largest region.

In terms of runtime performance, we would reasonably expect that region reuse can only lead to speedups in comparison with copying versions. It is because, with region reuse, the memory effect is the same but we normally have a smaller number of regions and virtually no copying overhead. (We would have the overhead of the extra assignments between related (region) variables in the region-annotated assignments but it would be negligible.) So the execution times of **rlife** and **rdna** in Figure 9.7 can be considered as the upper bound for **life** and **dna**, respectively, in the RBMM system with region reuse.

Last but not least, we would not need to manually study the logic of programs to rewrite them with copying to have better memory reuse, at least for the popular class of programs we are studying here . This practice has long been experienced as a nontrivial and tedious task. We experienced it ourselves in this research and the efforts to ease this pain can be found in (Tofte, Birkedal, Elsman, and Hallenberg 2004) (Sections 4 and 5).

10.6.2 Future Work

Implementing the improvement presented in this chapter is part of future work. We believe that the runtime support for regions in Chapters 7 and 8 is able to support region reuse with little change. Other than implementing the regions themselves and the region instructions, the runtime support for regions has two tasks in supporting backtracking. First, it provides protection for regions that are removed during forward execution if they are still needed when backtracking happens, ensuring that these regions will only be reclaimed when they are not used in any future computation. Second, it allows instant reclaiming of *new* regions and *new* allocations into *old* regions at choice points. The essential elements in the support are the time-stamping of the regions when they are created in

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

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

Figure 10.10: A chance for optimization.

order to distinguish old and new regions and the estimation of the sets of regions that are created, removed and allocated into by a goal. The region reuse affects none of these because the time-stamp of a reused region is intact when the reuse happens and because of the fact that the region is reused the created, removed, and allocated region sets of any goals are not changed.

The class of programs that potentially benefit from this novel extension includes state transition programs in which a sequence of states are computed iteratively and typically we are only interested in the last state. In the set of benchmark programs used in this thesis, there are two typical examples of this for which the region reuse improvement should be beneficial, namely **life** and **dna**.

The attentive reader might get worried about procedures such as `qsort` in Figure 10.10. In the approach in (Phan and Janssens 2007) all the variables involved with the accumulating parameter and the result are put in the same region. This is perfect because the accumulator is gradually built up to become the final result. Our new approach seems to be spoiling this. First notice that `A1` and `S2` are in the same region `R6` due to the construction. In the first execution path we have a same-edge at (2) from the region `R2` of `A` to the region `R3` of `S`. In the second execution path we have a same-edge at (5) from `R2` to `R6` and a same-edge at (7) from `R6` to `R3`. All same-edges are "safe". The resulting region-annotated program has a region-annotated assignment at (2). When we mimic the region-annotated assignment by a copying of the list skeleton, we see that region reuse can be done and that the maximal number of words used is the same as with the existing RBMM system, thus optimal. However, this instruction is sheer overhead. It would be removed by the following optimization step. For `qsort` in all execution paths we have a same-edge *path* from the region `R2` of `A` to the region `R3` of `S`. Thus, we could put all the involved variables again in one region. Note that the optimization condition does not hold for e.g., `q` in Figure 10.9. More programs

will also have to be studied in order to know the relevance and the impact of this optimization step.

10.7 Conclusion

In this chapter we have presented an improvement for the region analysis and transformation in Chapters 4, 5, and 6 that results in better memory reuse for a certain class of programs. By making the region points-to analysis path-sensitive (in the sense that it distinguishes different local execution paths in a procedure) we achieve a more precise region model of memory use. This information then is verified against the liveness information to make sure that the resulting transformation is sound. We formulated the local and global liveness safeness conditions for this purpose. In terms of runtime performance we believe that region reuse can only lead to faster code.

Chapter 11

Related Work

It is not our intention to give a detailed overview of the research on RBMM in general. When we started our research on region-based memory management in the context of Mercury around the year 2004, the RBMM research for functional programming was already quite developed. An in-depth review of the RBMM research for functional programming can be found in (Tofte, Birkedal, Elsman, and Hallenberg 2004). In this chapter, we only discuss the publications that are important and related to our work.

Region model. The research on automated region-based memory management for programming languages started with the work of Tofte and Talpin (Tofte and Talpin 1997) for functional programming, in particular for a simplified call-by-value lambda calculus. They divide program terms into regions using a technique similar to unification-based type inference in which the types have been annotated with region variables. The lifetimes of the regions are computed based on the lexical scope of the expressions and the regions themselves are forced to operate in a stack-like manner in the sense that the last region created is the first one destroyed. While lexically-scoped regions and stack discipline seem natural to the evaluation of lambda expressions and simplify the task of deciding region lifetime they, in certain cases, give the regions lifetimes that are longer than needed, limiting the chances of reusing memory and spoil "tail call optimization". There have been several subsequent refinements (Birkedal, Tofte, and Vejlstrup 1996; Aiken, Fähndrich, and Levien 1995) and new developments (Henglein, Makholm, and Niss 2001) in the functional programming paradigm. Birkedal et al. present *Storage Mode Analysis* (Birkedal, Tofte, and Vejlstrup 1996) that mitigates the restrictions by resetting regions. However, programs are often required to be rewritten in an non-intuitive way so that region resetting is possible. In (Aiken, Fähndrich, and Levien 1995), the authors, with the observation that forcing the stack-like relation on the lifetimes of regions is too strict, have decoupled region creation and removal

so that regions can have arbitrarily overlapping lifetimes. More prominently in this decoupling direction, an imperative sublanguage on regions is proposed in (Henglein, Makholm, and Niss 2001). In this imperative model, regions are allowed not only to have arbitrary lifetimes but also to change their bindings. A region also contains a reference counter that can give this system more flexibility in controlling the region's lifetime. The most complete functional programming system with RBMM is the MLKit (Tofte, Birkedal, Elsman, Hallenberg, Olesen, and Sestoft 2006) in which storage is managed solely by RBMM and whose performance can compete with state-of-the-art SML compilers.

Our static region analysis and transformation for the logic programming language Mercury are inspired by the work in (Cherem and Rugina 2004), which abandoned the lexical scope and stack discipline of regions, allowing the regions to have shorter, arbitrarily-overlapped lifetimes. The analyses in (Cherem and Rugina 2004) take into account the data flow in a Java program in order to determine the regions and their lifetimes. Therefore the analyses had to be redefined for Mercury to deal with unification and a control flow that are fundamentally different from object manipulation and the control flow in Java. We have also exploited the type declarations in Mercury to achieve a finite representation of the storage of (recursive) structures in terms of regions using type-based region graphs and region points-to graphs (Chapter 3). Using some kind of graphs to model storage is not new in the research about structures of the heap (Chase, Wegman, and Zadeck 1990; Steensgaard 1996). Our graphs share many features with annotated types in which each type constructor is annotated with a location or a region (e.g., in (Baker 1990; Tofte and Talpin 1997)). In particular Baker in (Baker 1990) pointed out that such annotated types can give information about sharing, very similar to the concept of *region-sharing* in Chapter 3.

The liveness information derived in Chapter 5 allows interprocedural creation of regions which was not handled in (Cherem and Rugina 2004). This can give finer lifetimes to regions, which can result in better memory reuse in certain situations. For example, for a region like `R1` in `p` in Figure 6.3, the system in (Cherem and Rugina 2004) will force `R1` to be live throughout `p`. If we had replaced the literal at (4) with a recursive call to `p` (such as `p(A - 1, B)`) all the temporary memory allocated at (1) would have built up in `R1`.

RBMM in logic programming. The first application of RBMM to logic programming was reported by Makholm in (Makholm 2000b) and (Makholm 2000a) for Prolog. He realized the idea that backtracking can be handled completely by runtime support, which can keep the region inference simple. However, the Prolog system (mostly the intermediate code) used in this work is not standard. This shortcoming was fixed in (Makholm and Sagonas 2002) where Makholm and Sagonas presented extensions to the WAM to enable region-based memory management. The main differences between their work and ours are that Mercury supports if-then-elses with conditions that can succeed more than once, and the

Mercury implementation generates specialized code for many situations that Prolog handles with a more general mechanism (e.g., Mercury has separate implementations for nondet and semidet disjunctions). This also means that we make use of declarations of determinism in Mercury. The first difference required new algorithms, while the second posed an engineering challenge in keeping overheads down, since any given overhead would hurt Mercury more than Prolog due to Mercury's higher speed.

Region Reuse. *Storage Mode Analysis* presented in (Birkedal, Tofte, and Vejlstrup 1996) targets the same class of programs as our region reuse optimization. Being an extra phase after the region inference in (Tofte and Talpin 1997) that puts all such states into the same region, Storage Mode Analysis then aims to *reset* the region before each iteration if it is safe to do so. The decision is also based on liveness information. However for Game of Life, this analysis requires manually rewriting the program with a copying function so that the resetting is possible (Tofte, Birkedal, Elsman, Hallenberg, Olesen, and Sestoft 2006; Henglein, Makholm, and Niss 2001). In (Henglein, Makholm, and Niss 2001) the authors develop an expressive region type system that can accept several region-annotated versions for a program. Their region inference based on that type system also can produce an annotated version with the same region behaviour for Game of Life as our proposal with region reuse, without requiring rewriting. One interesting open problem with their region type system is to have a strong region inference that, among a number of accepted annotated programs, can choose to generate an optimal one. We start from an analysis algorithm that performs well generally and extend it to obtain better results in a popular pattern of code, which is well known to be difficult for RBMM. By mimicking region reuse using copying procedures, we showed that it can effectively reduce memory consumption. In (Makholm and Sagonas 2002) the authors use an adapted version of the region inference system in (Henglein, Makholm, and Niss 2001) for SML to Prolog in a WAM-based system. However, for the benchmark similar to our **dna** a region-friendly version is still needed to reduce memory consumption.

Chapter 12

Conclusion and Future Work

12.1 Conclusion

We have made region-based memory management available as an alternative storage management technique in Mercury, which is a strongly typed and moded logic programming language.

We have developed a region analysis and a program transformation, which are based on a region model in terms of region points-to graphs. From a typical Mercury program our system automatically produces a region-annotated program that is augmented with region constructs for memory management. We have proven that our analysis and transformation algorithms are always terminating and that memory accesses in the annotated programs are safe in the absence of backtracking.

We implemented the runtime support for region-based memory management in the Mercury compiler (MMC). Other than providing the implementation of regions and region operations the runtime support also guarantees the safeness of memory accesses when backtracking happens by protecting resurrected regions, and only reclaims them when they are both forward and backward dead. Moreover, we can support instant reclaiming in our region system, which in some programs can help reusing memory effectively. The main challenge for the runtime support is to support backtracking correctly without incurring significant overhead in deterministic programs. Our experiments show that using RBMM instead of the Boehm collector yields remarkable speedups, from near 10% to more than 65%, for two third of the benchmark programs. More notably, several of those with large speedups are actually difficult cases for RBMM. This indicates that the runtime support we provided for backtracking incurs very modest overhead in general, contributing to the overall better performance. The memory use results of the benchmarks are also positive, in some programs we obtain optimal memory consumption. Over all

the benchmarks the average memory saving is about 73%, while if excluding the region-friendly programs, it is above 65%. We also need to take into account the fact that we have not yet included many optimizations that have been studied for RBMM, such as stack allocation of regions (Birkedal, Tofte, and Vejlstrup 1996; Cherem and Rugina 2004), merging regions that are removed at the same points, i.e., having the same lifetimes (Makholm 2000a).

Finally, we provided an in-depth study of the relation between sharing and memory reuse in region-based systems (Section 9.4). The study reveals two sources of imprecision from which region-based systems usually suffer. They are the handling of some kind of assignments (even though in declarative languages we do not explicitly have assignments) and of recursive data. We propose an automated solution for the first problem in (Phan and Janssens 2009). The second problem with recursive types is well-known in the research about type systems and inference and heap structures where we have to accept the loss of precision to have a finite model. To improve memory reuse in this case we can combine RBMM with other techniques, such as runtime or compile-time garbage collection. The copying approach as demonstrated in Chapters 9 and 10 can be viewed as the mimicking of runtime copying garbage collection. Combining RBMM with copying garbage collection has been realized in the MLKit (Hallenberg, Elsman, and Tofte 2002).

12.2 Future Work

12.2.1 Support the Whole Mercury Language

The main limitation of our work is that currently, the program analysis underlying our system supports only a subset of Mercury and does not support higher-order code, foreign language code and multi-module programs. These extensions to the analysis should not require any changes to the runtime system we have presented in this thesis. Being able to work with a larger subset of Mercury will allow us to evaluate RBMM in bigger and more realistic programs.

Multi-Module Programs

A module system is a vital feature of a programming language as it facilitates the development of large software applications. It offers several benefits, such as team work where programmers can work independently on their modules, software reuse by the use of library modules, less effort to maintain source code, just to name a few.

A standard Mercury program uses modules and also some library modules from a rich set provided by the language implementation. Therefore to have practical use for Mercury, analysis and optimization techniques must work with modules.

The region analysis and transformation as we have described in this thesis are

designed for *monolithic* Mercury programs, i.e., those written in just one module in one file. In this section, we look at the possibilities of adapting them to support *modular* programs.

There has been research on modular program analysis and optimization (Codish, Debray, and Giacobazzi 1993; Vanhoof and Bruynooghe 2000; Puebla, Correas, Hermenegildo, Bueno, de la Banda, Marriott, and Stuckey 2004). The considerations for design and implementation such modular techniques were described in (Puebla and Hermenegildo 2000) in which the authors discussed different scenarios for modular analysis and optimization when programs are decomposed into modules. A generic modular analysis framework has been proposed in (Puebla, Correas, Hermenegildo, Bueno, de la Banda, Marriott, and Stuckey 2004) for abstract interpretation. The framework provides the machinery for context-sensitive program analysis. We think that it can also be used for our region analyses in this thesis. However, here we will not try to adapt our analyses following this framework. Instead we would like to look at the characteristics of the specific analyses in our RBMM system to study what needs to be done to make them modular.

We consider the scenario where our region analysis and transformation can access the source code of all the modules in a program. We expect that one module is analyzed at a time and it can also be analyzed more than once if needed. We will discuss the handling of library modules later.

Region points-to analysis. We will sketch how we can adapt the region points-to analysis to work with modules by exploiting its characteristics.

The first observation is that the intraprocedural region points-to analysis can be applied to a procedure independently because it only analyzes (specialized) unifications. Therefore we can do intraprocedural analysis for all the modules.

The second characteristic of the region points-to analysis is that the analysis information in the interprocedural analysis only propagates from a callee to its caller. If there is no mutually recursive calls among modules (i.e., if no circular dependencies among modules exist), we can do the interprocedural analysis bottom-up in the dependency graph of the modules. Each leaf module, namely one does not invoke procedures of other modules, can be analyzed in the way we analyze the whole program in this thesis. That is, we divide the call dependency graph of the procedures in the leaf module into strongly connected components then analyze each component until a fixpoint is reached. We have already shown that such analysis is terminating.

Then when a non-leaf module (i.e., a module that depends on other modules) is analyzed the analysis information of all the procedures it calls, which do not belong to the module being analyzed, is already available for use. In this situation, each module needs to be analyzed just once and in the bottom-up order in the dependency graph of modules.

If we admit circular dependencies, i.e., there are mutually recursive calls among modules, then we may want to decompose the dependency graph of the *modules*

into strongly connected components. For each component of modules, we need to reach a fixpoint for all the modules. If we assume to have a call dependency graph of all the procedures in these modules then the interprocedural analysis can again be applied to each strongly connected component of this graph.

For library modules (or predefined modules), we can see that procedures in library modules will normally be invoked in programs that use the modules. In other words, they are callees and therefore, the information in their region points-to graphs will only be propagated to programs that use them. Their region points-to graphs will certainly not be invalidated due to the region analysis of the programs. Hence, we can do region points-to analysis for all the library modules in advance. The resulting region points-to graphs can be saved and used later on when analyzing the programs that invoke library procedures. This will require saving and re-reading analysis information (in this case region points-to graphs) somehow but should be feasible to do.

Region liveness information. After having the region points-to graph of a procedure, the local region liveness information of the procedure (i.e., region variables that are live before and after program points) depends only on its source code. So we can compute this information for each procedure separately. This local region liveness analysis has been described in Section 5.2 and can be run for each procedure in each module without changes.

The global region liveness information of a procedure, the liveness of its region arguments, depends on its calling contexts. With the assumption that we have the source code of all modules, we can analyze all the call sites of the procedure. Hence, it should be straightforward to adapt this to work with modules.

Handling global region liveness for procedures in library modules poses a bigger problem because we normally cannot know in advance the contexts in which they will be called. One simple and naive way to handle this is to compute and store the global region liveness for the most pessimistic and the most optimistic cases. The most pessimistic case means that all region arguments of a procedure are considered outliving any calls to it. That means output region arguments are created somewhere before the call and input ones are removed somewhere after the call. This is always safe but at the same time can lead to inefficient memory reuse because regions may be given longer lifetimes than they should be. The most optimistic case means that all region variables in the initial *deadR* set are *assumed* becoming dead, all in the initial *bornR* set becoming live in the call, and there is no alias among them. This will lead to the generation of two specialized versions for a library procedure. The most pessimistic one is always safe to use while the most optimistic one can only be used if the assumptions are satisfied by a calling context.

We conjecture that only experiments can give us more insights on whether the above naive approach is good enough. A better but more complicated way would be to figure out which specialized versions, other than the most pessimistic one, are

worth being generated. There are several options to do this. Either we allow the library module developers to specify them or we can use *deep* profiling information (i.e., at the level of calls) on the usage of a library procedure for guidance.

Other features

Two other popular features of Mercury are higher-order programming and the interface to foreign code.

A higher-order procedure in Mercury is represented by a higher-order term that is a pointer to a memory block whose first two fields are used to store the number of arguments and the address of the involved procedure. The other fields in the block are for the arguments themselves. So it is possible to treat a higher-order term like a normal data structure. That means we use one region to store the memory block, the number of arguments and the address of procedure are stored right in this region, i.e., do not require regions. Each argument then may be stored in its own region depending on its type. The region points-to graph of the procedure in the higher-order term can also be produced in the normal way.

It seems that the liveness analysis of variables having higher-order types and their region variables can be the same as that for the program variables we have been dealing with. At some point the higher-order term will be called, which basically means that its procedure is called. These calling contexts will influence what the procedure can do to its nonlocal region variables, i.e., whether any nonlocal region variables can become live or become dead inside the procedure or not. However, in the worst case we just end up at the most pessimistic but always safe situation where the procedure does not create or remove any nonlocal region variables, leaving this task to the calling contexts.

Interface to foreign code refers to procedures in Mercury that are written in a different programming language, such as C. Several library procedures have been implemented in this way. We probably can just ignore these procedures and let the foreign code manage its memory. For a Mercury variable that refers to the memory that is allocated by the foreign code we still assign a region for it, but the allocation by the foreign code actually will not happen in the region. This region can still safely be removed as decided by our region analysis when it is no longer referred to.

12.2.2 Combining with Compile-Time Garbage Collection

In Section 9.4, we pointed out that RBMM does not reuse memory efficiently in programs where recursive structures are updated, such as in **isort** and **bsolver**. In such situations the integration of RBMM with compile-time garbage collection (CTGC), which provides in-place updating, can be useful. In this formulation, the dead memory cells in the long-lived region can be reused for the new allocations into it.

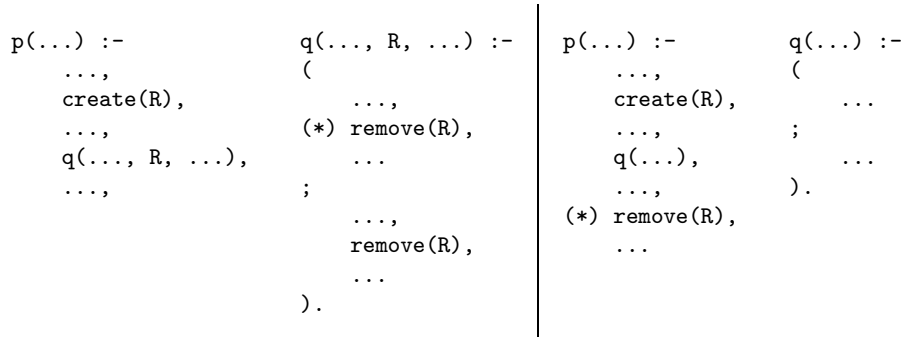
For Mercury, CTGC has been reported in (Mazur, Janssens, and Bruynooghe 2000; Mazur, Ross, Janssens, and Bruynooghe 2001; Mazur 2004). So the challenge would be to combine in a correct way the two approaches that differ at the level of granularity. RBMM works at the level of regions, i.e., all memory allocated at an allocation site normally goes into one region, while CTGC tracks every single allocation. Another major difference of the two approaches is that the CTGC system in (Mazur 2004) takes into account backward execution when computing liveness information, whereas our region liveness analysis (Chapter 5) only deals with forward execution.

There are also situations in which RBMM can improve the performance of CTGC. In CTGC, when there is a dead cell in a procedure but locally no new memory needs to be allocated or the size of the dead cell is smaller than that of the new cell that is going to be allocated (e.g., the motivational example in Section 1.2), the dead cell is not reused and therefore it is potential a memory leak. In (Mazur 2004) the author uses a global cache to keep track of such dead-but-not-locally-reused cells for future allocations. It was reported in (Mazur 2004) that using the global cache can reduce memory footprint but damages runtime performance at the same time. The incurred overhead can be the maintenance cost of the cache itself. The cost of each new allocation is also increased for having to search for a matched cell in the cache. The operation of the cache may also interfere with the runtime garbage collection. Combining with RBMM we can hope that such cells are allocated into separate regions that can be released in time.

We suggest a possible direction for combining the reuse analyses in CTGC in (Mazur 2004) and the region analyses in this thesis. The ultimate decision resulting from the analyses in (Mazur 2004) is that, locally inside a procedure, the memory cell referred to by a procedure variable is reused for another procedure variable. If CTGC decides so, in our region points-to analysis we can put these two variables into the same region. In the `split` procedure in the *quicksort* example in Figure 2.2, CTGC may decide that the memory cell referred to by `L` is dead after the deconstruction at (4) and will be reused for either `L1` or `L2`. Influenced by this decision from CTGC, in our region points-to analysis we then can say that these three variables should be in the same region. This results in in-place updating inside a long live region, which should give us equally good memory reuse and probably better runtime performance because there is less overhead of handling regions.

12.2.3 Backward Region Liveness Analysis

In this thesis we chose to support backtracking by runtime support. This decision was based on several factors. First, we have evidence that it can be done (Makholm 2000a; Makholm and Sagonas 2002). Second, from the experience in (Mazur 2004) we know that taking into account backward execution can make program analysis



(a) Case 1. (b) Case 2.

Figure 12.1: Backward liveness of region variables.

much more complicated. Finally, we would certainly need more insights on the interaction between regions and backtracking at runtime before trying to handle it at compile-time.

So one direction for future work would be to consider backward liveness in the region liveness analysis. Consider two procedures `p` and `q` as in Figure 12.1 in which `q` contains an explicit disjunction with two disjuncts. Assume that the region `R` becomes live in `p`.

In the first case assume also that `R` is forward dead in `q`. When we only take into account forward liveness, as in this thesis, a `remove(R)` instruction is added to each disjunct. The one at (*) is executed but does not do anything because the region is disj-protected. If we also consider backward liveness, we can see that at (*) `R` is still backward live so we should not add the `remove` instruction at (*) anymore. This optimization will certainly result in speedup. We think that this optimization can be either integrated directly into the analysis and transformation algorithms or implemented as a "post-processing" step on the region-annotated programs.

In the second case assume that `R` is live through `q` or does not relate to `q` at all. There is a point, e.g., (*), where `R` is forward dead. However, we will not be able to find out a point where `R` is both forward and backward dead. So in this case the runtime support is still needed to reclaim `R`.

This shows that such an approach that also considers backward liveness of region variables can reduce the runtime overhead of supporting backtracking but the runtime support as we provided in this thesis may still be necessary. It would be interesting to compare the effectiveness and complexity of this approach to protecting backward live regions in this thesis.

12.2.4 Further Optimizations

Region reuse. In Chapter 10 we propose a region reuse improvement, which is very interesting, but has not been implemented yet. Its implementation should be included in order to study whether runtime performance can also get improved when memory use is more efficient or not.

Region merging. We often claim that we would like to put terms having the same lifetimes into the same regions so that they can be reclaimed all at once when their lives end. So when our transformation decides for example that two regions are removed at the same point, i.e., their lives end at the same point, we should "merge" them. Doing so will reduce the number of regions that a program has to handle. In our experiments we have evidence that handling too many regions can damage runtime performance therefore merging regions that are dead at the same time seems to be a good idea. This practice has been mentioned in (Makholm 2000a) but the author did not report any related experimental result in the work.

Stack allocations. It has been reported that allocating finite regions, i.e., ones whose size are known at compile-time, on the stack can lead to speedup (Birkedal, Tofte, and Vejlstrup 1996). In our region analysis, for a procedure we already detect local regions whose scopes are the same as that of the procedure. We can also try to determine their sizes and then allocate those with finite sizes on the stack. In (Birkedal, Tofte, and Vejlstrup 1996) the finite regions are ones that are allocated into exactly one time. Basically they are regions that contain just one memory cell. The regions containing higher-order terms, for example, should often be finite because they typically contain only one closure. There are cases where this is not true. For examples, programs may build lists of closures, but we conjecture that such situations are not very popular.

Appendix A

Using the region-based systems

This appendix describes the installation and usage of the region-based systems. The systems are available as part of the *Release of the Day* of the Mercury compiler, which can be downloaded at this address:
`http://www.cs.mu.oz.au/research/mercury/download/rotd.html`.

Installation

During the description we assume that the compiler will be installed on a machine running Linux with a compatible GNU C compiler `gcc` and GNU `make` as instructed on the above site. In this thesis we used `gcc` version 3.3.5 and `make` version 3.81.

1. Download and decompress the compiler package into a folder.

   You should get used to the installation process of the Mercury compiler, which is explained in the `INSTALL_CVS` file that comes inside the package.

2. Create a file named Mmake.params, add this text `GRADE=asm_fast.gc.rbmm` and save it. If the file exists, just edit it and make sure the string is there. This is to ensure that the Mercury compiler that we are compiling can produce the region-annotated programs.

3. At the command prompt, execute the following command:

   ```
   ./configure --prefix=path --enable-libgrades=asm_fast.gc.rbmm,
   asm_fast.gc.rbmmd,asm_fast.gc.rbmmp,asm_fast.gc.rbmmdp
   ```

in which `path` is the path to the folder where the compiler will be installed.

4. Follow the next two steps after `./configure` as written in the `INSTALL_CVS` file:

   ```
   mmake depend
   mmake
   ```

   After this step the Mercury compiler has been produced and can be used. You can check this by looking for the executable `mercury_compile` in the subfolder `compiler`.

   Now if we want to bootcheck that compiler we can follow the last two steps. Otherwise we can just execute:

   ```
   make install
   ```

After this the Mercury compiler and libraries are installed in the folder specified in `path`.

Usage

After making sure that your system knows about the installed compiler (normally by exporting *path* to the `PATH` environment variable), we can use the compiler as follows, e.g., for compiling the program **nrev** in the file `nrev.m`.

1. To produce the executable `nrev`, which uses region-based memory management:

   ```
   mmc --grade=asm_fast.gc.rbmm --region-analysis nrev.m
   ```

2. To also write out the intermediate program with region annotations (i.e., one obtained after the region analysis and transformation) in the file whose name contains 240.

   ```
   mmc --grade=asm_fast.gc.rbmm --region-analysis --dump-hlds=240 nrev.m
   ```

3. To get profiling information, like in Table 9.4, add the call to `print_rbmm_profiling_info(!IO)` to the `main` procedure, then:

   ```
   mmc --grade=asm_fast.gc.rbmmp --region-analysis nrev.m
   ```

   This will produce the executable nrev that can report on profiling information about regions after it finishes.

4. To print out debugging information while running:

   ```
   mmc --grade=asm_fast.gc.rbmmd --region-analysis nrev.m
   ```

   The grade `asm_fast.gc.rbmmdp` is the combination of both debugging and profiling purposes.

5. By defaults the runtime system in the region-based system is function calls. To use the macros system we need to provide one extra flag, `-DMR_RBMM_USE_MACROS`, such as:

   ```
   mmc --grade=asm_fast.gc.rbmm --region-analysis --cflags -DMR_R
   BMM_USE_MACROS nrev.m
   ```

Bibliography

Aiken, A., Fähndrich, M., and Levien, R. 1995. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*. ACM Press, 174–185.

Aït-Kaci, H. 1999. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press.

Aspinall, D., Hofmann, M., and Konečný, M. 2008. A type system with usage aspects. *The Journal of Functional Programming 18(2)*, 141–178.

Baker, H. G. 1990. Unify and conquer. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*. ACM, New York, NY, USA, 218–226.

Birkedal, L., Tofte, M., and Vejlstrup, M. 1996. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, 171–183.

Boehm, H. and Weiser, M. 1988. Garbage collection in an uncooperative environment. *Software – Practice and Experience 18*, 807–820.

Chase, D. R., Wegman, M., and Zadeck, F. K. 1990. Analysis of pointers and structures. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. ACM, New York, NY, USA, 296–310.

Cherem, S. and Rugina, R. 2004. Region analysis and transformation for Java programs. In *Proceedings of the 4th International Symposium on Memory Management*. ACM Press., 85–96.

Cherem, S. and Rugina, R. 2006. Compile-time deallocation of individual objects. In *ISMM '06: Proceedings of the 5th international symposium on Memory management*. ACM, New York, NY, USA, 138–149.

Chin, W., Craciun, F., Qin, S., and Rinard, M. 2004. Region inference for an object-oriented language. In *PLDI '04: Proceedings of the ACM SIGPLAN*

2004 conference on Programming language design and implementation. ACM, New York, NY, USA, 243–254.

Codish, M., Debray, S., and Giacobazzi, R. 1993. Compositional analysis of modular logic programs. In *In Proc. Twentieth Annual ACM Symp. on Principles of Programming Languages*. ACM Press, 451–464.

Gay, D. and Aiken, A. 1998. Memory management with explicit regions. In *SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press., 313–323.

Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., and Cheney, J. 2002. Region-based memory management in Cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.* ACM Press., 282–293.

Guyer, S. Z., McKinley, K. S., and Frampton, D. 2006. Free-me: a static analysis for automatic individual object reclamation. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. ACM, New York, NY, USA, 364–375.

Hallenberg, N., Elsman, M., and Tofte, M. 2002. Combining region inference and garbage collection. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. ACM, New York, NY, USA, 141–152.

Henglein, F., Makholm, H., and Niss, H. 2001. A direct approach to control-flow sensitive region-based memory management. In *Principles and Practice of Declarative Programming*. ACM Press., 175–186.

Hudak, P. and Bloss, A. 1985. The aggregate update problem in functional programming systems. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, New York, NY, USA, 300–314.

Jones, S. B. and Le Métayer, D. 1989. Compile-time garbage collection by sharing analysis. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*. ACM, New York, NY, USA, 54–74.

Makholm, H. 2000a. A region-based memory manager for Prolog. In *Proceedings of the 2nd International Symposium on Memory Management*. ACM Press., 25–34.

Makholm, H. 2000b. Region-based memory management in Prolog. M.S. thesis, University of Copenhagen.

Makholm, H. and Sagonas, K. 2002. On enabling the WAM with region support. In *Proceedings of the 18th International Conference on Logic Programming*. Springer Verlag.

Mazur, N. 2004. Compile-time garbage collection for the declarative language Mercury. Ph.D. thesis, Department of Computer Science, Katholieke Universiteit Leuven.

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

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

McCarthy, J. 1960. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM 3,* 4, 184–195.

Mercury. 2009. *The Mercury programming language reference*. http://www.cs.mu.oz.au/mercury/information/doc-latest/mercury_ref.

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

Phan, Q. and Janssens, G. 2007. Static region analysis for Mercury. In *Proceedings of the 23rd International Conference on Logic Programming*. Springer, 317–332.

Phan, Q. and Janssens, G. 2009. Path-sensitive region analysis for Mercury programs. In *Proceedings of the 11th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*. ACM Press.

Phan, Q., Somogyi, Z., and Janssens, G. 2008. Runtime support for region-based memory management in Mercury. In *Proceedings of the 7th International Symposium on Memory Management*. ACM Press, 61–70.

Puebla, G., Correas, J., Hermenegildo, M. V., Bueno, F., de la Banda, M. G., Marriott, K., and Stuckey, P. J. 2004. A generic framework for context-sensitive analysis of modular programs. In *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development, number 3049 in LNCS*. Springer-Verlag, 234–261.

Puebla, G. and Hermenegildo, M. 2000. Some issues in analysis and specialization of modular ciao-prolog programs. In *In Special Issue on Optimization and Implementation of Declarative Programming Languages, volume 30 of Electronic Notes in Theoretical Computer Science. Elsevier - North*.

Sastry, A. V. S., Clinger, W., and Ariola, Z. 1993. Order-of-evaluation analysis for destructive updates in strict functional languages with flat aggregates. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*. ACM, New York, NY, USA, 266–275.

Somogyi, Z., Henderson, F., and Conway, T. 1996. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming 29(1-3)*, 17–64.

Steensgaard, B. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, 32–41.

Tofte, M., Birkedal, L., Elsman, M., and Hallenberg, N. 2004. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation 17*, 245–265.

Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N., Olesen, T., and Sestoft, P. 2006. *Programming with Regions in the MLKit*. http://www.it.edu/research/mlkit/dist/mlkit-4.3.0.pdf.

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

Vanhoof, W. and Bruynooghe, M. 2000. Towards modular binding-time analysis for first-order Mercury. In *In Special Issue on Optimization and Implementation of Declarative Programming Languages, volume 30 of Electronic Notes in Theoretical Computer Science. Elsevier - North*.

Wilson, P. R. 1995. Uniprocessor garbage collection techniques.

Wilson, P. R., Johnstone, M. S., Neely, M., and Boles, D. 1995. Dynamic storage allocation: A survey and critical review. In *IWMM '95: Proceedings of the International Workshop on Memory Management*. Springer-Verlag, London, UK, 1–116.

Index

List of Publications

Contributions at international conferences

Quan Phan and Gerda Janssens. Path-sensitive region analysis for Mercury. *The International Conference on Principles and Practice of Declarative Programming (PPDP 2009)*, Coimbra, Portugal, 1–9.

Quan Phan, Zoltan Somogyi and Gerda Janssens. Runtime support for region-based memory management in Mercury. *The International Symposium on Memory Management (ISMM 2008)*, Tucson, AZ, USA, 61–70.

Quan Phan and Gerda Janssens. Static Region Analysis for Mercury. *The 23rd International Conference on Logic Programming (ICLP 2007)*, Porto, Portugal, 317–332.

Quan Phan. Static Memory Management for Logic Programming Languages. *The 22nd International Conference on Logic Programming, Doctoral Consortium (ICLP 2006)*, Seattle, WA, USA, 465–466.

Quan Phan and Gerda Janssens. Towards Region-Based Memory Management for Mercury Programs (Abstract). *The 22nd International Conference on Logic Programming (ICLP 2006)*, Seattle, WA, USA, 433–435.

Contributions at international workshops

Quan Phan and Gerda Janssens. More Precise Region-Based Memory Management for Mercury Programs. *The 9th International Colloquium on Implementation of COnstraint Logic Programming Systems (CICLOPS 2009)*, Pasadena, CA, USA, 1–15.

Quan Phan and Gerda Janssens. Towards Region-Based Memory Management for Mercury Programs. *The 6th International Colloquium on Implementation of COnstraint Logic Programming Systems (CICLOPS 2006)*, Seattle, WA, USA, 1–15.

Technical reports **(K.U.Leuven, Dept. Computer Science)**

Quan Phan and Gerda Janssens. Region-based memory management for Mercury programs. Part 1: Static analysis and transformation. *Report CW540*, April 2009, 1–32.

Quan Phan and Gerda Janssens. A proposal for runtime region support for Mercury programs. *Report CW482*, March 2007, 1–16.

Quan Phan and Gerda Janssens. A proposal for region-based memory management for deterministic Mercury programs. *Report CW424*, September 2005, 1–22.

Biography

Quan Phan was born on the 5th of June, 1977 in Hanoi, Vietnam. He got the degree of Engineering in Information Technology at Hanoi University of Technologies, Vietnam in the year 2000. He then moved to the south of Vietnam, during the next three years working as a software developer in Saigon.

In 2003, he got married and then went to Katholieke Universiteit Leuven (KUL), Belgium for the program Master in Artificial Intelligence and graduated cumme laude in June 2004. In October 2004, he joined the research group "Declarative Languages and Artificial Intelligence" (DTAI) at the Department of Computer Science in the same university, pursuing a PhD degree under the supervision of Professor Gerda Janssens. He was supported financially by various research projects of DTAI.

From May to July 2007, he visited the Mercury research and development team at the University of Melbourne, Australia, under the guidance of Professor Zoltan Somogyi.