Memory Management with Explicit Regions

by

David Edward Gay

Engineering Diploma (Ecole Polytechnique Fédérale de Lausanne, Switzerland) 1992
M.S. (University of California, Berkeley) 1997

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Alex Aiken, Chair
Professor Susan L. Graham
Professor Gregory L. Fenves

Fall 2001

The dissertation of David Edward Gay is approved:

Chair Date

Date

Date

University of California at Berkeley

Fall 2001

Memory Management with Explicit Regions

Copyright 2001
by
David Edward Gay

Abstract

Memory Management with Explicit Regions

by

David Edward Gay
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Alex Aiken, Chair

Region-based memory management systems structure memory by grouping objects in regions under program control. Memory is reclaimed by deleting regions, freeing all objects stored therein. Our compiler for C with regions, RC, prevents unsafe region deletions by keeping a count of references to each region. RC's regions have advantages over explicit allocation and deallocation (safety) and traditional garbage collection (better control over memory), and its performance is competitive with both—from 6% slower to 55% faster on a collection of realistic benchmarks. Experience with these benchmarks suggests that modifying many existing programs to use regions is not difficult.

An important innovation in RC is the use of type annotations that make the structure of a program's regions more explicit. These annotations also help reduce the overhead of reference counting from a maximum of 25% to a maximum of 12.6% on our benchmarks. We generalise these annotations in a region type system whose main novelty is the use of existentially quantified abstract regions to represent pointers to objects whose region is partially or totally unknown.

A distribution of RC is available at `http://www.cs.berkeley.edu/~dgay/rc`.

Professor Alex Aiken
Dissertation Committee Chair

Contents

List of Figures **iv**

List of Tables **vi**

1 Introduction **1**
1.1 Contributions 2
1.1.1 Language Design 2
1.1.2 Types 3
1.1.3 Implementation Techniques 4
1.1.4 Detailed Performance Study 4
1.2 Comparison to Other Memory-Management Styles 5
1.2.1 Explicit Deallocation 5
1.2.2 Tracing Garbage Collection 6
1.2.3 Reference-counted Garbage Collection 8
1.3 Dissertation Outline 8

2 Related Work **10**
2.1 Regions 10
2.1.1 Static Safety 10
2.1.2 Dynamic Safety 12
2.1.3 No Safety 14
2.2 Other Styles of Memory-Management 15
2.3 Safe C Dialects 15
2.3.1 Language Changes 16
2.3.2 Conservative Garbage Collection 16
2.3.3 Instrumentation 17
2.3.4 Interpretation 17

3 Language Design **19**
3.1 C@ 19
3.2 RC 22
3.2.1 C@ Lessons 23
3.2.2 Assumptions 25
3.2.3 Region Library 25

3.2.4 Type Qualifiers . . . 30
3.2.5 Restrictions . . . 32
3.2.6 Linking to Existing Object Code . . . 36
3.2.7 Fallback to C . . . 36
3.2.8 Debugging Support . . . 37
3.2.9 Examples . . . 37
3.3 Titanium . . . 41
3.3.1 Parallelism in Titanium . . . 42
3.3.2 Shared and Private Regions . . . 43
3.3.3 Region-Based Allocation in Titanium . . . 44
3.3.4 Titanium Example . . . 45
3.4 RC Extensions . . . 47
3.4.1 Alternative semantics for `deleteregion` . . . 47
3.4.2 Further Type Annotations in RC . . . 48
3.4.3 Expressing Locality . . . 50

4 Implementation Techniques **52**
4.1 Compiling to C . . . 52
4.2 Region Library . . . 53
4.2.1 Regions . . . 53
4.2.2 Allocators . . . 55
4.2.3 Page Allocator . . . 56
4.2.4 Page Map . . . 58
4.3 Reference Counting . . . 59
4.3.1 Deleting Regions . . . 61
4.3.2 Reference Counting for Local Variables . . . 61
4.3.3 Local Variable Reference Counts in C@ . . . 62
4.3.4 Local Variable Reference Counts in RC . . . 64
4.3.5 Alternate Reference Counting Implementations . . . 67
4.3.6 Variations on Standard Reference Counting . . . 68
4.3.7 Reference Counts Between Pairs of Regions . . . 69
4.4 Parallelism . . . 72
4.4.1 Parallel Region Implementation . . . 73
4.4.2 Creating and Deleting Regions . . . 74
4.5 Real-Time Regions . . . 76

5 rlang **78**
5.1 rlang Types . . . 78
5.2 Region Type Checking in rlang . . . 80
5.3 Semantics . . . 84
5.4 Soundness . . . 88
5.5 Soundness Proof . . . 90
5.6 Translating RC to rlang . . . 99
5.7 Alternate Translations of RC to rlang . . . 101
5.7.1 Extensions to RC . . . 102

5.7.2 Runtime Checks . . . 102
5.7.3 Annotation Inference . . . 103

6 Results **105**
6.1 Benchmarks . . . 105
6.2 Region Structure . . . 106
6.3 Changes to Benchmarks . . . 109
6.4 Allocators and Test Environment . . . 112
6.5 Benchmark Behaviour . . . 112
6.6 Memory Usage . . . 115
6.7 Performance . . . 117
6.7.1 Performance vs Other Allocation Techniques . . . 117
6.7.2 Performance of Alternative Reference-counting Implementations . . . 119
6.7.3 Reference-counting Overhead . . . 120
6.8 Qualifiers . . . 121
6.9 Local Variables . . . 126
6.10 Other Overheads . . . 128
6.11 Atomic Swaps . . . 128
6.12 Summary . . . 129

7 Conclusion **131**
7.1 Strengths and Weaknesses of Regions . . . 132
7.2 Extensions . . . 133

Bibliography **135**

A Standard C Library Compatibility **142**

List of Figures

1.1 An example of region-based allocation. 2

3.1 C@ Region API 20
3.2 List copy using regions in C@. 21
3.3 An example of region-based allocation. 25
3.4 Region API 26
3.5 An example of region-based allocation in Titanium. 42
3.6 A larger Titanium example 46

4.1 Region structure 54
4.2 Allocator structure 55
4.3 Block header structure 57
4.4 Reference counting and annotation checking 60
4.5 Region scan during `deleteregion(r)`. 62
4.6 Reference counting including same-region references. 68
4.7 Reference counting excluding parent pointers 69
4.8 Reference counting for reference counts between pairs of regions 71
4.9 Reference counting in a parallel language 74

5.1 Region type language 79
5.2 *rlang*, a simple imperative language with regions 80
5.3 Region Type Checking 81
5.4 Semantic reduction rules 85
5.5 Semantic assignment rules 86

6.1 Objects allocated, distributed by object size 113
6.2 Bytes allocated, distributed by object size 113
6.3 Maximum Memory Usage (in kB) and Overheads 116
6.4 Execution time 117
6.5 Execution time, showing time spent in memory management 118
6.6 L2 cache misses 118
6.7 Execution time with alternative reference-counting (non-zero time origin) . 119
6.8 Execution time with RC-pairs (non-zero time origin) 120
6.9 Effectiveness of qualifiers and qualifier check removal 122

6.10 Execution time with `sameregion`, `parentptr` and `traditional` (non-zero time origin) 123
6.11 Cost of reference-counting local variables (non-zero time origin) 127
6.12 Cost of not using `deletes` qualifier (non-zero time origin) 127
6.13 Other RC overheads (non-zero time origin) 128
6.14 Overhead of using atomic `swap` for pointer writes 129
6.15 RC vs naïve reference-counting 129

List of Tables

6.1 Complexity of benchmark changes, in number of lines changed. 110
6.2 Memory Allocation Rates 113
6.3 Region Statistics 114
6.4 Pointer write statistics 115
6.5 Maximum Memory Usage (in kB) and Overheads 116
6.6 Reference counting overhead in RC and C@ 121
6.7 `sameregion`, `parentptr` and `traditional`: static statistics 121
6.8 `sameregion`, `parentptr` and `traditional`: dynamic statistics 122
6.9 Reference count and runtime check rates 123
6.10 Local variable reference count operation rates 127

Acknowledgements

I thank my parents for bringing me here, and my advisor Alex Aiken for his help and advice. Many thanks also to my officemates over the years, Anders, Jeff, John, Manuel, Megan, Raph and Zhendong, for lots of interesting and fun conversations, and for putting up with the cheese.

To Olga, for the future.

David Gay
December 2001

Chapter 1

Introduction

Much research has been devoted to studies of and algorithms for memory management based on garbage collection or explicit deallocation (as in C's malloc/free). An alternative approach, region-based memory management, has been known for decades, but has not been well-studied until recently. In *region-based* memory management each allocated *object* is placed in a program-specified *region*. Objects cannot be freed individually; instead regions are deleted with all their contained objects. Figure 1.1's simple example builds a list and its contents (the `data` field) in a single region, outputs the list, then frees the region and therefore the list. The `sameregion` type qualifier is discussed below.

Traditional region-based systems such as *arenas* [32] are unsafe: deleting a region may leave dangling pointers that are subsequently accessed. We distinguish two kinds of memory safety: *temporal safety* (no accesses to freed objects) and *spatial safety* (no accesses beyond the bounds of objects). In this dissertation, we design, implement and evaluate *RC*, a dialect of C with regions that guarantees temporal safety dynamically. RC maintains for each region r a *reference count* of the number of *external* pointers to objects in r, i.e., of pointers not stored within r. Calls to `deleteregion` fail if this count is not zero. While our results are presented in the context of a C dialect, we show how our our design and techniques can be applied to other languages, including languages with support for parallelism.

RC does not address the issue of spatial safety. In the rest of this dissertation, we will use the words *safe, unsafe* or *safety* to refer to temporal safety.

```
struct rlist {
  struct rlist *sameregion next;
  struct finfo *sameregion data;
} *rl, *last = NULL;
region r = newregion();

while (...) { /* build list */
  rl = ralloc(r, struct rlist);
  rl->data = ralloc(r, struct finfo);
  ... /* fill in data */
  rl->next = last; last = rl;
}
output_rlist(last);
deleteregion(r);
```

Figure 1.1: An example of region-based allocation.

1.1 Contributions

This dissertation makes contributions in four areas. Firstly, RC is a realistic design for region-based programming in conventional programming languages. Our second contribution is in the area of type systems: RC's design incorporates type information that both makes the structure of region-based programs more explicit and reduces the cost of reference-counting. Thirdly, RC's design led to a set of novel implementation techniques. Our final contribution is a detailed performance study of region-based programming, including a comparison with malloc/free and conservative garbage collection.

1.1.1 Language Design

RC's design, presented in Chapter 3, is based on the lessons learned from an earlier version of C-with-regions, C@ [26]. We have used RC in large applications (Chapter 6.1) and found programming with regions both straightforward and productive. We found that many existing applications could be translated to RC's regions without too much difficulty. We present our experience with this translation process in Chapter 6.3.

Region-based programming is not restricted to C. We also included a similar design for region-based programming in Titanium [64], a dialect of Java designed for parallel, scientific computing (Chapter 3.3).

1.1.2 Types

The major change in RC over our previous system C@ [26], is the addition of static information in the form of three novel type annotations: `sameregion`, `traditional` and `parentptr`. These annotations are based on our observations of common programming patterns in large region-based applications:

- A pointer declared `sameregion` is *internal*, i.e., it is null or points to an object in the same region as the pointer's containing object. Sameregion pointers capture the natural organisation that places all elements of a data structure in one region.

- A pointer declared `traditional` never points to an object allocated in a region, e.g., it may be the address of a local variable. The most important use of traditional pointers is in integrating legacy code into region-based applications.

- In RC, a region can be created as a *subregion* of an existing region. A region can only be deleted if it has no remaining subregions. A pointer declared `parentptr` is null or points upwards in the hierarchy of regions.

These type annotations both make the structure of an application's memory management more explicit and improve the performance of the reference counting as assignments to `sameregion`, `traditional` or `parentptr` pointers never update reference counts. Excepting one benchmark in which reference counting overhead was negligible, we found that between 35% and 99.99% of pointer assignments executed were to annotated types. The correctness of assignments to annotated pointers is enforced by runtime checks (Chapter 3.2.4).

We also designed a type system for dynamically checked regions that provides a formal framework for annotations such as `sameregion`, `traditional` and `parentptr`. Analysis of the translation of RC programs into *rlang*, a language based on this type system, allows us to statically eliminate the checks from many runtime assignments to annotated pointers (Chapter 5). On our benchmarks, between 37% and 99.99% of checks are eliminated.

The combination of type annotations and static elimination of runtime checks reduces the largest reference counting overhead from 22.3% to 12.6% of runtime. For a full discussion of the results of the qualifiers and the qualifier-runtime-check elimination, see Chapter 6.8.

1.1.3 Implementation Techniques

Our dissertation proposes two new variations on the theme of *deferred reference counting* [22], to reduce the cost of reference counting for local variables (Chapter 4.3.2):

- *Lazy stack scanning*, which scans the stack for references to regions when `deleteregion` is called. This technique is suitable when integrating regions into an existing compiler, as it requires knowledge of the stack layout. We used this approach in C@.

- Moving reference-count operations for local variables away from the assignment statements, which allows many of these operations to be eliminated. We found that a simple scheme (placing reference-count operations only around calls to functions that might delete a region) was nearly as good as a provably optimal scheme (see *Function* vs *Optimal* in Chapters 4.3.2 and 6.9). The straightforward approach to handling local variable reference-counts gives overheads up to 25% on our benchmarks. With the *Function* scheme, our highest overhead is 12.6%.

  These approaches only involve moving reference-count operations around, rather than exploiting knowledge of the stack layout, which allows compilation into C. We used this approach in RC, allowing RC to be used on any platform with any C compiler.

To support efficient reference-counting for parallel programming languages, we propose the use of a separate reference-count per thread for each region. This allows us to avoid any synchronisation operations when updating reference counts (though we still need to update pointers via an atomic swap operation). This approach would be prohibitive with traditional reference-counting which has a reference count per object and must often check reference counts, but is quite reasonable with region-based reference-counting where reference-counts are far less numerous and only checked when deleting regions.

1.1.4 Detailed Performance Study

We used the benchmarks of Chapter 6.1 to perform a detailed comparison of region-based programming with malloc/free and conservative garbage collection. We compared memory usage (Chapter 6.6) and performance (Chapter 6.7) of both *unsafe regions* (i.e., with no safety guarantees, implemented without reference counting) and RC (with reference counting) with Doug Lea's high quality malloc/free implementation (see Chapter 6.4) and the Boehm-Weiser conservative garbage collector [13]. We found that safe regions are from

6% slower to 55% faster, and that memory usage is competitive (from 19% less to 4% more) except on applications which need only a few 100kB (up to 2.7x more memory needed for RC's regions).

1.2 Comparison to Other Memory-Management Styles

We compare region-based programming with the two traditional memory management styles, garbage collection and explicit deallocation. We distinguish *tracing garbage collection* (which periodically explores the graph of all reachable objects to identify garbage) from *reference-counted garbage collection* (which keeps a reference count per object, similar to RC's reference count per region) as they have different advantages and disadvantages.

RC's regions are well-suited to real-time use as all operations take an easily predictable amount of time (constant or linear), as discussed in Chapter 4.5. We mention real-time issues as they relate to each style of memory management.

We assume here, and in the rest of this dissertation, basic familiarity with techniques for garbage collection and explicit deallocation. Good overall surveys can be found in Wilson et al's garbage collection [60, 61] and explicit deallocation [62] papers.

1.2.1 Explicit Deallocation

Our region model is reminiscent of malloc/free in that allocation and deallocation are explicit. This gives the programmer increased control over the application, in particular increased control over memory usage.

The biggest problem with malloc/free is the lack of safety. This is a source of many hard-to-find bugs, as the symptoms of a mistaken deallocation of an object o show up at an unrelated point in the program. A problem will only occur when o's memory is used for a new object n, and o is read after a write to n (or vice-versa). RC avoids this problem by preventing deallocation of regions to which references remain. Even unsafe regions reduce the problem of incorrect deallocation to some extent: there are far fewer regions than individual objects, therefore it is easier for the programmer to keep track of these regions and deallocate them at the correct time.

A related problem with malloc/free is memory leaks. It is easy to forget to deallocate objects; typical malloc/free implementations provide no help in finding leaks.

Reference-counted regions can easily provide automatic deallocation of unreferenced regions by periodically checking the reference-counts of all regions and deallocating those whose count is zero. We did not choose to follow this approach in RC as we wished to preserve source-level compatibility with non-reference-counted regions. As with incorrect deallocations, the fact that there are fewer regions than objects helps even unsafe regions avoid leaks to some extent.

The last two points can be summarised as "malloc/free is hard to use". Applications are hard to write as the programmer must carefully figure out where every object will be deallocated, extra code must be written to deallocate data structures (e.g., trees), and bugs are hard to find. A number of commercial tools (Purify [33], CodeCenter [37]) exist to help address these problems, but they have a significant performance cost and do not detect all problems (they only guarantee spatial safety, not temporal safety). Regions reduce the complexity of memory management by reducing the number of entities that have to be managed, making applications easier to write. Reference-counted regions help find deallocation errors where they occur. Our type qualifiers help express a program's memory structure and catch violations of this structure at the assignment where the violation occurs.[1]

Performance is good with malloc/free, but even better with unsafe regions. On the *moss* benchmark, regions (safe or unsafe) are 49% faster than malloc/free because they allow the programmer to optimise the *moss*'s locality and hence reduce cache misses (see the discussion below). RC's safe regions generally have performance competitive with malloc/free (6% to 16% faster), except on *moss* (where RC is 48% faster). As discussed above, memory usage of our regions is generally competitive with malloc/free, except when applications use many small regions.

Malloc/free implementations can be real-time (e.g., the allocator underlying Johnstone's real-time garbage collector [35]).

1.2.2 Tracing Garbage Collection

Deallocation is not explicit with garbage collection, and may occur significantly later than the last use of deallocated objects. This has two causes: garbage collection

[1] A `parentptr` type qualifier helped us find a bug in RC where we had placed an object in the wrong region. Without the qualifier, the program would have failed at the region deallocation rather than at the assignment statement, making the problem harder to find.

happens at infrequent intervals and references to an object may remain even after it is no longer used. This last problem can lead to memory leaks even with garbage collection, as shown by the usefulness of heap profiling tools for garbage-collected languages [43, 45]. Programmers using garbage-collection have little control over object deallocation, which can lead to higher memory usage. Additionally, tracing garbage collectors require some fraction of memory over the application's requirement to perform efficiently. Wilson [61, p58] suggests a typical space overhead of 100%. On our benchmarks, we see space overheads between 44% and 772% for the Boehm-Weiser conservative garbage collector. In comparison, we find overheads between 2% and 174% for unsafe regions (with most benchmarks below 26%), and between 9% and 305% for RC (with most benchmarks below 41%). See Chapter 6.6 for more details.

Garbage collection is easier to use than regions as there is no need to track allocated objects or to write any deallocation code. But, as just discussed, this loss of control leads to increased memory usage and the possibility of memory leaks. In contrast, the control over deallocation of region-based memory management helps reduce space usage. Our reference-counts help detect leaks due to remaining references as these references will cause `deleteregion` to fail. We believe that while region-based memory-management requires more thought when designing a program, this extra thought pays off in better understanding of how objects are used and in reduced memory usage. In converting a garbage-collected program to regions, we found a bug where the application was using old instead of new data. This bug was obvious in the region-based version of the program as the region containing the old data could not be deleted.

Performance of garbage collection is reasonable, comparable to malloc/free and regions on most of our benchmarks (from 2% faster to 13% slower than RC). On one benchmark, garbage collection time is large and RC is 55% faster. Finally, on *moss* RC's locality advantage makes it 36% faster. Wilson [61, p58] suggests that with a good garbage collector an application should spend approximately 10% of its time in garbage collection, which is comparable to our 12.6% overhead for reference-counting. Note however that this 10% figure does not include other costs of garbage-collection (restrictions on pointers, object layout, optimisation, etc). Finally, the causes of safety overhead are different between garbage-collection and reference-counting: garbage collection overhead depends on the rate of allocation, the amount of extra memory available and (for copying collectors) on the size of objects; reference counting overhead depends mostly on the number of pointer writes and

secondarily on the number of pointers per object.

Garbage collection prevents local reasoning about performance by introducing unpredictable pauses. Real-time collectors [6, 35] eliminated this last problem at the cost of higher overhead.

1.2.3 Reference-counted Garbage Collection

Traditional reference-counted garbage-collection [16] does not have the space overheads of tracing garbage collection discussed above. Its space usage should be comparable to malloc/free, except that extra space is needed to store a reference count for every object.

Region-based reference-counting has two advantages over traditional reference-counting:

- Traditional reference-counting does not collect cyclical garbage [40], which can be addressed with a second mechanism to collect cycles [4]. Region-based reference-counting tolerates cycles as long as the objects forming the cycle belong to a single region. RC allow cycles that cross region boundaries to be deleted as long as all regions containing the cycle are deleted together (using `deleteregion_array` function, see Chapter 3.2.3).

- The space cost for storing reference-counts is negligible for regions (4 bytes per region), while it is significant for traditional reference-counting (up to 4 bytes per object, though various schemes [63, 51] can reduce this space overhead).

Reference-counting has been out of favour because of the problem with cycles, though Bacon et al's recent work [4] may change this perception somewhat. Reference-counting collectors have shorter pauses than traditional collectors, but they are generally not real-time as a single pointer write can take an unbounded amount of time if it leads to a large data structure being freed.

1.3 Dissertation Outline

The rest of the dissertation is organised as follows: Chapter 2 discusses more related work; Chapter 3 presents and motivates our design for region-based programming; Chapter 4 discusses the implementation of RC, except for the type annotations and type

system that are in Chapter 5; our benchmarks are presented, and their performance analysed in Chapter 6. Finally, we present our conclusions in Chapter 7.

Chapter 4.5 discusses the changes necessary to make RC's regions real-time. . Of course, malloc/free implementations can also be real-time, but without safety. And as mentioned above, real-time garbage collectors have a significant performance penalty.

Chapter 2

Related Work

We present three strands of related work: other region-based system (Chapter 2.1), other styles of memory-management (Chapter 2.2) and other systems that bring temporal and/or spatial safety to C or C++ (Chapter 2.3).

2.1 Regions

We divide this work into three parts: region-systems based on a region-type system which statically guarantees the safety of `deleteregion` (Chapter 2.1.1), region-systems with dynamic safety (Chapter 2.1.2) and unsafe region systems (Chapter 2.1.3).

2.1.1 Static Safety

The original region type system is part of Tofte and Talpin's *region inference* system [55], which automatically infers for ML programs how many regions should be allocated, where these regions should be freed, and to which region each allocation site should write. Although very sophisticated, the Tofte/Talpin system relies critically on the fact that regions, region allocation, and region deallocation are introduced by the compiler and not by the programmer. Besides being fully automatic, the Tofte/Talpin system has the advantage that the runtime overhead for memory management is reduced to an absolute minimum while also being safe. Unfortunately, region inference is not perfect. To avoid leaking a great deal of memory it is necessary for the programmer to understand the regions inferred by the compiler and to adjust the program so that the compiler infers better region assignments. Second, optimizations beyond the basic inference procedure make an enormous

difference in memory management performance [1, 10]. Both of these properties suggest that explicit first-class regions may be appropriate, but combining explicit programmer-controlled regions with region inference appears to be a very difficult problem.

Tofte and Talpin's type system has been extended by Crary, Walker and Morrisett [19] and again by Walker and Morrisett [57] to allow more flexible region type structures. In particular, Walker and Morrisett [57] propose a form of existentially quantified regions which allows for types such as a list of distinct regions (the types in the earlier systems were restricted to describing structures allocated in a finite set of regions).

Christiansen et al [15] extend C++ to include safe region-based memory management, based on Tofte and Talpin's type system. Class and method types include region annotations, and regions must be allocated in a stack-like fashion as with Tofte and Talpin's region inference. Deline and Fähndrich [20] have designed a programming language, *Vault*, that incorporates Walker and Morrisett's type system and allows static verification of region and other resource usage. Morrisett's Cyclone project at Cornell [25] is similar: it is a C-like language with statically-checked regions based on Walker and Morrisett's type system. Cyclone's data structure representations are designed to interoperate with C. Cyclone includes a garbage-collected heap in addition to region-based allocation.

There are two important differences between the type system of Walker and Morrisett and the type system of rlang (which generalises and formalises RC's type annotations, as detailed in Chapter 5) and hence between Vault or Cyclone (when using regions rather than the garbage-collected heap) and RC:

- Walker and Morrisett's type system can statically verify the safety of `deleteregion`, while rlang's cannot.
- rlang can represent the type structure of any existing program. For instance, the following program cannot be typechecked in Walker and Morrisett's system:

  ```
  region r[n];
  struct data *d[m];
  for (i = 0; i < n; i++) r[i] = newregion();
  for (i = 0; i < m; i++)
    d[i] = ralloc(r[random(0, n)], ...);
  ```

  There is a type for `r`, but no type for `d` in Walker and Morrisett's type system. This code is not very useful, but similar examples are found in real programs, e.g., one of our

benchmarks contains a list of nested environments with each environment allocated in its own region. Declarations are looked up in these nested environments, with the returned pointers stored in a separate data structure.

Our system preserves the safety of `deleteregion` via reference counting. We believe rlang's gain in expressivity, which allows straightforward porting of existing unsafe region programs to RC (even large ones such as the Apache web server) is in most cases worth the loss of static checking of `deleteregion`.

2.1.2 Dynamic Safety

We found that our previous version of C with safe regions, C@, had performance and space usage competitive (sometimes better, sometimes slightly worse) with explicit allocation and deallocation and with garbage collection [26]. C@'s overhead due to reference counting was reasonable (from negligible to 17% of runtime). Our new system, RC, has lower reference count overhead in absolute time and as a percentage of runtime, allows use of any C compiler rather than requiring modification of an existing compiler (`lcc` [24] for C@) and incorporates some static information about a program's region structure.

Stoutamire [49] adds *zones*, which are garbage-collected regions, to Sather [50] to allow explicit programming for locality. His benchmarks compare zones with Sather's standard garbage collector. Reclamation is still on an object-by-object basis.

Bobrow [11] is the first to propose the use of regions to make reference counting tolerant of cycles. This idea is taken up by Ichisugi and Yonezawa [34] for use in distributed systems. Neither of these papers includes any performance measurements.

Real-Time Java [14] is an extension of Java for real-time computing. It includes a version of regions, called *ScopedMemory areas*. A thread *enters* an area A by calling $A.enter(o)$, where o is an object with a `run` method. The area calls $o.run()$, and all subsequent allocations are made from A. When $o.run()$ terminates, the thread *exits* A and allocations revert to the previously entered area. Each thread thus has a *stack* of entered areas, and may enter an area multiple times. If thread 1 creates thread 2, thread 2 inherits a copy of thread 1's area stack. Different threads can enter areas in different orders, e.g., thread 1 can enter area A then B, while thread 2 enters area B then A. The objects in an area A are deallocated when the last thread exits A (this is detected by keeping a count of the number of threads which have entered an area). Temporal safety is guaranteed

by the following rule: a thread may not write a reference of an object in area A into an object in area B if it entered area B after A (note that this means that only the oldest entry on a thread's area stack is relevant for safety checking). Also, references to objects in ScopedMemory areas may not be written to static fields.

This model is reminiscent of RC's subregions: entering an area A from an area B is similar to creating a subregion of B. The restriction on pointers in Real-Time Java is then the same as requiring that all pointers be qualified with RC's `parentptr` type qualifier. At first glance, there is a significant different between Real-Time Java and RC: the fact that different threads can enter the same regions in a different order means that there is no area hierarchy comparable to the hierarchy of regions built by `newregion`/`newsubregion`.

However, at a deeper level this difference disappears: Real-Time Java's rules are such that when a thread t enters an area A that is not already on its area stack it cannot ever share a reference to an object in A with any other thread t', except if t creates t' directly or indirectly[1] before exiting A. The Real-Time Java rules also guarantee that after a thread exits the last entry for an area A on its area stack it cannot refer to any of the objects it created in A.[2] These consequences allows us to emulate Real-Time Java's model with our region model as follows:

- For each thread t and area A, we associate a region A_t. Given two arbitrary threads t and t', A_t may or may not equal $A_{t'}$.
- Allocations in thread t from area A are allocations from region A_t.
- If a thread t, in area B, enters an area A which is not on its area stack: we set $A_t = \texttt{newsubregion}(B_t)$.
- If a thread t creates a thread t', we set $A'_t = A_t$ for all threads A on the area stack of t'.[3]
- If a thread t enters an area A which is already on its area stack, nothing changes.
- All pointers are qualified with `parentptr`.
- When a thread exits an area A which is still on its area stack, nothing happens.

[1] By indirect creation we mean that t creates a thread t'' that creates t' directly or indirectly.
[2] If these two consequences did not hold, Real-Time Java would not have temporal safety.
[3] t' inherited a copy of the area stack of t.

- When a thread exits an area A which has no other entries on the area stack: if some other thread shares A_t, nothing happens (as with standard Real-Time Java areas, this requires keeping a count of references to regions). If this is the last thread using A_t, we delete region A_t (it's reference count will be 0 as all pointers are `parentptr` and no references to A' can remain in any local variables).

This translation does not preserve all the properties of Real-Time Java's area model. For instance, two independent threads sharing a ScopedMemory area are still allocating from the same pool of memory while in the translation above they would get separate regions. But this translation does show that RC's region model is more general than Real-Time Java's, and hence suggests ways that Real-Time Java could be extended to have a more elaborate region model by incorporating other RC features (e.g., the `sameregion` type qualifier, or reference-counting). It also means that some of the techniques we developed for RC can be applied to Real-Time Java: low overhead runtime checks for `parentptr` (Chapter 4.3) and qualifier-runtime-check elimination (Chapter 5.6).

Beebee [59] reports on an implementation of Real-Time Java's regions, and finds that the overhead of runtime checks on assignments is very high (a slowdown of more than 5x on one benchmark). We expect that this overhead could be reduced with an implementation of runtime checks similar to RC's. Sălcianu and Rinard [52] present a pointer and escape analysis which can eliminate all the runtime checks for the benchmarks used by Beebee. Runtime checks for Real-Time Java assignments can be eliminated if the objects allocated in an entered `run` method cannot escape that method. These results are not directly comparable to our results for RC because of the differences in the language, benchmarks and analysis approach.

2.1.3 No Safety

Regions have been used for decades in practice, well before the current research interest. Ross [44] presents a storage package that allows objects to be allocated in specific *zones*. Each zone can have a different allocation policy, but deallocation is done on an object-by-object basis. Vo's [56] Vmalloc package is similar: allocations are done in *regions* with specific allocation policies. Some regions allow object-by-object deallocation; some regions can only be freed all at once. Hanson's [32] *arenas* are freed all at once. Barrett and Zorn [7] use profiling to identify allocations that are short-lived, then place these allocations

in fixed-size regions. A new region is created when the previous one fills up, and regions are deleted when all objects they contain are freed. This provides some of the performance advantages of regions without programmer intervention, but does not work for all programs. None of these proposals attempt to provide safe memory management.

Some well-known applications have been written using unsafe region libraries, e.g., the `gcc`[4] C compiler (before v3) and the apache web server.[5]

2.2 Other Styles of Memory-Management

There have been a number of studies of the performance of memory allocation. Grunwald and Zorn [30] and Detlefs, Dosser and Zorn [21] study the performance of various allocators. Vo's paper on regions [56] also compares the performance of the malloc/free-like allocator of the Vmalloc package with other malloc/free implementations. In these last two papers, Doug Lea's public-domain malloc/free implementation had the best tradeoff between efficiency and space usage. We therefore chose the latest version of this allocator in our comparison of regions to malloc/free in Chapter 6. Grunwald, Zorn and Henderson compare the performance and cache locality of different allocators [31]. None of these studies consider region-based allocation.

We have already extensively discussed the tradeoffs between regions, and the two dominant styles of memory management, garbage collection and explicit allocation and deallocation in the introduction. Detailed surveys of these styles were performed by Wilson et al for garbage collection [60, 61] and for explicit allocation and deallocation [62].

2.3 Safe C Dialects

Other approaches have been used to bring safe memory management to C (or equivalently C++). These can be broadly categorised into language changes (like RC), conservative garbage collection, code instrumentation and interpretation. Interpretation and instrumentation introduce significant performance penalties (execution times are at least doubled in all the systems examined below).

All these systems (except conservative garbage collection) exhibit spatial safety

[4] http://gcc.gnu.org/
[5] http://www.apache.org

(preventing accesses beyond the bounds of objects), but only some provide temporal safety (preventing access to freed objects). Spatial safety alone catches many, but not all, violations of temporal safety: accesses to a pointer p to a freed object are not caught if the freed memory is allocated to a new object before any access to p. RC provides temporal, but not spatial safety. We mention below those systems that do not provide temporal safety.

2.3.1 Language Changes

The Safe C++ language proposal [23] modifies C++ in a way that allows traditional garbage collector implementations. Additionally, programs written in a specific subset of C++ will then be safe. This system has not been fully implemented.

As already mentioned above, the Cyclone project [25] is a safe, C-like language designed to allow easy porting of C applications, and interoperation with existing C code. Unlike Cyclone, RC only brings temporal safety to C code (for instance, it does not check for out-of-bound array accesses), but will run most C code with no changes.

Necula, McPeak and Weimer's CCured system [41] brings type safety to C programs through a mixture of static analysis to find provably safe pointers and runtime-checks for other pointers. Small changes to existing C applications are required when running them with CCured. The design of CCured allows the use of accurate garbage collection, though the current implementation uses the Boehm-Weiser conservative garbage collector to guarantee temporal safety.

2.3.2 Conservative Garbage Collection

Conservative garbage collection [13] allows traditional garbage collection to be used with C programs, without special compiler support.[6] Conservative garbage collection works like a normal garbage collection system but does not have any type information. It assumes that any value that looks like a pointer is in fact a pointer. Thus it may retain objects that are in fact unreachable, and cannot copy objects as it cannot safely modify any values (as these values may not in fact be pointers).

An alternative to purely conservative garbage collection is "mostly-copying collection" [8, 9, 65, 46] which conservatively scans the stack and accurately scans the heap.

[6] In fact, some compiler optimisations could break conservative garbage collection, but these do not occur in practice [12].

Objects that are not apparently referenced from the stack may be moved during garbage collection. The programmer must provide scanning-functions for heap-allocated objects. These scanning functions are similar to the `rc_adjust_x` functions required by RC (Chapter 3.2.5). Smith and Morrisett [46] found that their mostly-copying collector required more memory than the Boehm-Weiser conservative garbage collector, but had a lower runtime overhead.

2.3.3 Instrumentation

Safe-C [3] changes C's pointer type to include enough information (object base, object size and information to identify the object's lifetime) to allow all pointer accesses to be checked for safety. These checks, however, come at a high cost: from 130% to 540% time overhead, and up to 100% space overhead. This compares to RC's 11% time overhead and generally competitive space usage (Chapter 6). Also, Safe-C does not have object-code compatibility with existing C code (e.g., the standard C library) as it changes the pointer representation.

Patil and Fischer [42] use a representation similar to Safe-C's to catch all pointer errors. Their overhead is less than 10%, but is achieved by using a second processor to check for errors. They also use a reference-counted garbage collector to detect memory leaks (also using the second processor).

Purify [33][7] is a commercial product that instruments C code to find spatial safety errors and other problems. Purify does not have temporal safety and has a significant runtime overhead (5x slower, or worse). However, unlike Safe-C, it preserves object-code compatibility with C. Several other systems [47, 36, 39] bring spatial, but not temporal, safety to C. These systems also have significant runtime overheads (no better than Purify).

2.3.4 Interpretation

Saber-C [37] (now called CodeCenter[8]) is a C interpreter that detects most C errors through runtime checks. The freely available EiC C interpreter[9] catches array-out-of-bounds accesses, but does not detect attempts to access freed memory. Neither of these

[7] http://www.rational.com/products/purify_unix/index.jsp
[8] http://www.centerline.com/productline/code_center/code_center.html
[9] http://www.kd-dev.com/~eic/

systems have temporal safety. Another freely available C interpreter, CInt[10], allows for optional use of a garbage collector.

All these interpreter-based systems are of course far slower than compiled C code, or the instrumentation-based systems of the previous section.

[10]http://root.cern.ch/root/Cint.html

Chapter 3

Language Design

Our first design for C with regions was *C@* (Chapter 3.1). The lessons we learned from this prototype were incorporated into our final design for C with regions, *RC* (Chapter 3.2). We have also incorporated region-based memory allocation into *Titanium* (Chapter 3.3), a parallel dialect of Java designed for scientific computation. We end this chapter with a discussion of design alternatives and possible extensions to RC (Chapter 3.4).

3.1 C@

C@ is a dialect of C extended with region-based memory-management. C@ distinguishes two kinds of pointers: normal pointers and *region pointers*, i.e., pointers to objects in regions. Region pointers are defined with '@' instead of '*,' e.g., `int @x`. The types T@ and $T*$ are different types, and no implicit conversion exists between them, although explicit casts are allowed. These casts are unsafe, but are necessary in C@ as the standard C libraries are not aware of region pointers. In particular `deleteregion` does not account for region pointers cast to normal pointers.

We chose to have two kinds of pointers for several reasons. First, region pointers require reference counting, while normal pointers do not. By having a special type for region pointers we avoid any overhead when manipulating normal pointers. Secondly, existing C source code may perform operations that cause problems with region pointers, e.g., casting integers to pointers. By using a separate type for region pointers, we are forced to address these issues when converting a C program to use C@'s region's. Thirdly, existing C object code does not know about regions and reference counting, so should not be passed region

```
typedef struct region @region;
typedef size_t (*cleanup_t)(/* struct ??? @x */);
typedef size_t (*cleanuparray_t)(/*size_t n, struct ??? @x */);

region newregion(void);
int deleteregion(region *r);

void @ralloc(region r, size_t size, cleanup_t cleanup);
void @rarrayalloc(region r, size_t n, size_t size, cleanuparray_t cleanup);
void @rstralloc(region r, size_t size);

region regionof(void @x);
```

Figure 3.1: C@ Region API

pointers. We decided, as a result of our experiences with C@, that the disadvantages of having two kinds of pointers outweighed these benefits. See Chapter 3.2.1 for more details.

Figure 3.1 shows the region interface in C@. A region is created with `newregion`. Objects are allocated with `ralloc`, arrays with `rarrayalloc`. Objects or arrays that do not contain any region pointers can be allocated with `rstralloc`; the `cleanup` arguments to `ralloc` and `rarrayalloc` are discussed below. The memory returned by `ralloc` and `rarrayalloc`, but not `rstralloc`, is cleared. An object's region is returned by `regionof`.

An attempt to delete a region is made by calling `deleteregion(x)`. The deletion succeeds if there are no references (excepting `*x`) to the region in live variables or in other regions. On success, `*x` is set to `NULL`, and `1` is returned. On failure `*x` is unchanged, and `0` is returned.

Figure 3.2 shows a simple example that copies a list into a region, then later deletes that region.

C@ has a number of restrictions and changes from regular C:

- All region pointers must be initialised, as the adjustment of a reference count depends on the old as well as the new value of a pointer. C@ requires that all local variables be initialised and clears the memory for all objects allocated in regions. If an object containing region pointers is allocated with `malloc` or `realloc`, that object should be cleared with `memset` before any of its region pointers are modified.
- C@ forbids the copying of `struct`s or `union`s containing region pointers.

```
struct list {
  int i;
  struct list @next;
};

size_t cleanup_list(struct list @x) {
  destroy(x->next);
  return sizeof *x;
}

struct list @cons(Region r, int x, struct list @l) {
  struct list @p = ralloc(r, sizeof(struct list), cleanup_list);
  p->i = x; p->next = l;
  return p;
}

struct list @copy_list(Region r, struct list @l) {
  if (l == NULL) return NULL;
  else return cons(r, l->i, copy_list(r, l->next));
}

void work(struct list @l) {
  region tmp = newregion();

  l = copy_list(tmp, l);
  ... do something with l ...
  deleteregion(&tmp);
}
```

Figure 3.2: List copy using regions in C@.

- A deleted region r may contain pointers to objects in other regions. To adjust the reference counts of other regions we examine all the region pointers in objects allocated in r. The user supplies the function that performs this task as the `cleanup` argument to `ralloc` and `rarrayalloc`. This function must call `destroy` on every region pointer in the allocated object and return the size of the object. We require the user to provide this function (see `cleanup_list` in Figure 3.2 for an example) for the same reason that we forbid copying `struct`s or `union`s that contain region pointers: the presence of C's `union`s makes it impossible for the compiler to locate every region pointer. For cases without `union` the `cleanup` function could be generated automatically by the

compiler. This is the approach taken in RC.

- C@ does not support threads.
- As mentioned above, casts to and from region pointers are unsafe, as are casts between region pointer types. For instance, casting a `struct list @` type to `char @` and then copying the contents byte-by-byte is unsafe:

```
struct list @l1, @l2;
char @c1 = (char @)l1, @c2 = (char @)l2;
int i;

for (i = 0; i < sizeof(struct list); i++) c1[i] = c2[i];
```

3.2 RC

RC's design was motivated by lessons learned from C@ (Chapter 3.2.1). This design is described in Chapters 3.2.2 (assumptions about C), 3.2.3 (region API), 3.2.4 (new type qualifiers) and 3.2.5 (restrictions in RC not present in C). Chapter 3.2.6 describes the conditions under which RC programs can invoke C object code. Chapter 3.2.7 explains how RC programs can be compiled with a regular C compiler (with the loss of all checks on `deleteregion` and assignments to qualified pointers). Finally, Chapter 3.2.8 discusses debugging RC programs and Chapter 3.2.9 gives some more examples of RC code.

We first formally define the meaning of various concepts used in RC:

- *region*: an unbounded area of memory in which objects can be allocated.
- *subregion*: a region r may be allocated as a *subregion* of a region q. We say that r is a *child* of q, or q is r's *parent*. A region may not be deleted while it has children. We also define *descendant* and *ancestor* regions in the obvious fashion.
- *region reference count*: the reference count of a region r is the number of pointers to objects in r from outside r (i.e., from other regions, local and global variables). References from local variables count as long as the local variable is live. A local variable whose address is taken is considered live while it is in scope. A region may not be deleted if its reference count is nonzero.

- The `traditional` region: RC programs may still use `malloc` and `free`. The memory returned by `malloc`, and memory used for global or local variables is considered to live in the `traditional` region. This region may not be deleted.
- A *qualified* pointer is a pointer qualified with one of the `sameregion`, `traditional` or `parentptr` qualifiers. An *unqualified* pointer has none of these qualifiers.

3.2.1 C@ Lessons

The changes from C@ to RC were motivated by lessons learned from porting the benchmarks of Chapter 6 to C@'s region-based memory management and by some new design goals:

- C@ was implemented as a modification to an existing C compiler. We wished to increase RC's ease of portability by compiling to C. This requires a few restrictions in RC which were not present in C@: `setjmp` and `longjmp` are not supported in RC and variables and `struct` or `union` tags may not shadow each other. RC programs must also use the `deletes` keywords on functions that may delete a region. See Chapters 3.2.4 and 3.2.5 for more details.
- One of our goals in designing RC was to allow RC programs to be compiled with a regular C compiler with an appropriate region-allocation library. This allows RC programs to be run in an environment where an RC compiler is not available, at the expense of the loss of RC's safety guarantees. As a consequence, in RC, `deleteregion` aborts the program if a region still has remaining references rather than returning a failure code as in C@. In our benchmarks at least, we found this to be of no consequence: in the C@ version, all our calls to `deleteregion` had been of the form:

```
if (!deleteregion(&x))
  abort();
```

- In retrospect, C@'s two kinds of pointers were a mistake:
  - They force small syntactic changes (replacing `*` by `@`) all over an existing program. While these are not hard to make, they are pervasive.
  - Even in a program that uses regions for all its memory allocation there are still *traditional* C pointers: addresses of local or global variables, and objects

allocated by the standard C library. Having two kinds of pointers that cannot be mixed forces some code to be duplicated (for region and traditional pointers) and makes by-reference arguments hard to use. For instance, in C@ a function with two results must be declared as

```
either: void f(int *result1, int *result2);
    or: void f(int @result1, int @result2);
```

In the first case, results cannot be placed in a region, in the second they cannot be placed in local variables. In RC there is essentially one kind of pointer that can hold both region and non-region pointers. RC still allows the declaration of pointers that are always traditional with the `traditional` type qualifier (see below).

– The standard C library does not know about region pointers. This forced uses of casts for all pointer parameters in calls to standard library functions. RC's use of a single type of pointer avoids this problem. We document in Chapter 3.2.6 when it is safe to pass region pointers to C object code that was not compiled with RC (as is generally the case for the standard C library).

- Writing the cleanup functions that are passed as the `cleanup` argument to `ralloc` and `rarrayalloc` in C@ is time consuming. For most types, RC generates these functions automatically. The programmer only has to provide these functions for types containing pointers in `unions` (see the discussion of `union` in Chapter 3.2.5).

- We observed that the information provided by the `cleanup` functions can also be used to allow copying of `structs` and `unions` containing pointers. This operation is thus allowed in RC.

- Finally, adding an initialisation to every local pointer variable is annoying. In RC, such variables are automatically initialised to `NULL`. Note that if you wish RC code to be compilable with a regular C compiler then you must not rely on this initial value.

RC also incorporates a number of concepts which were not present in C@:

- subregions: we observed that in several of our benchmarks some regions were guaranteed to have a lifetime strictly contained within that of another region. In fact, the `apache`'s web server's own regions explicitly incorporate this in the notion of *sub-pools*.

```
struct rlist {
  struct rlist *sameregion next;
  struct finfo *sameregion data;
} *rl, *last = NULL;
region r = newregion();
```

```
while (...) { /* build list */
  rl = ralloc(r, struct rlist);
  rl->data = ralloc(r, struct finfo);
  ... /* fill in data */
  rl->next = last; last = rl;
}
output_rlist(last);
deleteregion(r);
```

Figure 3.3: An example of region-based allocation.

By incorporating subregions, we increase RC's expressiveness. In conjunction with the `parentptr` type qualifier (that expresses that a pointer points from a subregion to a parent region) subregions helped us catch a memory bug in RC at a point in the code which made the error obvious.

- Examination of applications written using regions shows that some of their pointers have properties of interest to both the programmer (to make the intent of the program clearer and to catch violations of this intent) and to the RC compiler (to reduce the overhead of maintaining the reference counts). RC allows specification of three such properties: `traditional`, `sameregion` and `parentptr`. See Chapter 3.2.4 for details.
- Finally, RC includes a richer region API which makes programming more convenient: a function to copy arrays, `rarraycopy` (copying objects containing pointers cannot be done with `memcpy`); extendable array support with `rarrayextend`; and some support for writing generic code (the `typed_...` functions). See Chapter 3.2.3 for details.

3.2.2 Assumptions

RC assumes a C implementation where filling an object with zero bytes sets all pointers in that object to `NULL`. This is not guaranteed by the C standard, but is true on the vast majority of machines with a C compiler.

3.2.3 Region Library

A simple RC program is given in Figure 3.3. This example builds a list and its contents (the `data` field) in a single region, outputs the list, then frees the region and

```
typedef struct region *region;

region newregion(void);
region newsubregion(region r);
void deleteregion(region r) deletes;
void deleteregion_ptr(region *r) deletes;
void deleteregion_array(int n, region *regions) deletes;

/* ralloc, rarrayalloc, rarrayextend are not functions
   (they take a type as last argument) */
void *ralloc(region r, type);
void *rarrayalloc(region r, size_t n, type);
void *rarrayextend(region r, void *old, size_t n, type);

/* Miscellaneous allocation functions */
char *rstralloc(region r, size_t size);
char *rstralloc0(region r, size_t size);
char *rstrdup(region r, const char *s);
char *rstrextend(region r, const char *old, size_t newsize);
char *rstrextend0(region r, const char *old, size_t newsize);

/* Out-of-memory handling */
typedef void (*nomem_handler)(void);
nomem_handler set_nomem_handler(nomem_handler newhandler);

/* Miscellaneous functions. rarraycopy is not a function */
region regionof(void *x);
void rarraycopy(void *to, void *from, size_t n, size_t size, type);

/* Dynamic type information and low-level functions.
   rctypeof is not a function */
type_t rctypeof(type);

void *typed_ralloc(region r, size_t size, type_t type);
void *typed_rarrayalloc(region r, size_t n, size_t size, type_t type);
void *typed_rarrayextend(region r, void *old, size_t n,
                         size_t size, type_t type);
void typed_rarraycopy(void *to, void *from, size_t n,
                      size_t size, type_t type);
```

Figure 3.4: Region API

therefore the list.

RC's region API is summarised in Figure 3.4. This API defines two types, `region` and `type_t`. A value of type `region` represents a region, and can be freely stored in the heap, and passed or returned from functions. A value of type `type_t` represents the type information that RC needs for a particular type. For more details, see Chapter 3.2.3.

`ralloc`, `rarrayalloc`, `rarrayextend`, `rarraycopy` and `rctypeof` are not functions (they take a C type as their last argument), but they are presented as functions in Figure 3.4 for simplicity.

This API is defined by the `regions.h` header file. This file is automatically included at the start of every RC file. It can be useful to explicitly include this file however, as this well then allow the RC source code to be compiled with a regular C compiler. See Chapter 3.2.7 for details.

`region_main`

The main function of the program should be called `region_main` rather than `main`.

Creating Regions

A region is created by calling either `newregion()` to create a region with no parent, or `newsubregion(r)` to create a new region as a child of region r. See the 'Out-of-memory handlers' section below for how `newregion` and `newsubregion` behave if they cannot allocate memory for the new region.

Deleting Regions

A call to `deleteregion(r)` deletes region r if it's reference count is zero. If r's reference count is non-zero, the program is aborted. If r is a local variable (whose address is not taken) r is considered to be dead just before the call to `deleteregion` occurs. If r is a global variable, or the address of r is in a region other than r, then the call to `deleteregion` will abort the program as r's reference count is non-zero. In this case one can use `deleteregion_ptr` instead. A call to `deleteregion_ptr(q)` (where `q` is a pointer to a region) deletes the region in `*q` and sets `*q` to `NULL`. Calls to `deleteregion_ptr` fail if `*q` is not the only reference to region `*q`.

Several regions can be deleted at once with `deleteregion_array(n, a)`. This deletes regions `a[0], ..., a[n-1]` and sets `a[0], ..., a[n-1]` to `NULL`. Deleting a group of regions succeeds as long as all references to the deleted regions are in `a` or in one of the deleted regions. If `deleteregion_array` fails then the program is aborted. An example of the use of `deleteregion_array` is given with the discussion of cyclical data structures in the Examples chapter (3.2.9).

See Chapter 3.2.4 for details on the `deletes` function qualifier.

Allocation

An object of type `t` is allocated in region `r` by `ralloc(r, t)`. The type `t` cannot be a function or array type, nor can it be a variable-size type (variable-size types are a `gcc`-specific extension to C). If `t` is (or contains) a `union` type, the programmer may need to provide a reference-count-adjusting function. See Chapter 3.2.5 for details.

An `n` element array of objects of type `t` is allocated in region `r` by `rarrayalloc(r, n, t)`. The restrictions on `t` are the same as with `ralloc`.

The memory returned by `ralloc` and `rarrayalloc` is zero-filled (as with the traditional C function `calloc`). This is necessary for correct reference-counting.

A call to `rstralloc0(r, n)` is equivalent to `rarrayalloc(r, n, char)`. The `rstralloc` function is equivalent to `rstralloc0` except that it does not zero-fill the allocated memory. A call to `rstrdup(r, s)` is equivalent to `strdup(s)` except that the new string is allocated in region `r`.

See the 'Out-of-memory handlers' section below for behaviour if these functions cannot allocate the requested memory.

Extendable Arrays

An extendable array, i.e., an array whose size can be extended (or reduced), is allocated with `rarrayextend(r, old, n, t)`. This creates a new array *new* of type `t` with `n` elements in region `r`. If `old` is `NULL` the new array is zero-filled and returned. If `old` is not `NULL` then it must have been allocated with `rarrayextend` in region `r` and the same type `t`. If `old` had less than `n` elements then its contents are copied to *new* and the extra elements of *new* are zero-filled, if `old` had more than `n` elements then only the first `n` elements of `old` are copied to *new*. The implementation will try to reuse the memory

occupied by `old` for the new object.

A call to `rstrextend0(r, n)` is equivalent to `rarrayextend(r, n, char)`. The `rstrextend` function is equivalent to `rstrextend0` except that it does not zero-fill the allocated memory.

See the 'Out-of-memory handlers' section below for behaviour if these functions cannot allocate the requested memory.

Out-of-memory Handlers

When any of the allocation functions (and `newregion`, `newsubregion`) cannot obtain the memory they need, they attempt to call a *no-memory* handler. This handler is a function with no arguments and no results. If there is no no-memory handler, or the handler returns then the program is aborted.

The no-memory handler h is set by a call to `set_nomem_handler(h)`. This returns the previous no-memory handler. A null value for h means there is no no-memory handler.

A future version of RC will allow for more options than just aborting or exiting the program.

Miscellaneous Functions

The region of a pointer `p` is returned by `regionof(p)`. A call to `rarraycopy(to, from, n, t)` copies `n` elements of type `t` of array `from` to array `to`. The source and destination areas of memory must not overlap. The restrictions on `t` are the same as with `ralloc`. This function is useful as `memcpy` cannot be used on objects containing pointers (see Chapter 3.2.5).

Dynamic Types

The functions described above all require that the type of object being allocated be statically known. In regular C, it is possible to allocate any type of object using `malloc` as long as the size of the object is known. This is useful when writing generic types, e.g., an automatically expanding array. To allow similar code to be written in RC we introduce the `rctypeof(t)` syntax (similar to `sizeof`) which returns the type information needed by RC when allocating objects. The restrictions on the type t passed to `rctypeof` are the same as for `ralloc` (Chapter 3.2.3).

The function `typed_ralloc` behaves identically to `ralloc`, except that the type to be allocated (or copied) is specified by its size and `type_t` value. In fact, `ralloc` is defined in terms of `typed_ralloc` as follows:

```
#define ralloc(r, type) typed_ralloc((r), sizeof(type), rctypeof(type))
```

The functions `typed_rarrayalloc`, `typed_rarrayextend` and `typed_rarraycopy` have the same relation to `rarrayalloc`, `rarrayextend` and `rarraycopy` as `typed_ralloc` has to `ralloc`.

3.2.4 Type Qualifiers

RC introduces three new pointer type qualifiers (`sameregion`, `traditional` and `parentptr`) and one new function type qualifier (`deletes`). The syntax for function qualifiers is the same as in C++, i.e., they follow the argument list:

```
int f(int x) deletes;
int f(int x) deletes
{
  return x + 1;
}
```

Function Type Qualifiers

RC requires that any function which may delete a region (including by calling another function that may delete a region) be annotated with the `deletes` qualifier. For instance, the following code will produce a compile-time error:

```
void f(region r)
{
  deleteregion(r);
}
```

as `f` must have the `deletes` annotation. Formally, the rule is that any function that calls a function with the `deletes` qualifier must itself have the `deletes` qualifier.

The `deletes` annotation is part of a function's type, so the following code will produce a warning (this is consistent with `gcc`'s behaviour on similar type errors):

```
void f() deletes;
void (*g)();
g = f; /* error: drops the deletes qualifier */
```

The requirement for `deletes` qualifiers is necessary for efficient reference counting in the presence of separate compilation.

Pointer-type Qualifiers

The `traditional` type qualifier (`int *traditional x`) declares that a pointer is null or points into the traditional region. Updating a `traditional` pointer never changes any reference counts. The compiler guarantees, by static analysis or by insertion of a runtime check (whose failure aborts the program), that only pointers to the traditional region are written to `traditional` pointers. Pointers declared `traditional` can be used in any portion of a program where there is a need, for whatever reason, to use conventional C memory management. Also, pointers to functions are `traditional`.

We found that complex data structure are often intended to be allocated in a single region. In that case, the data structure's internal pointers will never point to another region (e.g., the `next` and `data` fields of `rlist` in Figure 3.3). The `sameregion` type qualifier declares that a pointer stays within the same region or is null. As with the `traditional` annotation, writes to `sameregion` pointers do not change any reference counts (they do not create or destroy any references from outside the region). The compiler ensures, as for `traditional` pointers, that values written to `sameregion` pointers are either null or belong to the correct region.

When using subregions to structure an application's memory management, many pointer types always point to an object in the same region or a parent, grandparent, ... region. Such pointers can be specified with a `parentptr` type qualifier. Pointers from `parentptr` qualified pointers need not be included in the reference counts as RC requires that subregions be deleted before their parent regions. As with the other qualifiers, the compiler enforces by static analysis or a runtime check that all assignments to `parentptr` fields are correct.

It is not possible to use more than one of these qualifiers on any given pointer, e.g, `int *sameregion traditional x` is an error. However different qualifiers can be used for different pointers in the same type, e.g., `int *sameregion *traditional x`.

3.2.5 Restrictions

RC imposes a number of restrictions on the C programming language. Most of these are to allow the tracking of pointers necessary for accurate reference counting. A few are due to the decision to output C code and compile this code with an existing C compiler. We start with a summary of the restrictions, each restriction is then detailed in a separate section:

- RC does not support threads.
- RC does not allow use of `setjmp` and `longjmp`.
- Pointer-typed objects may only contain valid pointers.
- Pointer-values cast to an integral type are not included in RC's reference counts and are not considered when deleting regions.
- Objects containing pointers may not be written byte-by-byte (i.e., via a `char *` pointer). This typically affects `memcpy` and `memset`.
- `malloc` cannot be used to allocate objects containing pointers. Use `calloc` instead.
- Local variables may not *shadow* (i.e., have the same name as a local or global variable in an enclosing scope) other variables. Tags (for `struct`, `union`, `enum`) may not shadow enclosing tags either.
- For every type t which is (or contains) a `union` containing pointers, the programmer must provide a function that adjusts the reference counts of regions referenced by pointers in t.
- All local pointer variables are implicitly initialised to NULL (this is not a restriction of course, but is a change from C).

Finally, the new type qualifiers, and the macros in `regions.h`, increase the set of reserved words in C.

Threads

Efficient reference-counting in the presence of threads would require integration of RC's extensions (and knowledge of threads) into the C compiler. We have chosen not to sup-

port threads in RC. Chapter 4.4 discusses the implementation consequences of parallelism on region reference-counting in detail.

setjmp and longjmp

As with threads, efficient reference-counting in the presence of `setjmp` and `longjmp` would require integration of RC's extensions into the C compiler (Chapter 4.3.3 discusses how to implement `setjmp` and `longjmp` assuming compiler support). RC programs that use `setjmp` and `longjmp` will find that the reference counts for regions may become incorrect.

Pointers and Integers

Pointer arithmetic is fully supported, but pointer-typed variables, fields, etc must contain a valid pointer at all times. Casting an arbitrary integer to a pointer may cause incorrect behaviour. But, assuming a large-enough integer type, a pointer can be cast to integer and later back to a pointer with no problems. While stored as an integer, the pointer value will not affect any reference counts. In the following code,

```
void f(void) deletes {
  region r = newregion();
  int *x = ralloc(r, int);
  long y = (long)x;

  deleteregion(r);
  x = (int *)y;
}
```

the call to `deleteregion` will succeed as both `r` and `x` are not live when the call occurs. The variable `y` is live but integers are ignored in reference counts.

Copying Objects with char * Pointers

Copying an object containing an unqualified pointer with byte-by-byte operations will lead to incorrect reference counts. For instance:

```
void *x0, *x1;
char *y0 = (char &)&x0, *y1 = (char &)&x1;
int i;
for (i = 0; i < sizeof(void *); i++) y0[i] = y1[i];
```

is allowed in C, but not in RC.

Copying an object containing qualified pointers with byte-by-byte operations will bypass the usual runtime check on assignment to these pointers.

Such copying is most often done with memcpy (or bcopy). Uses of memcpy should be examined and can often by replaced with calls to rarraycopy. Similarly, uses of memset (or bzero) should be examined and replaced if necessary.

malloc, realloc

If part of an application still wishes to use malloc-style allocation, any allocations of objects containing unqualified pointers should use calloc instead. This is necessary to guarantee that all pointer-typed fields, etc contain a valid pointer at all times. An alternative (necessary for realloc) is to clear the newly allocated memory with memset. See Appendix A for details.

union

For every type that is, or contains, a union with unqualified pointers, the RC programmer must provide a function to correctly adjust the reference counts. This is necessary as the RC compiler has no way of knowing which field of the union is currently being used and therefore does not know if the pointer(s) in the union are valid or not.

For instance, in the type

```
struct intptr {
  enum { an_int, a_ptr} kind;
  union {
    int i; /* valid if kind == an_int */
    void *p; /* valid if kind == a_ptr */
  } u;
  int *stuff;
  int *sameregion y;
  double some_nonpointer_data;
};
```

the u union contains a pointer if and only if the kind field is a_ptr. The stuff field is always a valid pointer. To communicate this information to the RC compiler, the programmer must provide the following function:

```
size_t rc_adjust_intptr(void *x, int by)
{
  struct intptr *p = x;
  RC_ADJUST_PREAMBLE;

  if (p->kind == a_ptr)
    RC_ADJUST(p->u.p, by);

  RC_ADJUST(p->stuff, by);
  /* No adjustment for p->y as it is sameregion */

  return sizeof *p;
}
```

The rules for these `rc_adjust_...` functions are as follows:

- If the program uses (assigns, allocates) a `struct` or `union` with tag *tag* type containing a `union` with unqualified pointers then the programmer must write an `rc_adjust_`*tag* function. A declaration for this function must precede the assignment or allocation site.
- The skeleton of this function must be:

  ```
  rc_adjust_tag(void *x, int by)
  {
    struct/union tag *p = x;
    RC_ADJUST_PREAMBLE;

    /* RC_ADJUST statements go here */
    return sizeof *p;
  }
  ```
- There must be a statement `RC_ADJUST(`*f*`, by)` for every valid unqualified pointer field *f* that is part of the `struct` or `union` type *tag*.

Shadowing

If shadowing of local variables or tags were allowed, the C code generated by RC might mistakenly refer to the wrong local variable. This could be fixed by renaming the

local variables in the output C code, but this would make source-level debugging unpleasant. Therefore RC forbids all shadowing of global, parameter or local variables by other parameter or local variables. Similarly, `struct`, `union` or `enum` tags cannot be redeclared inside a function. For instance,

```
/* global */
int x;
struct a { int z; };

void f(double x /* error */)
{
  struct a { double y; }; /* error */

  {
    void *x; /* an error (shadows both parameter and global) */
  }
}
```

3.2.6 Linking to Existing Object Code

RC programs can call object code produced by a regular C compiler as long as the following simple condition is met: the called code must not write any pointers to any variable or section of memory that is also written by the RC program. Thus most C library functions, such as `printf`, `fopen`, (generally only available as object code) can be called as is.

Note that this rule allows the called C code to save a pointer it is passed in its own memory and variables. However, these saved pointers will not be included in any region's reference count.

Appendix A summarises the compatibility rules for RC when calling the standard C library.

3.2.7 Fallback to C

An RC program can be compiled with a regular C compiler as follows:

- The `deletes`, `traditional`. `sameregion`, `parentptr` qualifiers must be defined as the empty string, e.g., by including the following options when compiling:

  `-Ddeletes= -Dtraditional= -Dsameregion= -Dparentptr=`

- The `regions.h` supplied with RC's distribution in the `rc/libcompat` directory be included in every RC file. This can be accomplished with a `#include <regions.h>` in every RC file, which will be redundant when compiling with RC's compiler.

- The application must be linked with `rc/libcompat/regions.c`.

The RC compiler, written in RC, is itself compiled with a regular C compiler in this way. See `Makefile.in` in RC's distribution for a detailed example.

3.2.8 Debugging Support

RC programs can be debugged with source level debuggers with no restrictions. There is also some support for tracking down references which prevent a region from being deleted. Within the debugger, one can:

- Print a region's reference count.

- Find the region of an arbitrary pointer.

- Scan memory to find pointers to a region r.

It is also possible to compile RC programs with reference counting from local variables disabled. If the region can now be deleted, then the problematic references are in local variables and can usually easily be found by examining the local variables in the call stack.

3.2.9 Examples

This section shows some data and control structures for programs using regions. These are mostly distilled from the benchmarks of Chapter 6.

Complex Data Structures

Some data structures must be represented by more than one object, even though the various objects form one logical whole. In a region-based system, all these objects will be allocated from the same region. In RC, the internal pointers for this data structure can then be declared `sameregion`. For instance, a multi-precision fraction might look like:

```
struct fraction {
  int numerator_length, denominator_length;
  int *sameregion numerator, *sameregion denominator;
};

struct fraction *allocate_fraction(region r, int nl, int dl)
{
  struct fraction *f = ralloc(r, struct fraction);

  f->numerator_length = nl;
  f->denominator_length = dl;
  f->numerator = rarrayalloc(r, nl, int);
  f->denominator = rarrayalloc(r, dl, int);

  return f;
}
```

Circular Data Structures

Many data structures have cycles, e.g., because of pointers back from children to parents. These can easily be accommodated with RC's reference-counted regions as long as the data structure as a whole is contained in a single region. For instance, a binary tree with backwards pointers from child to parent looks like:

```
struct tree {
  int data;
  struct tree *sameregion left, *sameregion right;
  struct tree *sameregion parent;
};
```

and elements can be added to an existing tree with the following function:

```
struct tree *add_left(struct tree *parent, int newdata) {
  struct tree *new = ralloc(regionof(parent), struct tree);
  new->data = newdata;
  new->parent = parent;
  parent->left = new;

  return new;
}
```

Note also that as an element is being added to an existing data structure we can use `regionof` to be sure of allocating the new element in the correct region. As all the pointers

stay within the same region, they are not included in the region's reference count so the tree's region can be deleted without, e.g., clearing the `parent` pointers.

If a circular data structure spans two regions `r1` and `r2` then both regions can be deleted simultaneously using `deleteregion_array` without breaking the cycles.

Temporary Data Structures

A temporary data structure is often built out of pointers to a more permanent data structure. For instance, we might want to make a list of the tree elements from the previous example. We can use sub regions, and the `parentptr` type qualifier to enforce the concept that this list is a "child" of the tree:

```
struct treelist {
  struct treelist *sameregion next;
  struct tree *parentptr element;
};

struct treelist *add_to_treelist(region r, struct tree *element,
                                 struct treelist *next)
{
  struct treelist *new = ralloc(r, struct treelist);

  new->element = element;
  new->next = next;

 return new;
}

void process_tree(struct tree *t) deletes
{
  region r = newsubregion(regionof(t)); /* region for treelist */
  struct treelist *list = NULL;

  while (element = pick_an_element_from(t))
    list = add_to_treelist(r, element, list);

  do_something_with(list);

  deleteregion(r);
}
```

Note that process_tree requires deletes as it deletes a region. While this example is somewhat contrived, the RC compiler exhibits a more complex example of this idea: it builds a temporary graph out of nodes from the abstract syntax tree representing the RC program.

This example also illustrates a common control structure: a region (in this case r) used to hold temporary data needed for some computation and which is deleted when that computation is finished.

Phase-based Computations

Many programs can naturally divide their work into nearly independent pieces or phases. For instance, a compiler mostly compiles a function at a time and needs to keep little data between functions. Such a compiler can use the following region structure: a permanent region that stores the data that must be kept between functions, and a per-function region that stores all the data needed for compiling a single function. The per-function region is created when the compilation of a function starts, and deleted when it ends:

```
region permanent;

void compile(void) deletes
{
  permanent = newregion();

  while (some functions remain) {
    region per_function = newsubregion(permanent);

    compile_next_function(per_function);

    deleteregion(per_function);
  }

  deleteregion(permanent);
}
```

Note also the use of newsubregion to create the per-function region.

Iterative Computations

A slightly more complicated region control structure is found in many iterative computations: the data at each stage depends on the results from the last few stages. This

situation often occurs in scientific computations. A similar structure to the one above can be used, but the data for a particular step can only be deleted when no later stage needs it. For instance, if each stage needs the data from the previous stage, the control structure would look like:

```
void iterate_until(double end_time, double step)
{
  double current_time = 0;
  region last_region = NULL, current_region;
  struct data *current, *last;

  while (current_time < end_time) {
    current_region = newregion();
    current = compute_from(last);
    last = current;

    /* data from last (in last_region) not needed anymore */
    if (last_region)
      deleteregion(last_region);
    last_region = current_region;

    current_time += step;
  }
  /* do something with the result data in current */
  /* and then... */
  deleteregion(current_region);
  if (last_region)
    deleteregion(last_region);
}
```

3.3 Titanium

Titanium [64] is a language and system for high-performance parallel scientific computing. Titanium uses Java [29] as its base, thereby leveraging the advantages of that language. Titanium features a number of changes from Java; here we will only describe Titanium's region-based memory management.

Java uses garbage collection to reclaim unreachable storage. Titanium retains this mechanism but also includes explicit region-based memory allocation, as shown in Figure 3.5. Each iteration of the loop allocates a small array. The call `r.delete()` frees all arrays.

```
class A {
  void f() {
    PrivateRegion r = new PrivateRegion();

    for (int i = 0; i < 10; i++) {
      int[] x = new (r) int[i + 1];
      work(i, x);
    }
    try {
        r.delete();
    }
    catch (RegionInUse oops) {
        System.out.println("oops - failed to delete region");
    }
  }
  void work(int i, int[] x) { }
}
```

Figure 3.5: An example of region-based allocation in Titanium.

Titanium's region model is missing some of the features of RC (subregions, type qualifiers), but these could easily be added. The major change in Titanium from RC is the support for parallelism. We briefly summarise Titanium's model of parallelism in Chapter 3.3.1, then extend regions to deal with this model of parallelism in Chapter 3.3.2. Chapter 3.3.3 shows that region-based allocation fits cleanly into Java's syntax and semantics. We end this section with a more elaborate example of region programming in Titanium (Chapter 3.3.4).

3.3.1 Parallelism in Titanium

Titanium has an *SPMD* model of parallelism, rather than Java's thread-based parallelism: at program startup a fixed number n of processes are created which run in parallel until the program completes. An important aspect of SPMD programming is *global operations*: these operations must be executed by all processes. *Barrier synchronisation* is an example of a global operation: the first $n-1$ processes to execute the barrier synchronisation statement wait until the nth process executes the barrier synchronisation statement. Then all n processes resume execution:

```
for (int i = 0; i < 10; i++) {
  // No process executes work1 in parallel with work2 as processes
  // wait at the barrier until they have all finished work1 or
  // work2 respectively.
  work1();
  Ti.barrier();
  work2();
  Ti.barrier();
}
```

On some platforms, e.g., networks of workstations, each process of a Titanium program has its own *local memory*. Access to the local memory of other processes is still possible (Titanium has a shared-memory programming model), but is much slower.

3.3.2 Shared and Private Regions

There are two kinds of regions in Titanium: *shared* regions and *private* regions. Objects created in a shared region are called *shared objects*; all other objects are called *private objects*. Garbage-collectible objects are taken to reside in an anonymous shared region. It is an error to store a reference to a private object in a shared object. As a consequence, it is impossible to obtain a private pointer created by another process.

The processes of a Titanium program may create and delete private regions independently. But creating and deleting shared regions are global operations. This makes it easy to implement Titanium's regions efficiently on machines where access to the local memory of other processes is slow: each process can keep a separate reference count for a region r. Deleting a region is safe if the sum of all these reference counts is zero, which is easily checked as all processes must cooperate to delete a region. More details on this implementation can be found in Chapter 4.4.

When creating a shared region, each process gets a separate object (in its local memory) that represents the region. The object that represents the shared region created by a process p is called the *representative* of that region in process p (see the `Object.regionOf` method below). The objects allocated by a process p are always placed in p's local memory.

While a program can be written solely with shared regions, the fact that creating and deleting these regions is a global operation may be inconvenient when a process is engaging in purely local computation. In that case it is easier to use private regions.

3.3.3 Region-Based Allocation in Titanium

Shared regions are represented as objects of the `ti.lang.SharedRegion` type, private regions as objects of the `ti.lang.PrivateRegion` type. The signature of the types is as follows:

```
package ti.lang;

final public class PrivateRegion extends Region {
  public PrivateRegion() { }
  public void delete() throws RegionInUse;
};

final public class SharedRegion extends Region {
  public SharedRegion() { }
  public void delete() throws RegionInUse;
};

abstract public class Region {
};
```

The following changes are made to Java (T is any type but `ti.lang.PrivateRegion` and `ti.lang.SharedRegion`):

- `new ti.lang.PrivateRegion()` or `new ti.lang.SharedRegion()`: creates a region containing only the object representing the region itself. Creating a `SharedRegion` is a global operation and implies a barrier synchronisation.
- `new T...`: allocate a garbage-collected object, as in Java.
- `new (expression) T...` creates an object in the region specified by `expression`. The static type of `expression` must be assignable to `ti.lang.Region`. At runtime the value v of `expression` is evaluated. If v is:
  - `null`: allocate a garbage-collected object, as in Java.
  - an object of type `ti.lang.PrivateRegion` or `ti.lang.SharedRegion`: allocate an object in region v.
  - In all other cases a runtime error occurs.

- The `delete` method deletes a region. For `SharedRegions`, this is a global operation, and implies a barrier synchronisation. A region is said to be *externally referenced* if there is a reference to an object allocated in it that resides in
  - A live local variable;
  - A static field;
  - A field of an object in another region.

  The process of attempting to delete a region r proceeds as follows:

  1. If r is externally referenced, throw a `ti.lang.RegionInUse` exception.
  2. Run the `finalize` methods of all objects in r for which it has not been run.
  3. If r is now externally referenced, throw a `ti.lang.RegionInUse` exception.
  4. Free all the objects in r and delete r.

- The class `java.lang.Object` is extended with the following method:

  ```
  public final ti.lang.Region regionOf();
  ```

  This returns the region of the object, or `null` for garbage-collected objects. For shared objects, the local representative of the shared region is returned.

Garbage-collected objects behave as in Java. In particular, deleting such objects differs from the description above in that finalization does not wait for an explicit region deletion.

3.3.4 Titanium Example

Figure 3.6 shows a more complex Titanium programming example, abstracted from a gas dynamics code. The n Titanium processes call the `doWork` method simultaneously, and create an array of `Level` structures. Each `Level` on each processor creates its own data (in this simplified example this is just an array of `doubles`), but in a region shared across all processors. The processes then call the Titanium array primitive `exchange` to get a pointer to every other processor's array (using `numProcs` method to get the value of n).

Each `Level` structure has its own region, this allows each `Level`'s data to be deleted and recreated independently of the other levels by the `refine` method. The main

```
class Level {
  SharedRegion myRegion;
  double[] myData;
  double[][] allData;

  Level () {
    myRegion = new SharedRegion();
    myData = new (myRegion) double[100];
    allData = new double[Ti.numProcs()][];
    allData.exchange(myData);
  }

  void refine(Level[] allLevels, double current_time) {
    SharedRegion oldRegion = myRegion;
    int myself = Ti.thisProc();
    myRegion = new SharedRegion();
    double newData = new (myRegion) double[100];

    // compute newData from allLevels, current_time and this level
    ...

    /* and switch to newData */
    myData = newData;
    allData.exchange(myData);
    try {
      oldRegion.delete();
    } catch (RegionInUse oops) { fail(); }
  }
}

class LevelArray {
  void doWork() { // executed by all processes in parallel
    SharedRegion me = new SharedRegion();

    Level[] allLevels = new (me) Level[4];
    for (int i = 0; i < 4; i++)
      allLevels[i] = new (me) Level();

    for (double t = 0; t < end_of_the_universe; t += 1)
      for (int i = 0; i < 4; i++)
        allLevels[i].refine(allLevels, t);
  }
}
```

Figure 3.6: A larger Titanium example

loop of `doWork` iterates over time and refines each level independently. Note that in this program, the creation and deletion of the shared regions, and the calls to `exchange` are all global operations.

3.4 RC Extensions

The design choices in C@, RC and Titanium are far from exhausting possible variations in region-based memory management. Some of these possibilities have been explored by others and are discussed in related work (Chapter 2). Here we discuss some of the ideas we considered for RC, but did not implement. These ideas come in three groups: alternative semantics for `deleteregion` (Chapter 3.4.1), more elaborate type annotations (Chapter 3.4.2), and extensions to express locality (Chapter 3.4.3).

3.4.1 Alternative semantics for `deleteregion`

There are many possible semantics for deleting regions. They differ in how the system knows when to delete a region, and what action the system takes when deleting a region fails because there are remaining references to the region(s) being deleted.

The first choice is implicit region deletion: at regular intervals, the system checks every region's reference count and deletes those whose count is zero. This approach is most similar to garbage collection, especially reference-count-based garbage collectors [60]. While this check could be made at every pointer update (as in reference-count-based garbage-collection), we think that this is too expensive. Instead, such a system could check for regions with a zero reference count when it is running out of memory, or at every nth memory allocation (this is closer to the approach of traditional mark-and-sweep or copying collectors).

The choice taken in C@ and Titanium is to make `deleteregion` an explicit operation and return a failure indication if the region cannot be deleted. For instance, a library routine could take a region for its temporary allocation and attempt to delete it before returning.

In RC, we preferred to make `deleteregion` fail by aborting the program. We found that this was the natural approach in all our benchmarks, and it allows migration to a system with unchecked regions (one of our design goals for RC).

With subregions, an obvious extension is to delete child regions automatically when their parent is deleted. The rules for reference counting should also be redefined to not count references from a child, grandchild, ... of r in r's reference count. This allows cycles, e.g., between a parent and a child as long as the child is not deleted before the parent.

We did not choose to follow this approach in RC as it increases the overhead of reference counting (see the discussion of its implementation in Chapter 4.3.5 and the "xpp" results in Chapter 6.7.2). Instead, we provide a more general operation, `deleteregion_array`, that deletes an arbitrary group of regions, allowing cycles within this group. We can implement this operation with no additional overhead as long as failure of `deleteregion_array` aborts the program. See the discussion of the implementation of `deleteregion_array` in Chapter 4.3.1 for more details.

3.4.2 Further Type Annotations in RC

We considered adding a `childptr` annotation to RC to declare pointers that point from a region r to an object in a descendant region. We did not incorporate this qualifier as it would require both a runtime check and a reference-count update and would thus slow programs down. However it can increase the number of `parentptr` qualifiers that can be statically checked, as in the following code:

```
struct f {
  struct g *childptr child;
} *a;
struct g {
  struct f *parentptr parent;
};

a->child->parent = a;
```

With the `childptr` annotation, our qualifier-verification framework (see Chapter 5.6) can verify this assignment and eliminate the runtime check. Without it, it cannot.

We also considered adding a way to specify that two pointers are *linked*, i.e., that both always point to the same region. We saw a few instances of this pattern in our benchmarks. We did not do so for several reasons:

- The syntax for this is not immediately obvious. A declaration of some variable or field x could be followed by an annotation such as `sameregionas(y)` but what are

the rules for y ? Allowing an arbitrary expression would be confusing and hard to analyse.

- Once the pointers have been set to some region r, they would both have to be set to NULL before they could be changed to a different region r': the first update of one of a pair of linked pointers will break the invariant that they point to the same region. An analysis (or some special syntax) to detect consecutive updates of linked pointers did not seem worthwhile.

- It is already possible to express this pattern using sameregion and a small change to the code. For instance, if two global variables x and y are linked, the programmer can instead write the following code:

```
struct xandy {
  char *sameregion x, *sameregion y;
} *xandy;

/* when setting x and y's value to some new region r */
xandy = ralloc(r, struct xandy);
xandy->x = rarrayalloc(r, 10, char);
xandy->y = rarrayalloc(r, 20, char);

/* when just modifying one of x or y */
xandy->x = rarrayalloc(regionof(xandy), 30, char);
```

In functions such as new_rlist:

```
struct rlist {
  struct rlist *sameregion next;
  int i;
};

struct rlist *new_rlist(region r, struct rlist *next)
{
  struct rlist *new = ralloc(r, ...);
  new->next = next;
  return new;
}
```

the programmer intends that the parameter next be NULL or in the same region as r. This kind of requirement can be expressed with a function type qualifier such as

```
struct rlist *new_rlist(region r, struct rlist *next) sameregion(r, next)
```

This requires that both parameters be in the same region if both are not NULL. The compiler will ensure either statically or through a runtime check that this requirement holds at every call to new_rlist. This pattern is common, e.g., it occurs frequently in the RC compiler. It is not possible to eliminate the region parameter r and replace the allocation of the new object with

```
rlist *new = ralloc(regionof(next), ...);
```

because next may be NULL.[1] Another possible function annotation, parentptr(p, q), expresses the fact that the p argument must be in an ancestor region of q.

A language designed from scratch to use regions could assume that by default all pointers are sameregion and require parentptr or crossregion (points anywhere) declarations when pointers point to other regions. The crossregion annotation would make it explicit that assignments to such pointers are more expensive as they require a reference-count operation. Local and global variables should probably be implicitly crossregion in this model. Bacon et al's Guava language [5], a dialect of Java that statically prevents data races, has something of this flavour: Guava distinguishes *monitors* which can be referenced from any thread and *objects* which can only be referenced from the thread that created them. This is akin to having one region per thread and annotating all object references with sameregion and all monitor references with crossregion.

3.4.3 Expressing Locality

As discussed in the introduction, and in Stoutamire's dissertation [49], regions can be used to express some locality properties. RC's regions can also be used in this way, and RC's implementation places objects allocated in different regions in different pages of memory. However, this can lead to placing objects that belong naturally (from the point of view of object lifetime) in the same region into two (or more) regions. This then prevents the use of our type qualifiers such as sameregion and forces the use of deleteregion_array to delete the two regions.

Instead, RC's region model could be extended with *areas*: each region would have a number of areas in which allocation can occur, but all areas of a region would be logically in the region and would share a single reference count. The implementation would then

[1] In a new language it would be possible to have a separate null value for each region, which would allow this idiom to work.

guarantee that objects allocated in separate areas would live in separate pages of memory. This approach would separate the locality and lifetime aspects of regions.

Chapter 4

Implementation Techniques

Chapter 4.1 discuss the tradeoffs between compiling RC to C versus integrating knowledge of regions into an existing compiler. This choice has little effect on the implementation of the region library (Chapter 4.2), but strongly influences the implementation of reference counting (Chapter 4.3). It also restricts implementation choices for reference-counted regions in parallel programming languages, including C-with-threads (Chapter 4.4). At the end of this chapter, we show how reference-counted regions can be used for a safe, real-time language (Chapter 4.5).

4.1 Compiling to C

The basic tradeoff is that compiling to C increases portability of the RC compiler, but reduces implementation options and hence performance. However, much of the code necessary to implement RC can be expressed in C: the region library itself, the basic reference count update operation on unqualified pointer writes and the runtime checks for qualified pointer writes can all be written in C. The extensions to C available with the `gcc`[1] compiler (global register variables, statement blocks in expressions) can optionally be used to bring the performance of generating C code even closer to what could be obtained by integrating RC into an existing C compiler and generating assembly code.

The main disadvantage of compiling to C is the lack of information on stack layout and register usage. This prevents the RC compiler from scanning the stack for pointers to regions in `deleteregion` as was done in C@. Chapters 4.3.3 and 4.3.4 contrast the

[1] http://gcc.gnu.org

approaches for handling reference counts from local variables in C@ and RC. Compiling to C also restricts the options open to a reference-counted-region compiler for a parallel language (Chapter 4.4.2).

4.2 Region Library

The region library must support the region API of Figure 3.4. To preserve compatibility with C, RC keeps the same data representation as C, including for pointers. The implementation of reference counting, and of `regionof`, need to map a pointer to the region of the pointed object. Therefore the region library must maintain a data structure that supports `regionof`. Beyond these basic requirements:

- Memory allocation and deallocation should be efficient, especially allocation of small objects.
- The space overhead for regions should be low. This overhead has two sources: actual overhead for regions and each object in a region, and losses due to *internal* and *external fragmentation* [62].
- The region library should provide whatever support is necessary for efficient reference counting.
- The library should be easy to port to a new platform or C compiler.

This led to the following three-level design: a region is built out of allocators; an allocator can allocate arbitrary-sized objects, where each object can have an arbitrary-sized header; allocators obtain *blocks* of memory from the *page allocator*. Each of these three components is described in detail in the next sections.

4.2.1 Regions

A region, whose structure is shown in Figure 4.1, is composed of a reference count and two allocators, the `normal` allocator for objects containing unqualified pointers, and the `pointerfree` allocator for all other objects. When deleting a region, references from the now dead region to other regions are removed by scanning all the objects allocated by the `normal` allocator, using type information recorded when the objects were allocated. The

```
struct region {
  int rc, id, nextid;
  struct allocator normal;
  struct allocator pointerfree;
  struct region *parent, *sibling, *children;
};
```

Figure 4.1: Region structure

blocks of the `pointerfree` allocator need not be scanned as their pointers (all qualified) are not included in any region's reference count.

Objects allocated in the `normal` allocator have a header to allow implementation of this scan operation: for non-array objects this is a pointer to the `rc_adjust_x` function described in Chapter 3.2.5 (the RC compiler generates this function automatically for objects that do not contain pointers in `unions`). For array objects, this header is the array size and a pointer to the `rc_adjust_x` function for the array element type.

Objects allocated in the `pointerfree` allocator do not have this header, except if they are arrays allocated with `rarrayextend` (or `typed_arrayextend`).

The `parent`, `sibling` and `children` fields store the region hierarchy: the `parent` pointer points from a child to its parent region; the parent region points to its first child with the `children` field; subsequent children can be found by following the `sibling` field through each child. The last child has a `NULL` `sibling` field.

The `id` and `nextid` fields are a depth-first numbering of the region hierarchy, which allows efficient implementation of runtime checks for the `parentptr` qualifier. This numbering is recomputed every time a region is created. We have not investigated more efficient approaches for computing this numbering as its overhead is not significant in our benchmarks.

The `region` structure above is stored towards the beginning of the first 8kB block of memory allocated for a region. Rather than place the `region` structure at offset 0 in this block in all regions, we place the structure for the first region at offset 0, the second at offset 64, the third at 128, etc. Without this offset, two `region` structures would most likely conflict in a processor's L1 cache (which is typically small—8kB-32kB) as all blocks are aligned on 8kB boundaries. After reaching offset 1024, we restart at offset 0.

```
struct allocator {
  struct block_allocator smallblock;
  struct block_allocator superblock;
  struct block *usedpages;
  struct block *usedblocks;
};

struct block_allocator {
  char *base, *allocfrom;
};
```

Figure 4.2: Allocator structure

4.2.2 Allocators

Allocators allocate memory to regions in *blocks* whose size is a multiple of the page size (currently 8kB[2]) and which are aligned on a page-size boundary. An allocator, whose structure is shown in Figure 4.2, allocates memory for most objects from two blocks:

- The `smallblock` is 8kB: objects up to 4kB in size (all sizes include the header size) are allocated here.
- The `superblock` is 16kB: objects up to 8kB in size are allocated here.

Objects greater than 8kB are each allocated in a separate block. This scheme guarantees that at most 50% of a block will be wasted due to internal fragmentation: for objects up to 8kB, each object is limited to 50% of the size of it's block, objects greater than 8kB will waste at most 8kB (as a block size must be a multiple of 8kB). We investigated a similar scheme which guaranteed a maximum overhead of 25%, but found that in practice it required slightly more memory because a region may need three simultaneous blocks which are only very partially used.

Allocation in the `smallblock` and `superblock` blocks is sequential: the `base` pointer points to the start of the block; the `allocfrom` pointer points to the first free byte of the block. If the object and header fit in the space remaining at `allocfrom`, `allocfrom` is incremented and a pointer to space for the object and header is returned to the region library. If there is not enough space, a new block is obtained from the page allocator and

[2]This page size need not be the same as the system's page size.

used for the allocation. Allocations that do not require a new block are constant time. However, the region library must clear the memory before returning it to the user, therefore most allocations take time linear in the size of the allocated object.

The `usedpages` pointer points to the list of 8kB blocks allocated by this allocator; the `usedblocks` block points to all blocks greater than 8kB.

The `smallblock` and `superblock` are allocated on demand. The first block allocated for a `smallblock` uses the 8kB page which was allocated to hold the region object. Beyond internal fragmentation, there is some waste of memory due to the unused portions of blocks and to external fragmentation in the page allocator. Chapter 6.6 details the space usage and overheads of our allocation scheme on our benchmarks. In summary, memory usage is similar to a good malloc/free implementation, except when all regions are small (contain significantly less than 8kB). In this last case, we pay a significant cost (nearly 4x more memory usage than malloc/free on one benchmark) for our relatively large pages and separate `normal` and `pointerfree` allocators.

4.2.3 Page Allocator

The page allocator obtains memory from the system, allocates and frees blocks of memory of sizes that are a multiple of 8kB pages, and maintains a map from pages to regions (described in detail in the next section).

The problems faced by the page allocator are essentially the same as a general purpose `malloc` and `free` implementation, except that allocations are less frequent and the minimum object size is much larger (8kB rather than 8 or 16 bytes). Furthermore, most allocations and frees are of 8kB blocks. The greatly increased allocation size makes it possible to use a relatively large header on blocks without incurring significant memory overhead.

Using the terminology of Wilson et al [62], the page allocator is a sequential best fit allocator with coalescing, and special support for allocating/freeing 8kB blocks. The free and allocated blocks of memory each start with the header of Figure 4.3. These headers are used as follows:

- The allocator maintains two doubly-linked (via the `next` and `previous` fields) lists of free blocks: `single_blocks` is a list of free 8kB blocks; `unused_blocks` is a list of arbitrary-sized free blocks.

```
struct block
{
  /* Next block in region or in free list */
  struct block *next;

  /* Doubly linked list of blocks sorted by address */
  struct block *next_address, *prev_address;

  /* number of pages in this allocation unit. */
  unsigned int pagecount : PAGECOUNTBITS;

  unsigned int free : 1;

  /* Only in free blocks not in the single_blocks list */
  struct block *previous;
};
```

Figure 4.3: Block header structure

- All blocks (free and in-use) are kept in a doubly-linked list sorted by address via the `next_address` and `prev_address` fields.
- `pagecount` is the number of pages in a block.
- `free` is 1 iff this block is in the `unused_blocks` list. Allocated blocks, and blocks in the `single_blocks` list have `free == 0` to prevent them being coalesced with adjacent free blocks.
- On a 32-bit system, this header takes 16 bytes out of every allocated block. At worst (only 8kB blocks), this is an overhead of 0.2%.

The algorithm for allocating an 8kB block is:

1. If `single_blocks` is `NULL`: allocate a number of 8kB pages approximately equal to 1/128th of current total memory usage (but always at least one page), and place these individual 8kB blocks on the `single_blocks` list.
2. Return the first block from the `single_blocks` list and remove it from `single_blocks`.

To free an 8kB block: if the pages on the `single_blocks` list accounts for less than 1/64th of current total memory usage, add the block to the start of the `single_blocks` list

(but we always allow at least two pages in `single_blocks`). Otherwise free the block like blocks greater than 8kB (see below).

To allocate a block of size $n = m \times 8kB$ with $m \geq 2$ (the following algorithm is a *sequential best fit* [62]): find the smallest block b in `unused_blocks` list whose size is greater or equal to n. If b is exactly n bytes, unlink b from the `unused_blocks` list and return it. Otherwise split b into two parts, and return the n byte part. If the `unused_blocks` list does not contain a block of size at least n, obtain an n byte page-aligned block b' of memory from the system (see below for details). Return b'.

To free a block b of size n (n greater than 8kB): if the previous and/or next blocks in the sorted-by-address blocks are free (i.e., if their `free` field is 1), coalesce b with the adjacent free blocks. Otherwise add b to the start of the `unused_blocks` free list.

Our region library can obtain memory from the system either using `mmap` or `malloc`. Using `malloc` is most portable, but has one disadvantage: we need to obtain memory aligned on page-size boundaries. As this is not guaranteed by `malloc`, we must request an extra 8kB with every allocation and align the returned pointer to an 8kB boundary. To reduce the space overhead this entails, we always allocate at least one megabyte of memory in every call to `malloc`. This limits wasted memory to less than 1%. The extra memory is added to the start of the `unused_blocks` list.

Using `mmap` has the advantage that we can simply request the amount of memory we need, but is less portable. We do not use `sbrk` as it is not portable to non-Unix machines and is likely to break the C library's `malloc` implementation (`malloc` is required in RC for correct interoperation with legacy C code).

4.2.4 Page Map

Each page belongs to one region. The page allocator maintains a *region map* from pages to regions to allow efficient implementation of the `regionof` function of Figure 3.4 and of reference counting.

On a 32-bit system, this map is simply an array indexed by *page number*, i.e., the address of a page divided by 8kB, to regions. This array has $\frac{2^{32}}{8kB} = 2^{19}$ entries and occupies 2 megabytes of virtual address space. Only the parts of this array that correspond to virtual addresses that are actually used by the program are ever touched, so the actual amount of RAM necessary for this region map is actually 4 bytes per 8kB page.

On a 64-bit system, this simple approach is not possible as, in a naïve approach, the array would take 2^{54} bytes of virtual address space. In fact, available 64-bit processors do not implement a full 64-bit address space. Instead, they constrain the top c bits of an address to be equal to the $(c+1)$th bit. For instance, in the Alpha 21264, $c = 16$ or 21 [17, p1-2]. But even with $c = 21$, the array would take 2^{33} bytes. We have considered two approaches for such systems:

- A two-level region map: the 51-bit page number is split into three sections of c, a and b bits respectively, with $c + a + b = 51$. The upper c bits of the page number can be ignored. A statically allocated 2^a element array points to 2^b element arrays of pointers to regions. The 2^b element arrays are allocated as necessary. This is the approach taken in Titanium [64] on 64-bit platforms, with $c = 15$, $a = b = 18$.

- The two-level map increases the cost of reference counting by requiring an extra memory access and a few extra arithmetic operations. These could be avoided by reserving, but not allocating, $8 \times 2^{51-c}$ bytes of virtual address space for the region map array. Only the parts of this array that are actually needed are then allocated, e.g., using the `mmap` system call. This approach requires cooperation with the page allocator and `malloc` to ensure that the reserved part of the address space is not used. We have not implemented this approach.

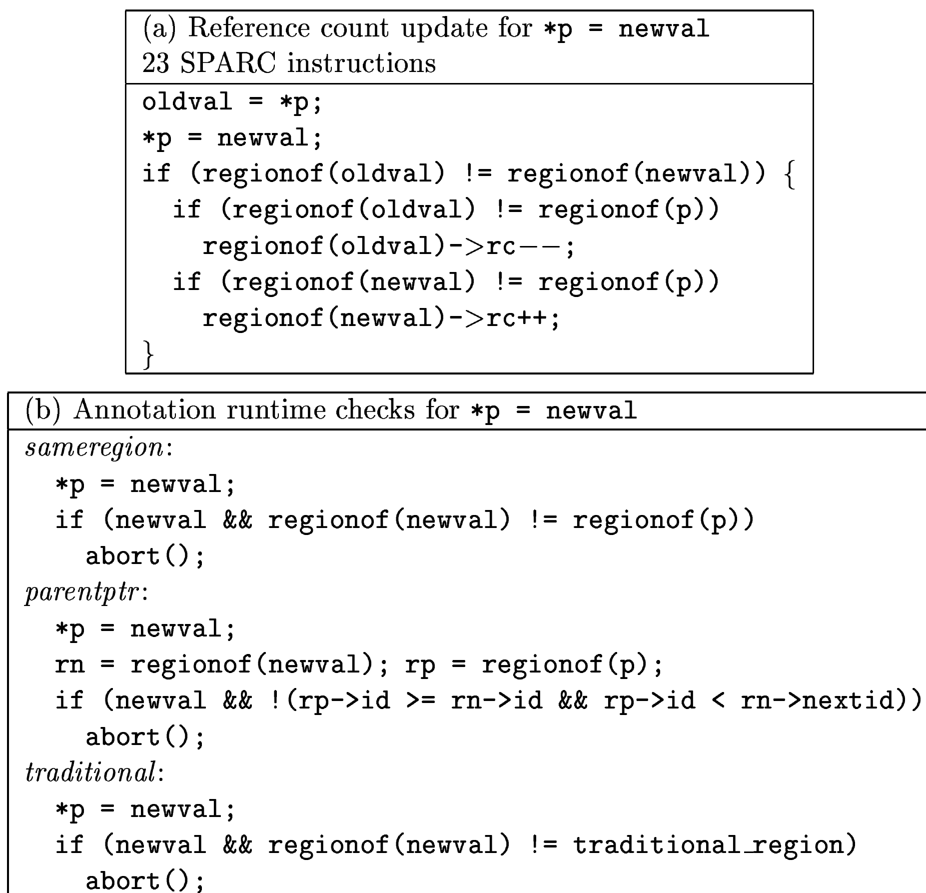
  On some machines it is possible to allocate address space without reserving virtual memory until the first access to an operating system page, e.g., on Solaris using the `MAP_NORESERVE` flag with `mmap`. This would make this second approach very easy to implement.

4.3 Reference Counting

Reference count updates may occur on any pointer assignment[3] and when a region is deleted (Chapter 4.3.1). Allocation and deallocation occur only once, but a pointer may be assigned many times. The straightforward implementation of reference count updates for pointer assignment (Figure 4.4(a)) takes 23 SPARC [58] instructions, [4] so maintaining reference counts is potentially very expensive. Most pointer assignments are updates of

[3]Copies of structured types containing pointers can be viewed as copying each field individually.

[4]On SPARC, the RC implementation keeps the page map in a global register using `gcc`'s global register variables.



(a) Reference count update for `*p = newval`
23 SPARC instructions

```
oldval = *p;
*p = newval;
if (regionof(oldval) != regionof(newval)) {
  if (regionof(oldval) != regionof(p))
    regionof(oldval)->rc--;
  if (regionof(newval) != regionof(p))
    regionof(newval)->rc++;
}
```

(b) Annotation runtime checks for `*p = newval`

sameregion:
```
  *p = newval;
  if (newval && regionof(newval) != regionof(p))
    abort();
```
parentptr:
```
  *p = newval;
  rn = regionof(newval); rp = regionof(p);
  if (newval && !(rp->id >= rn->id && rp->id < rn->nextid))
    abort();
```
traditional:
```
  *p = newval;
  if (newval && regionof(newval) != traditional_region)
    abort();
```

Figure 4.4: Reference counting and annotation checking

local variables. RC and C@ take different approaches to reducing the cost of assignments to local variables (Chapter 4.3.2). The type qualifiers of Chapter 3.2.4 also reduce the cost of reference counting: assignments to `sameregion`, `parentptr` and `traditional` pointers only need one of the runtime checks of Figure 4.4(b) rather than a reference count update. These checks take between 7 and 15 SPARC instructions and do not need to read the value being overwritten. Chapter 5.6 discusses how we eliminate a significant fraction of these runtime checks. The runtime check for `parentptr` relies on the depth-first numbering of the region hierarchy discussed in Chapter 4.2.1.

With RC's technique for handling local variables, and using the naïve reference counting operation of Figure 4.4(a) for writes to unqualified pointers, the cost of reference counting is low: 12.6% or less on our benchmarks (see Chapter 6.7.3). Chapter 4.3.5 dis-

cusses some alternate implementations of reference-counting we have investigated to further reduce this overhead.

4.3.1 Deleting Regions

As discussed above, deleting a region removes references from the now dead region to other regions by scanning all the objects in the region, using the `rc_adjust_x` functions recorded when the objects were allocated. Pseudo-code for this operation is given in Figure 4.5. This pseudo-code uses `alignof(t)` to return the alignment needed for type `t` and assumes that `double` is the C type with maximal alignment. Also `ALIGN` aligns a pointer to the specified alignment. We have found that the cost of this scan operation remains reasonable (2% or less on all benchmarks).

To delete a group of regions with `deleteregion_array`, the region library simply removes all the references from the regions being deleted as in Figure 4.5. It then checks that all the regions being deleted have a zero reference count, and aborts the program if not. Thus cycles between the regions being deleted will be removed from the region's reference counts before the check and will thus not prevent the regions from being deleted. This implementation imposes no overhead over deleting each region individually. However, if we wished to have `deleteregion_array` return a failure code when it cannot delete all the regions, we would have to rescan the deleted regions to re-increment the appropriate region reference counts. Under this model, calls to `deleteregion_array` that failed would be very slow.

The alternate approach to reference counting discussed in Chapter 4.3.7 avoids the need for the region scan and allows for an efficient implementation of `deleteregion_array` that returns a failure code, but does not extend well to large numbers of regions.

4.3.2 Reference Counting for Local Variables

If a local variable has its address taken, we treat it like a global variable, i.e., we use the pseudo-code of Figure 4.4(a) on assignments to these variables. This is necessary as local variables whose address is taken may also be updated indirectly (via the pointer value returned when the variable's address was taken). In the next two sections we will ignore these variables and will use "local variable" to mean "local variable whose address is not taken".

```
struct block *b;

for all blocks b of region r's normal allocator {
  char *deleting, *end;

  deleting = &b->previous;
  end = (char *)b + b->pagecount * 8192;
  while (1) {
    deleting = ALIGN(deleting, alignof(type_t));
    if (deleting >= end)
      break;

    type_t cln = *(type_t *)deleting;
    /* the end of unfilled blocks is marked with a NULL */
    if (!cln)
      break;

    deleting = ALIGN(deleting + sizeof(type_t), alignof(double));
    deleting += cln(deleting, -1);
  }
}
```

Figure 4.5: Region scan during `deleteregion(r)`.

The reference count of a region need only be correct when `deleteregion` is called; at all other times we need only maintain enough information to compute this reference count We use this basic observation to implement two different forms of deferred reference counting [22] in C@ (Chapter 4.3.3) and RC (Chapter 4.3.4) which allow us to greatly reduce the cost of assignments to local variables whose address is not taken. Both of these approaches have effects on `setjmp` and `longjmp` and, if used in implementing reference counting for other languages (e.g., Java), on exception handling. These effects are addressed below.

4.3.3 Local Variable Reference Counts in C@

The C@ compiler saves the runtime stack layout for the region runtime system's use. The compiler records at each function call site the set of registers and offsets in the current call frame that contain live local region pointers. Because our implementation of C@ is based on `lcc` [24], which does not have liveness information available, our prototype

considers all variables in scope to be live. The liveness information is static data whose location is recorded in the unused bits of a NOP instruction at the call site (A more complex implementation would avoid this extra instruction.)

We considered two implementations of local variable reference counting in this context: *no stack references* and *lazy stack scanning.* Our implementation of C@ uses lazy stack scanning.

No Stack References

In this approach, the reference count stored with a region excludes the references from local variables whose address is not taken. When deleting region r, the stack is scanned using the recorded stack layout information to see if there are live references to r. If so, deleting r fails.

The `setjmp` and `longjmp` C library functions can be implemented if the compiler also records the offsets in each stack frame of local region pointer variables whose address is taken. The implementation of `longjmp` must decrement the reference counts of regions pointed to by such local variables for all the frames jumped over by the `longjmp` call. Similar techniques apply for exception handling implementations. Note that if the language with exception handling does not allow the address of local variables to be taken, then the exception handling implementation can just ignore the issue of references from local variables.

Lazy Stack Scanning

With lazy stack scanning, the *actual reference count* stored with a region is the region's reference count excluding references from live variables in all active call frames below the *high water mark* on the stack (note the stack grows downward on the SPARC). The high water mark is just a location on the stack, with some frames above and some below. Our system maintains the following invariant:

($*$) *The number of frames below the high-water mark is always at least one.*

Thus writes to local variables never update reference counts.

When `deleteregion(`r`)` requires the exact reference count of r it scans the portion of the stack below the high water mark and updates the region reference counts. At that

point the actual and normal reference counts are equal. The stack scan sets the high water mark above the frame of `deleteregion`, which is not itself scanned.

Invariant ($*$) is maintained by a procedure call, but some work may be required on procedure return. If control returns to a call frame at the high-water mark, then the region reference counts attributable to local variables are decremented and the high water mark is adjusted above the call frame. This is achieved by modifying return addresses during the scan to point to a special `unscan` function that decrements the region reference counts, adjusts the high water mark above the call frame, and then jumps to the original return address.

The implementation of `setjmp` and `longjmp` is nearly the same as with the no stack references approach, except that `longjmp` has to be made aware of the high-water mark and must unscan any frames necessary to maintain the ($*$) invariant. Similar techniques apply for exception handling implementations.

4.3.4 Local Variable Reference Counts in RC

As we are compiling RC to C we cannot use C@'s approach and have `deleteregion` scan the stack for pointers to regions from local variables. Instead we place operations to increment and decrement the reference counts of regions referred to from local variables in positions that guarantee that the reference count is correct when calling `deleteregion` but that allows the count to be incorrect at other times. We investigated three placement schemes. For each scheme, we will write `incrc`(v) and `decrc`(v) for the operation that increments or decrements the reference count of the region referred to by local pointer variable v:

- *Assignment*: in functions that might call `deleteregion`, for each variable v, we add an `incrc`(v) operation when v goes dead to live on a control-flow-graph edge and a `decrc`(v) operation when v goes from live to dead on a control-flow-graph edge. This has the effect of wrapping each assignment to v in a `decrc`(v) and `incrc`(v) pair and is very similar to the usual reference counting rules.
- *Function*: before every call to a function that might call `deleteregion` we add an `incrc`(v) for every live variable v. After every such call we add a `decrc`(v) operation.
- *Optimal*: we place the `incrc`(v) and `decrc`(v) operations so as to minimise the number

of operations executed while maintaining the invariant that the reference counts of regions are correct at all calls to functions that might call `deleteregion`. The details are found below.

For all these approaches, RC needs to know which functions may delete a region. While this information is easily derived using a simple whole-program analysis, we sought to maintain separate compilation of source files in RC. Therefore RC requires that the programmer add a `deletes` keyword to each function that may delete a region. This annotation is part of the function's type.

These approaches have one drawback: it is hard to implement `setjmp` and `longjmp`. There are two problems:

- The call to `longjmp` must decrement the references from local pointer variables whose address is taken in the stack frames that are jumped over.

- If the `setjmp` targeted by a call to `longjmp` occurs in a function that may delete a region, then some live local pointer variables v in frames between the target `setjmp` and the `longjmp` call will have been included into some region's reference count with an `incrc` operation. These region reference counts must be adjusted by the `longjmp` call.

A satisfactory implementation of `setjmp` and `longjmp` appears to require compiler support in the style of C@ (Chapter 4.3.3). RC does not support `setjmp` and `longjmp`.

Minimising Local Variable Reference Counting

Our optimal scheme for reference counting local variables is based on the following observation. For each local variable v (that contains a pointer), the statements of a function f can be divided into three sets:

- S: The statements where the reference counts must take v into account. The variable v is said to be *counted*. These statements are all calls to functions that may delete a region.

- U: The statements where the reference counts must not take v into account. The variable is said to be *uncounted*. These statements are assignments to v and all points where v is dead.

- O: All other statements.

The `incrc` operation changes variable v from uncounted to counted. A `decrc` operation changes variable v from counted to uncounted.

By assigning every statement in O to either S or U, the places where `incrc` and `decrc` operations must be inserted are completely determined. To maximise performance, an assignment of statements to S and U must be chosen that minimises the total time spent in `incrc` and `decrc` operations. We show how to compute the optimal assignment under the assumption that all `incrc` and `decrc` operations take the same time, and given an execution frequency profile.

Formally, each function f has an edge-weighted control-flow graph $G = (V, E, W)$ and variables $v_1, v_2, \ldots, v_n$. The edge weights correspond to execution frequencies. The sum of the weights on all incoming edges to a node n must be equal to the sum of the outgoing weights for all nodes n except the entry and exit of f.

Each local variable v_i is considered independently. Each node n of G is assigned an initial colour:

- *white*: if v_i must be counted at n.
- *black*: if v_i must be uncounted at n.
- *grey*: all other nodes.

Minimising reference counting then becomes equivalent to assigning the colour white or black to each grey node so as to minimise the sum of weights between white and black nodes in the resulting coloured graph (an `incrc` or `decrc` operation must be inserted on every edge between black and white nodes).

To find this minimum, we first note that the solution will be unchanged if we collapse the graph $G = (V, E, W)$ into a graph $G' = (V', E', W')$ as follows:

- $V' = \{v | v \in V \wedge v \text{ is a grey node}\} \cup \{sb, sw\}$ where sb represents all black nodes, and sw represents all white nodes.
- E', W' are the obvious edges and weights obtained when merging the black (white) nodes of G into sb (sw): multiple edges between the same pairs of nodes are merged into a single edge with weights summed.

An optimal assignment of white and black nodes for G' is the same as a partition of G' into two graphs, with sb and sw separated, which minimises the weight of the cut edges. In other words, the optimal assignment is a minimum cut of G' viewed as an undirected graph and with sb, sw separated by the cut. This minimum cut can be found by finding the maximum flow [18] on G' (viewed as an undirected graph) with source sw and sink sb. The actual cut is found by disconnecting all edges saturated in the maximum flow. All nodes reachable from sb are black, all other nodes are white.

There is an $O(|V'||E'|\log(|V'|^2/|E'|))$ algorithm for maximum flow [28]. In control-flow graphs $|E| \leq 2|V|$, so for our problem the complexity is $O(|V|^2)$. A separate minimum cut problem must be solved for each local variable, so the total complexity is $O(|V|^3)$ (assuming the number of local variables is proportional to the function size).

Implementation

We use a simple static estimate (loops executed ten times, `if`'s split 50/50) to get an execution profile for the control-flow graph.

Our implementation uses a cubic time minimum cut algorithm, giving a worst case complexity of $O(|V|^4)$ where $|V|$ is function size. However the optimal placement of `incrc` and `decrc` operations is found in less than a few seconds on all but one function. This one exception takes 77s, however the subsequent compilation in `gcc` (with optimisation on) takes 182s. For this early version of RC it did not appear worthwhile to implement a faster minimum cut algorithm.

4.3.5 Alternate Reference Counting Implementations

We have considered two basic kinds of reference counting: our default choice, used in the rest of this dissertation, is to define a region r's reference count as the number of pointers to objects in r from outside r (see Chapter 3.2). We discuss variations on this standard approach in Chapter 4.3.6. A second approach is to define the *reference count between regions a and b* as the number of pointers to objects in b from objects in region a. Chapter 4.3.7 discusses the advantages and disadvantages of this approach, and several possible implementations.

The RC compiler has flags that allow the user to choose between these different approaches.

(a) Reference count update for `*p = newval` 14 SPARC instructions

`oldval = *p;`
`*p = newval;`
`regionof(oldval)->rc--;`
`regionof(newval)->rc++;`

(b) Page-based reference count update for `*p = newval` 12 SPARC instructions

`oldval = *p;`
`*p = newval;`
`pagerc(oldval)--;`
`pagerc(newval)++;`

Figure 4.6: Reference counting including same-region references.

4.3.6 Variations on Standard Reference Counting

All these approaches require that `deleteregion` perform the region scan of Figure 4.5, and the associated space overhead to store the `rc_adjust_x` functions.

We consider two independent variations on the standard reference counting scheme: *include same-region references* and *excluding parent pointers.*

Include Same-Region References

The reference count update code of Figure 4.4(a) spends a lot of effort to avoid counting references that stay within a region. A radical change is to include all references in a region's `rc` field, and to use the region scan that occurs in `deleteregion` (Figure 4.5) to compute the region's reference count. This leads to the pseudo-code for reference count update of Figure 4.6(a).

A further simplification of the reference count update code (which avoids 2 load instructions) is shown in Figure 4.6(b): a separate reference count is kept for each 8kB page of each region. A region can be safely deleted if, after the `deleteregion` region scan all pages have a 0 reference count.

Performance results for these approaches are found in Chapter 6.7.2.

Reference count update for `*p = newval`
35 SPARC instructions

```
oldval = *p;
*p = newval;
if (regionof(oldval) != regionof(newval)) {
  if (!childof(regionof(p), regionof(oldval)))
    regionof(oldval)->rc--;
  if (!childof(regionof(p), regionof(newval)))
    regionof(newval)->rc++;
}

int childof(region r, region of)
{
  return r->rid >= of->rid && r->rid < of->nextid;
}
```

Figure 4.7: Reference counting excluding parent pointers

Excluding Parent Pointers

The standard reference count definition includes pointers from r's descendants in r's reference count. At the cost of the more complex reference count sequence of Figure 4.7 we can exclude these pointers. With this approach, the check that a region and all its descendants can be deleted is simply that all these regions have a zero reference count. Chapter 6.7.2 summarises the performance of this approach.

4.3.7 Reference Counts Between Pairs of Regions

In this section, we will write $\text{RC}(a, b)$ for the reference count between regions a and b (the number of pointers to objects in region b from objects in region a). We use the term *RC-pairs* for our implementation of regions that keeps a count of references between pairs of regions rather than the count of references to each region.

Keeping reference counts between pairs of regions has the following tradeoffs:

- Deleting a region becomes much simpler and faster than in the standard approach. There is no need for the region scan of Figure 4.5. Instead, the pseudo-code for deleting region r is simply:

```
for all regions a
  if (RC(a, r) != 0) abort();
free all blocks of r
```

When a region r is created, it is also necessary to initialise $\mathrm{RC}(a, r)$ to zero for all regions a.

- Deleting a group of regions is equally simple: the same code as above is used, except that the reference count between two regions being deleted is ignored.

- Because there is no need for the region scan, there is no need to save type information with regions. This saves space for each object allocated It also allows all allocations to use the `pointerfree` allocator, which can save up to 24kB per region (one `smallblock` and `superblock` block).

- The code for reference count updates, shown in Figure 4.8(a), becomes more complicated (and hence slower). Thus the performance of reference counts between pairs of regions relative to the standard reference count is not immediately obvious: it will depend on the number of times each pointer in a region is updated. We can also follow the "include same-region references" approach, this gives the reference count update code of Figure 4.8(b). In this last case, the check for deleting region r should ignore the value of $\mathrm{RC}(r, r)$. The exact cost of these reference count updates depends on the representation of $\mathrm{RC}(a, b)$, and is discussed further below.

- The biggest problem with reference counts between pairs of regions is that the obvious implementations of $\mathrm{RC}(a, b)$ is some form of array indexed by a unique identifier for a and b. This array will need $4n^2$ bytes to store the reference counts for n regions, rather than the $4n$ bytes of the standard approach. While this is not an issue for small numbers of regions (up to a few hundred) it does make it impossible to have more than a few thousand simultaneous regions.

  A sparse array representation for $\mathrm{RC}(a, b)$ is possible and would save a lot of space in programs with many regions. In such programs, each region is unlikely to have pointers to more than a small fraction of the other regions. However, the reference count update code will then become much bigger and slower. We did not investigate this approach as we did not think it would be useful. Also, all our benchmarks run with few simultaneous regions (less than 32).

(a) Reference count update for `*p = newval`

```
oldval = *p;
*p = newval;
if (regionof(oldval) != regionof(newval)) {
  if (regionof(oldval) != regionof(p))
    RC(regionof(p), regionof(oldval))--;
  if (regionof(newval) != regionof(p))
    RC(regionof(p), regionof(newval))++;
}
```

(b) Reference count update for `*p = newval`

```
oldval = *p;
*p = newval;
RC(regionof(p), regionof(oldval))--;
RC(regionof(p), regionof(newval))++;
```

Figure 4.8: Reference counting for reference counts between pairs of regions

We considered, but did not implement, a version of reference counts between pairs of regions that degenerates into the standard approach when more than some number M of regions exist simultaneously: regions created while there are M or more simultaneously existing regions are called *extra* regions. The reference count between pairs of regions $\mathrm{RC}(a, b)$ does not exist if either a or b is an extra region. Instead there is a reference count $\mathrm{RCextra}(r)$ that counts references to region r (extra or normal) from extra regions. Extra regions need the same type information as in the standard approach. A region r cannot be deleted if $\mathrm{RCextra}(r)$ is not zero.

The RC compiler gives the user a choice between two different implementations of $\mathrm{RC}(a, b)$. Both restrict the user to a maximum number of regions M (currently 32, but easily changed). Both approaches identify a region r by a number between 0 and $M - 1$. This identifier, written r`->id` below is different from the `rid` field of a region's depth-first numbering. The *flat array* approach is to use a flat array indexed by a`->id` and b`->id`. The *split array* approach gives each region a field `rc` which is an array of M reference counts and stores $\mathrm{RC}(a, b)$ as b`->rc[`a`->id]`. The reference count update costs for Figure 4.8(a) are respectively 27 and 33 SPARC instructions for the first and second approach, for Figure 4.8(b) they are respectively 18 and 24 SPARC instructions. Chapter 6.6 shows that reference counts between pairs of regions uses somewhat less memory than the

standard approach on our benchmarks (which have at most 25 simultaneously-live regions) and that performance is similar (Chapter 6.7.2).

Both of these approaches can be extended to allow the limit M on the number of regions to be increased at runtime, with slightly higher runtime cost. The array can be resized when the number of regions increases beyond M. This requires an extra memory read to find the current value of M in each reference count update. Alternately, a register can be dedicated to holding the value of M. In the second approach, the `rc` field becomes a pointer to an array of M reference counts. These arrays can then to be reallocated when M increases. This adds an extra memory read to obtain the value of `rc` on each reference count update.

4.4 Parallelism

While RC's implementation does not support threads for reasons that are discussed below, reference-counted regions can be implemented in a parallel language without undue difficulty. This section starts by giving a general overview of parallel reference-counted region implementation techniques. We then discuss how three different kinds of regions can be created and deleted in a parallel setting: *private regions*, *shared regions with global creation and deletion* and *shared regions with independent creation and deletion*. Titanium (Chapter 3.3) has private regions and shared regions with global creation and deletion.

In the rest of this section, we will assume a parallel language with a shared-memory programming model. We will use the word *thread* to represent a thread of control in the parallel program. With this terminology, Titanium's processes are threads. Note that such a language can be implemented on machine's without shared memory (e.g., some implementations of Titanium run on uniprocessor machines connected by a network). In the discussion below, we will assume that the parallel program runs on a number of *local memories*. In the network example, each thread has a separate local memory. The opposite extreme is a single local memory for all threads, as when a parallel program runs on a single SMP machine. When two or more threads use the same local memory we say that the memory is *shared*. A thread can only perform hardware reads or writes to its local memory, access to another local memory m is mediated by one of m's thread's.

4.4.1 Parallel Region Implementation

We address the two main issues of region implementation: memory allocation and reference counting.

Memory Allocation

The region implementation of Chapter 4.2 can handle allocation by several threads in a near-independent fashion by giving each thread separate allocators. Each local memory has a single page allocator which is responsible for allocating and freeing the blocks for each thread's allocators. When more than one thread shares a local memory this page allocator will need to use locking to avoid corrupting its local data structures, but most object allocations do not lead to calls to the page allocator so the locking overhead is low.

Reference Counting

If we follow the reference count update techniques of Chapter 4.3, reference count updates must be protected by a lock/unlock operation. We believe that this approach is too expensive.

Instead, each thread has a separate reference count for each region. The region's reference count is the sum of all these thread reference counts. Note that the reference count in a single thread may be negative if that thread destroys references created by another thread. We saw above that each thread also has its own allocators. Thus it makes sense to give each thread t a separate region object with separate reference counts and allocators. We call this region object the *representative* of the region for t. All these region objects are tied together by giving them the same unique identifier, distinct from that of other regions. If the parallel language has subregions, we can reuse the `id` or `nextid` fields of the `region` structure for this purpose (Chapter 4.2.1).

The `regionof` function is then changed to return the representative of the region in the current thread, rather than the thread itself, and the reference count code of Figure 4.4(a) can be used nearly unchanged. But there is still a problem when several threads share a local memory: consider the following code:

```
Thread 1                    Thread 2
q = new T();
o->ptr = q;
o->ptr = NULL;              o->ptr = NULL;
```

(a) Reference count update for `*p = newval`

```
oldval = newval;
swap (*p, oldval);
if (regionof(oldval) != regionof(newval)) {
  if (regionof(oldval) != regionof(p))
    regionof(oldval)->rc--;
  if (regionof(newval) != regionof(p))
    regionof(newval)->rc++;
}
```

Figure 4.9: Reference counting in a parallel language

and assume that `o` has the same value in both threads, and that the two assignments to `NULL` are executed at nearly the same time. It is then possible for both thread 1 and 2 to get `oldval == q` in the reference count update code of Figure 4.4(a). Then the reference count of the region of `q` will be decremented twice, rather than once. To avoid this problem we have to use an atomic swap operation to update `*p`. The new reference count code is shown in Figure 4.9.

The atomic swap is more expensive than a load and store. Chapter 6.11 gives the cost of replacing pointer updates by atomic swaps on our collection of benchmarks.

4.4.2 Creating and Deleting Regions

We consider the implementation of three different kinds of regions. A *private region* r is created by a thread t and only allows t to allocate objects in r, and does not allow reads or writes of objects in r by threads other than t. A *shared region with global creation and deletion* does not have any access or allocation restrictions, but all threads must cooperate in creation and deletion of the region. Finally, a *shared regions with independent creation and deletion* has no restrictions: such a region can be created or deleted by any thread at any time.

Private Regions

A private region r and its contents are not accessible to threads other than its creator t. Therefore r has a single reference count, and creation and deletion of r can be handled in the same way as the non-parallel case.

If the language with parallel threads also supports some form of shared region (or other ways of sharing memory), then it is necessary to enforce the restriction that private regions and their objects are not accessible to other threads. In Titanium, this is enforced by a runtime check that no pointer to an object of a private region is stored in an object of a shared region. An alternate approach is to use static type checking that distinguishes private and shared objects and prevents access to private objects from other threads. See Liblit's work [38] for details on one such type system.

Shared Region with Global Creation and Deletion

For this type of region, the code executed by each thread must include an explicit call to an operation to create or delete the region (see Chapter 3.3.4 for an example of this style in Titanium).

It is thus easy for all threads to agree on a unique identifier for the region when it is created and for each thread to immediately create its representative for the region. When a region is deleted, all the regions can perform a parallel sum to find the reference count of the region. If this sum is non-zero, the region deletion fails.

As the points where a region is deleted are explicitly visible in the source code it is easy to implement either of the deferred reference counting techniques discussed in Chapter 4.3.2.

Shared Regions with Independent Creation and Deletion

These regions are essentially unrestricted: at any time, a thread can create a region. Later, the same or another thread can decide to delete this region without any explicit operations in the code of any other thread. Region creation remains straightforward in this model: the thread creating the region creates the representative of the region for all other threads, and ensure that the region gets a unique identifier.

When a thread t wants to delete region r it must stop the other threads as they may be updating their reference count for r. The technique used to stop the other threads also interacts with the implementation of deferred reference counting for local variables (Chapter 4.3.2). The two techniques we discuss here for stopping other threads are also used by parallel garbage collectors:

- The other threads can be stopped using operating system facilities. Then a thread t'

might be in the middle of a reference count update that concerns region r. If the compiler records stack layout information to allow deferred reference counting of local variables in the style of C@ (Chapter 4.3.3), then this information can easily be extended to preserve safety of reference counting in the presence of threads: the local variable `newval` must be considered live until the end of the `regionof(newval)->rc++` statement of Figure 4.4(a). Also, information on local variables must be available for every instruction rather than just at function call sites. Stichnoth et al [48] report a space overhead of 20% for a Java compiler which records such information.

This technique is not compatible with deferred reference counting local variables in the style of RC (Chapter 4.3.4) as we cannot know what code another thread is currently executing.

- The program's generated code can include explicit *safe points*, i.e., points at which it is safe to stop a thread. These points would not be placed inside a reference count update. At each safe point, a thread checks to see if another thread has requested that it be stopped. This has the advantage that the information on local variables takes less space. Tarditi [54] reports a space overhead of 3.6% for such a scheme, but does not give the cost in execution time of the safe point checks. However, in his dissertation [53, p52] Tarditi did report a cost of 4-6% for the safe point checks used in the SML/NJ compiler [2].

  With this approach, both RC or C@'s deferred reference counting techniques can be used. We did not implement thread support in RC because of the increased complexity, and because we have no region-based multithreaded C benchmarks.

4.5 Real-Time Regions

We believe that reference-counted regions are suitable for use in a safe, real-time language, though RC does not attempt to be real-time. All time costs incurred with reference-counted regions are predictable, assuming an underlying unsafe real-time region library (e.g., an implementation similar to the `LTMemory` class of Real-Time Java [14]):

- Region creation: an extra constant overhead over the underlying region library is needed to initialise the region's reference count to zero.

- Object allocation: objects must be cleared, which takes time linear in the allocated object size.

- Region destruction: in addition to the underlying unsafe region time, we must check the reference count (constant time) and scan the deleted region to remove references to other regions. This last operation takes time linear in the size of the region, and is therefore predictable.

- Reference-count updates (local variables): the runtime costs of both the *Function* and *Assignment* schemes of Chapter 4.3.2 for reference-counting local variables are predictable. For *Assignment*, all assignments to local variables in functions that might delete a region incur a constant overhead. For *Function*, all calls to functions that might delete a region incur an overhead proportional to the number of live local pointer variables. In both cases, an explicit `deletes` qualifier makes reasoning easier by making it obvious which functions might delete regions.

  The *Optimal* scheme is inappropriate for real-time use as it's results are not obvious without running the min-cut algorithm, and may change unpredictably following small code changes.

- Reference-count updates (all other writes): the reference-count updates for other writes add a constant cost to pointer writes.

- Qualifier runtime checks: these also add a constant cost to pointer writes. While the qualifier-runtime-check-elimination system of Chapter 5.6 could be used in a real-time context, it should probably not be relied on as small changes to an application can cause checks to be eliminated or retained. This system could be used in a setting where compile-time errors are reported when runtime checks cannot be eliminated.

Chapter 5

rlang

We have designed *rlang*, a simple C-like language with a region type system to formalise the concepts behind RC's annotations. These annotations can then viewed as a simple way for the user to specify the more complex rlang types.

We first present rlang's type system (Chapter 5.1), then rlang and its type checking rules (Chapter 5.2). Next we give a simple semantics for rlang (Chapter 5.3) and use it to show the soundness of our type checking rules (Chapters 5.4 and 5.5).

We show how to build an analysis, based on translating RC programs to rlang, that statically verifies the correctness of some assignments to annotated RC types and thus reduces the runtime cost of safety in RC programs (Chapter 5.6). We end with a discussion of alternate translations of RC to rlang, e.g., to incorporate the language extensions of Chapter 3.4.

5.1 rlang Types

We first define a simple model for the heap of a region-based language. The heap H is divided into regions, each containing a number of objects. Objects are named structures with named fields containing pointers. Pointers can be `null`, point to objects, or to regions. We write $A_H = \{\top, r_1, \ldots, r_n\}$ for the set of regions of H. We define a partial order on A_H: $r' \preceq r$ if r' is a subregion of r. The region of an object pointer is the region of the targeted object. The region of a pointer v is $\top$ iff $v = \texttt{null}$. We define $r \preceq \top$ for all regions r.

The region type system of rlang (Figure 5.1) reflects this heap structure and explicitly specifies the region to which every pointer points with a *region expression* σ $(\ldots @\sigma)$.

$$
\begin{array}{ll}
\tau = \mu@\sigma \mid \exists\rho/\delta.\tau & \text{(types)} \\
\mu = \texttt{region} \mid T[\sigma_1, \ldots, \sigma_m] & \text{(base types)} \\
\sigma = \rho \mid R \mid \top & \text{(region expressions)} \\
\delta = \sigma \preceq \sigma | \neg\delta | \delta \vee \delta | (\delta) & \text{(region properties)} \\
\texttt{struct}\ T[\rho_1, \ldots, \rho_m]\{1 : \tau_1, \ldots, n : \tau_n\} & \text{(structure declarations)}
\end{array}
$$

T: type names, ρ: abstract regions, R: region constants

Figure 5.1: Region type language

To keep rlang simple, we only include types for pointers: pointers to regions (`region`), and pointers to named records with named fields. Function and non-pointer types could be added easily to both the heap model and type language.

Region expressions are either *abstract regions* ρ or elements of the set $C_R = R \cup \{\top\}$ of *region constants*. Region constants denote regions that always exist and cannot be deleted, such as the "traditional region". Abstract regions denote any region in A_H. Abstract regions are introduced existentially with the $\exists\rho/\delta.\tau$ construct, which means that ρ is a region in A_H that respects the property specified by boolean expression δ. For instance, the type $\exists\rho/\top \preceq \top.T[\ldots]@\rho$ represents an object of type T in any region (as the boolean expression is always true). To simplify notation, we write `true` as shorthand for $\top \preceq \top$ and $\exists\rho$ as a shorthand for $\exists\rho/\texttt{true}$. Structure definitions are parameterised over a set $\rho_1, \ldots, \rho_m$ of abstract regions; structure uses instantiate structure declarations with a set of region expressions. Function declarations also introduce abstract regions (see Chapter 5.2).

If two values point to the same abstract region ρ then the values must specify objects in the same region. As a consequence, if one of the values is null then $\rho = \top$ so the other value is null too. Existentially quantified regions must be used if two values can be null independently of each other, but point to the same region if non-null. For instance, in

$$
\begin{array}{l}
\texttt{struct}\ L[\rho]\ \{ \\
\quad v : \exists\rho'.\texttt{region}@\rho', \\
\quad next : \exists\rho''/\rho'' = \top \vee \rho'' = \rho.L[\rho'']@\rho'' \\
\} \\
x : L[\rho]@\rho
\end{array}
$$

x is a list stored in region ρ of arbitrary regions. Without the existentially quantified type

$$
\begin{array}{lll}
\text{program} ::= \text{fn}^* & & \\
\text{fn} & ::= & f[\rho_1, \ldots, \rho_m]/\delta(x_1 : \tau_1, \ldots, x_n : \tau_n) : \\
 & & \tau, \delta' \\
 & & \texttt{is } [\rho'_1, \ldots, \rho'_p] x'_1 : \tau'_1, \ldots, x'_q : \tau'_q, s, x \\
s & ::= & s_1; s_2 \\
 & | & \texttt{if } x\ s_1\ s_2 \\
 & | & \texttt{while } x\ s \\
 & | & x_0 = x_1 \\
 & | & x_0 = f[\sigma_1, \ldots, \sigma_m](x_1, \ldots, x_n) \\
 & | & x_0 = x_1.field \\
 & | & x_1.field = x_2 \\
 & | & x_0 = \texttt{null} \\
 & | & x_0 = \texttt{new } T[\sigma_1, \ldots, \sigma_m](x_1, \ldots, x_n)@x' \\
 & | & \texttt{chk } \delta
\end{array}
$$

Some predefined functions:

$\texttt{newregion}[]/\texttt{true}() : \exists\rho.\texttt{region}@\rho, \texttt{true}$

$\texttt{newsubregion}[\rho]/\texttt{true}() : \exists\rho'/\rho' \preceq \rho.\texttt{region}@\rho', \texttt{true}$

$\texttt{deleteregion}[\rho]/\texttt{true}(r : \texttt{region}@\rho) : \texttt{region}@\top, \texttt{true}$

$\texttt{regionof_}T[\rho, \rho_1, \ldots]/\texttt{true}(x : T[\rho_1, \ldots]@\rho) : \texttt{region}@\rho, \texttt{true}$

Figure 5.2: *rlang*, a simple imperative language with regions

the *next* field could not be null as it would be in the same region as its parent (which is obviously not null if *next* exists).

5.2 Region Type Checking in rlang

We chose to define *rlang* (Figure 5.2) as an imperative language both because this is closer to C and because the properties of abstract regions are flow-sensitive: they change as a result of function calls, field accesses and runtime checks and so may be different at every program point.

Functions f have arguments $x_1, \ldots, x_n$, local variables $x'_1, \ldots, x'_q$, body s and are parameterised over abstract regions $\rho_1, \ldots, \rho_m$. The result of f is found in variable x after s has executed. The set of abstract regions valid in the argument and result types of f is $\{\rho_1, \ldots, \rho_m\}$. The set of abstract regions valid in the types of local variables of f is $\{\rho_1, \ldots, \rho_m, \rho'_1, \ldots, \rho'_p\}$. The local variables $x'_1, \ldots, x'_q$ must be dead before s. Functions have an input property δ that expresses requirements that must hold between the abstract

$$\frac{\delta, L_s \vdash s, \delta' \qquad x : \tau \qquad \delta' \Rightarrow \delta'' \qquad \mathrm{fv}(\delta) \cup \mathrm{fv}(\delta'') \subseteq \{\rho_1, \ldots, \rho_m\} \qquad x'_1, \ldots, x'_q \text{ are dead before } s}{\vdash f[\rho_1, \ldots, \rho_m]/\delta(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau, \delta'' \ \mathtt{is}\ [\rho'_1, \ldots, \rho'_p] x'_1 : \tau'_1, \ldots, x'_q : \tau'_q, s, x} \ (\text{fndef})$$

$$\frac{x_0 : \tau_0 \qquad x_1 : \tau_1 \qquad \delta, L \vdash \tau_0 \leftarrow \tau_1, \delta', L'}{\delta, L \vdash x_0 = x_1, \delta'} \ (\text{assign})$$

$$\frac{x_0 : \tau_0 \qquad x_1 : \mu_1 @ \sigma_1 \qquad x_1.\mathit{field} : \tau'_1 \qquad \delta \wedge \sigma_1 \neq \top, L \vdash \tau_0 \leftarrow \tau'_1, \delta', L'}{\delta, L \vdash x_0 = x_1.\mathit{field}, \delta'} \ (\text{read})$$

$$\frac{x_1 : \mu_1 @ \sigma_1 \qquad x_1.\mathit{field} : \tau'_1 \qquad x_2 : \tau_2 \qquad \delta \wedge \sigma_1 \neq \top, L \vdash \tau'_1 \leftarrow \tau_2, \delta', L'}{\delta, L \vdash x_1.\mathit{field} = x_2, \delta'} \ (\text{write})$$

$$\frac{\begin{array}{c}\text{struct } T[\rho_1, \ldots, \rho_m]\{\mathit{field}_1 : \tau'_1, \ldots, \mathit{field}_n : \tau'_n\} \\ x_i : \tau_i \qquad \delta_i, L_i \vdash \tau'_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m] \leftarrow \tau_i, \delta_{i+1}, L_{i+1} \\ x_0 : \tau_0 \qquad x' : \mathtt{region} @ \sigma' \qquad \delta_{n+1}, L_{n+1} \vdash \tau_0 \leftarrow T[\sigma_1, \ldots, \sigma_m] @ \sigma', \delta', L'\end{array}}{\delta_1, L_1 \vdash x_0 = \mathtt{new}\ T[\sigma_1, \ldots, \sigma_m](x_1, \ldots, x_n) @ x', \delta'} \ (\text{new})$$

$$\frac{x_0 : \mu_0 @ \sigma_0 \qquad \delta, L \vdash \mu_0 @ \sigma_0 \leftarrow \mu_0 @ \top, \delta', L'}{\delta, L \vdash x_0 = \mathtt{null}, \delta'} \ (\text{null}) \qquad \frac{\mathrm{fv}(\delta') \subseteq L}{\delta, L \vdash \mathtt{chk}\ \delta', \delta \wedge \delta'} \ (\text{check})$$

$$\frac{\delta, L \vdash s_1, \delta' \qquad \delta', L_{s_2} \vdash s_2, \delta''}{\delta, L \vdash s_1; s_2, \delta''} \qquad \frac{\delta, L_{s_1} \vdash s_1, \delta' \qquad \delta, L_{s_2} \vdash s_2, \delta''}{\delta, L \vdash \mathtt{if}\ x\ s_1\ s_2, \delta' \vee \delta''}$$

$$\frac{\delta \vee \delta'', L_s \vdash s, \delta''}{\delta, L \vdash \mathtt{while}\ x\ s, \delta \vee \delta''}$$

$$\frac{\begin{array}{c} f[\rho_1, \ldots, \rho_m]/\delta'(y_1 : \tau'_1, \ldots y_n : \tau'_n) : \tau', \delta'' \qquad x_i : \tau_i \\ \delta_i, L_i \vdash \tau'_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m] \leftarrow \tau_i, \delta_{i+1}, L_{i+1} \qquad \delta_{n+1} \Rightarrow \delta'[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m] \\ \delta_{n+1} \wedge \delta''[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m], L_{n+1} \vdash \tau_0 \leftarrow \tau'[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m], \delta''', L' \end{array}}{\delta_1, L_1 \vdash x_0 = f[\sigma_1, \ldots, \sigma_m](x_1, \ldots, x_n), \delta'''} \ (\text{fncall})$$

Assignment

$$\frac{\sigma \in L \cup C_R \qquad \mathrm{fv}(\delta'[\sigma/\rho]) \subseteq L \qquad \delta \Rightarrow \delta'[\sigma/\rho] \qquad \delta, L \vdash \tau[\sigma/\rho] \leftarrow \tau', \delta'', L'}{\delta, L \vdash \exists \rho/\delta'.\tau \leftarrow \tau', \delta'', L'} \ (\exists\text{gen.})$$

$$\frac{\rho \notin L \qquad \delta \Rightarrow \delta'' \qquad \mathrm{fv}(\delta'') \subseteq L \qquad \delta'' \wedge \delta'[\rho/\rho'], L \cup \{\rho\} \vdash \tau \leftarrow \tau'[\rho/\rho'], \delta''', L'}{\delta, L \vdash \tau \leftarrow \exists \rho'/\delta'.\tau', \delta''', L'} \ (\exists\text{inst.})$$

$$\frac{\delta, L \vdash \sigma \leftarrow \sigma', \delta', L'}{\delta, L \vdash \mathtt{region} @ \sigma \leftarrow \mathtt{region} @ \sigma', \delta', L'}$$

$$\frac{\delta, L \vdash \sigma \leftarrow \sigma', \delta_1, L_1 \qquad \delta_i, L_i \vdash \sigma_i \leftarrow \sigma'_i, \delta_{i+1}, L_{i+1}}{\delta, L \vdash T[\sigma_1, \ldots, \sigma_m] @ \sigma \leftarrow T[\sigma'_1, \ldots, \sigma'_m] @ \sigma', \delta_{m+1}, L_{m+1}}$$

$$\frac{\sigma \in L \cup C_R \qquad \delta \Rightarrow \sigma = \sigma'}{\delta, L \vdash \sigma \leftarrow \sigma', \delta, L} \ (\text{equal}) \qquad \frac{\rho \notin L \qquad \delta \Rightarrow \delta' \qquad \mathrm{fv}(\delta') \subseteq L}{\delta, L \vdash \rho \leftarrow \sigma', \delta' \wedge \rho = \sigma', L \cup \{\rho\}} \ (\text{bind})$$

Figure 5.3: Region Type Checking

region parameters at all calls to f. The output property δ' expresses properties that are known to hold between the abstract region parameters when f returns.

The `chk` δ statement is a runtime check that the property specified by δ holds. If the check fails, the program is aborted. Instantiation and generalisation of existential types is implicit in the rules for assignment (Figure 5.3) rather than being done by explicit instantiate and generalise operations. The rest of the language is straightforward: `if` and `while` statements assume `null` is false and everything else is true; `new` statements specify values for the structure's fields; the program is executed by calling a function called `main` with no arguments. Figure 5.2 also gives signatures for the predefined `newregion`, `newsubregion`, `deleteregion` and `regionof_`T (one for each structure type T) functions.

We write $X[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m]$ for substitution of region expressions for (free) abstract regions in region expressions, boolean expressions and types. The notation $x : \tau$ and $x.field : \tau$ asserts that x, or a field of x, has type τ. The set of free abstract regions of a boolean expression δ is $\mathrm{fv}(\delta)$.

Type checking for rlang (Figure 5.3) relies extensively on boolean expressions specifying properties of abstract regions. Statements of a function f are checked by the judgment $\delta, L_s \vdash s, \delta'$. The input property δ describes the properties of f's abstract regions before executing s, the output property δ' the properties of these abstract regions after executing s. The set L_s contains f's abstract region parameters and the abstract regions used in any live variable's type; L_s is used while typechecking assignments. We assume that L_s is precomputed for each statement s using a standard liveness analysis.

Rather than have constructs for binding of abstract regions and instantiation and generalisation of existential types, rlang allows these operations to be performed implicitly during assignment. The judgments $\delta, L \vdash \tau_1 \leftarrow \tau_2, \delta', L'$ of Figure 5.3 check that a value of type τ_2 is assignable to a location of type τ_1. These judgments take an input property δ and live abstract region set L and produce an updated (as a result of binding abstract regions) output property δ' and live abstract region set L'.

Assignment can bind abstract regions (the (bind) rule). For instance,

$$x : \texttt{region@}\rho_1$$
$$y : \texttt{region@}\rho_2$$
$$x = y$$

sets x to the value of y and binds ρ_1 to the same region as ρ_2. We require that the bound

region ρ not be a member of L. Rebinding an abstract region used in a live variable would be unsound. We also forbid rebinding the abstract regions used in a function's parameters: if these could be rebound, then the output property δ' of a function f (see Figure 5.2) would not describe the properties of the function's input region parameters, instead it would describe properties of whatever regions the abstract regions were rebound to. It is possible that the input property δ of an assignment described properties of the old value of ρ, these properties are removed by using a new property δ', implied by δ, that only has elements of L amongst its free variables.

Instantiation of an existential type is also implicit (the ($\exists$inst. rule). The assignment $x = y$ with $x : \texttt{region@}\rho_1$ and $y : \exists\rho/\rho \preceq \rho_2.\texttt{region@}\rho$ sets x to the value of y and binds ρ_1 to a region that is less than or equal to ρ_2. As with (bind), we require that the newly bound abstract region ρ not be in L and build a new input property δ'', implied by δ, that only has elements of L amongst its free variables. We then add to δ'' the properties from the instantiated existential type.

Generalisation of existential types is also possible in an assignment statement (the ($\exists$gen. rule). The assignment $x = y$ with $x : \exists\rho/\rho \preceq \rho_2.\texttt{region@}\rho$, $y : \texttt{region@}\rho_1$ and input property $\rho_1 \preceq \rho_2$ is valid and sets x to the value of y. This assignment is allowed as long as there is some region expression σ (in the example, $\sigma = \rho_1$) which satisfies the existential type's bound, and that $\tau[\sigma/\rho]$ is assignable from τ'.

The rest of the rules for assignment are traditional: base types are assignable if their region expressions match or if the target region expression can be bound to the source one using the (bind) rule. Two region expressions match if δ implies they are equal.

The rules for assigning local variables (assign), reading a field (read) or writing a field (write) check that the source is assignable to the target. Additionally, reading or writing a field of x guarantees that x is not `null`, hence that x's region is not $\top$. Object creation (new) is essentially a sequence of assignments from the field values to the fields of the newly created object, and of the newly created object to the `new` statement's target. Initialisation to `null` (null) requires only that the target variable's region be $\top$. After execution of a runtime check, the checked relation holds (check).

The rules for statement sequencing, `if` and `while` statements are standard for a forward data-flow problem. Function definition (fndef) is straightforward: the result variable's type must match the function declaration and the function's output property must be implied by the function body's output property. All local variables of the function

must be dead as they have not been initialised.[1]

The most complicated rule is a call to a function f (fncall). All references to elements of f's signature must substitute the actual region expressions at a call for f's formal region parameters. The second line checks that the call's arguments are assignable to f's parameters and that the properties at the call site imply f's input property. After the call, f's output property holds for the actual region expressions and f's result must be assignable to the call's destination.

5.3 Semantics

Our semantics concentrates on the regions of variables and objects and ignores the other aspects of the types to simplify our presentation. We assume, in both the semantics and soundness proof, that a non-null pointer of type `region` points to a region, and that a non-null pointer of type $T[\sigma_1, \ldots, \sigma_m]@\sigma$ points to some object of type T. Our semantics does represent the concrete regions corresponding to the abstract regions, both for local variables and for heap-allocated objects.

We first define a representation for heaps, values and regions:

- A *value* (or *pointer*) is represented as a unique natural integer. `null` pointers are represented by 0.

- A *region* is represented as a unique natural integer. The $\top$ region is represented by 0. We assume our partial order on regions ($\preceq$) is defined on these integers.

- Given a type `struct` $T[\rho_1, \ldots, \rho_m]\{f_1 : \tau_1, \ldots, f_n : \tau_n\}$, an *object o of type T* is represented as a pair (R, P) containing a tuple of regions $R = (r_0, r_1, \ldots, r_m)$ and a tuple of values $P = (v_1, \ldots, v_m)$. The region of o is r_0, r_i is the value of ρ_i and v_i is the value of f_i. As the $\top$ region contains no object $r_0 \neq 0$. The object representing region r is the pair $((r), ())$. Note that the $\top$ region is represented by object $((0), ())$.

- A *heap* H is a partial map from N to objects, with $0 \notin \text{dom}(H)$. Formally, $H : \mathrm{N} \hookrightarrow (\bigcup_{i=1}^{\infty} \mathrm{N}^i) \times (\bigcup_{i=0}^{\infty} \mathrm{N}^i)$. We assume that the set A_H of regions of H is available. For

[1] Initialising local variables to `null` at entry would not be correct as this would also imply that some abstract regions were $\perp$, e.g., for the local variable x : `region`$@\rho$. If ρ was an abstract region parameter of f this would be unsound.

$$\frac{x_0 : \tau_0 \qquad x_1 : \tau_1 \qquad < H, R, E(x_1), \tau_0, \tau_1 > \rightsquigarrow R'}{< H, E, R, x_0 = x_1 > \rightsquigarrow < H, E[x_0 = E(x_1)], R' >} \text{(s_assign)}$$

$$\frac{\begin{array}{c} x_0 : \tau_0 \qquad x_1 : T[\sigma_1, \ldots, \sigma_m]@\sigma \qquad \texttt{struct}\ T[\rho_1, \ldots, \rho_m]\{\ldots, f_i : \tau_i, \ldots\} \\ H(E(x_1)) = (_, (x_1, \ldots, x_n)) \qquad < H, R, x_i, \tau_0, \tau_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m] > \rightsquigarrow R' \end{array}}{< H, E, R, x_0 = x_1.f_i > \rightsquigarrow < H, E[x_0 = x_i], R' >} \text{(s_read)}$$

$$\frac{\begin{array}{c} x_1 : T[\sigma_1, \ldots, \sigma_m]@\sigma \qquad \texttt{struct}\ T[\rho_1, \ldots, \rho_m]\{\ldots, f_i : \tau_i, \ldots\} \qquad x_2 : \tau_2 \\ H(E(x_1)) = ((r_0, \ldots, r_m), (x_1, \ldots, x_n)) \\ < H, R, E(x_2), \tau_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m], \tau_2 > \rightsquigarrow R' \\ o = ((r_0, \ldots, r_m), (x_1, \ldots, x_{i-1}, E(x_2), x_{i+1}, \ldots, x_n)) \end{array}}{< H, E, R, x_1.f_i = x_2 > \rightsquigarrow < H[E(x_1) = o], E, R' >} \text{(s_write)}$$

$$\frac{\begin{array}{c} x_0 : \tau_0 \qquad x_i : \tau_i \qquad x' : \texttt{region}@\sigma' \qquad \texttt{struct}\ T[\rho_1, \ldots, \rho_m]\{f_1 : \tau_1', \ldots, f_n : \tau_n'\} \\ < H, R_i, E(x_i), \tau_i'[\sigma_1/\rho_1, \ldots, \sigma_m, \rho_m], \tau_i > \rightsquigarrow R_{i+1} \qquad v \notin \mathrm{dom}(H) \wedge v \neq 0 \\ H(E(x')) = ((r), ()) \qquad r \neq 0 \qquad o = ((r, R_{n+1}\sigma_1, \ldots, R_{n+1}\sigma_m), (E(x_1), \ldots, E(x_n))) \\ < H[v = o], R_{n+1}, v, \tau_0, T[\sigma_1, \ldots, \sigma_m]@\sigma' > \rightsquigarrow R' \end{array}}{< H, E, R_1, x_0 = \texttt{new}\ T[\sigma_1, \ldots, \sigma_m](x_1, \ldots, x_n)@x' > \rightsquigarrow < H[v = o], E[x_0 = v], R' >}$$
(s_new)

$$\frac{x_0 : \mu_0@\sigma_0}{< H, E, x_0 = \texttt{null} > \rightsquigarrow < H, E[x_0 = 0], R[\sigma_0 = 0] >} \text{(s_null)}$$

$$\frac{R \models \delta}{< H, E, R, \texttt{chk}\ \delta > \rightsquigarrow < H, E, R >} \text{(s_chk)}$$

$$\frac{< H, E, R, s_1 > \rightsquigarrow < H', E', R' > \qquad < H', E', R', s_2 > \rightsquigarrow < H'', E'', R'' >}{< H, E, R, s_1; s_2 > \rightsquigarrow < H'', E'', R'' >}$$

$$\frac{E(x) \neq 0 \qquad < H, E, R, s_1 > \rightsquigarrow < H', E', R' >}{< H, E, R, \texttt{if}\ x\ s_1\ s_2 > \rightsquigarrow < H', E', R' >}$$

$$\frac{E(x) = 0 \qquad < H, E, R, s_2 > \rightsquigarrow < H', E', R' >}{< H, E, R, \texttt{if}\ x\ s_1\ s_2 > \rightsquigarrow < H', E', R' >}$$

$$\frac{\begin{array}{c} E(x) \neq 0 \qquad < H, E, R, s > \rightsquigarrow < H', E', R' > \\ < H', E', R', \texttt{while}\ x\ s > \rightsquigarrow < H'', E'', R'' > \end{array}}{< H, E, R, \texttt{while}\ x\ s > \rightsquigarrow < H'', E'', R'' >}$$

$$\frac{E(x) = 0}{< H, E, R, \texttt{while}\ x\ s > \rightsquigarrow < H, E, R >}$$

$$\frac{\begin{array}{c} f[\rho_1, \ldots, \rho_m]/\delta(y_1 : \tau_1', \ldots y_n : \tau_n') : \tau', \delta'\ \texttt{is}\ [\rho_1', \ldots, \rho_p']w_1' : \tau_1'', \ldots, w_q' : \tau_q'', s, y \\ E_f = [y_1 = E(x_1), \ldots, y_n = E(x_n), w_1' = 0, \ldots, w_q' = 0] \\ < H, E_f, R_f, s > \rightsquigarrow < H', E_f', R_f' > \\ R_f = [\rho_1 = R_{n+1}\sigma_1, \ldots, \rho_m = R_{n+1}\sigma_m, \rho_1' = 0, \ldots, \rho_p' = 0] \\ x_i : \tau_i \qquad < H, R_i, E(x_i), \tau_i'[\sigma_1/\rho_1, \ldots, \sigma_m, \rho_m], \tau_i > \rightsquigarrow R_{i+1} \\ < H', R_{n+1}, E_f'(y), \tau_0, \tau'[\sigma_1/\rho_1, \ldots, \sigma_m, \rho_m] > \rightsquigarrow R' \end{array}}{< H, E, R_1, x_0 = f[\sigma_1, \ldots, \sigma_m](x_1, \ldots, x_n) > \rightsquigarrow < H', E[x_0 = E_f'(y)], R' >} \text{(s_fncall)}$$

Figure 5.4: Semantic reduction rules

$$\frac{< H, R, v, \tau[\sigma/\rho], \tau' > \rightsquigarrow R' \qquad \sigma \text{ from } \exists\text{gen.}}{< H, R, v, \exists\rho/\delta.\tau, \tau' > \rightsquigarrow R'} \text{(a_gen)}$$

$$\frac{\begin{array}{c}\text{there exists } r \in A_H \text{ such that} \\ R[\rho' = r] \models \delta' \text{ and } v : \tau' \text{ is partially consistent with } H \text{ under } R[\rho' = r] \\ \rho \text{ from } \exists\text{inst.} \qquad < H, R[\rho = r], v, \tau, \tau'[\rho/\rho'] > \rightsquigarrow R'\end{array}}{< H, R, v, \tau, \exists\rho'/\delta'.\tau' > \rightsquigarrow R'} \text{(a_inst)}$$

$$\frac{\begin{array}{c}\text{there does not exist } r \in A_H \text{ such that} \\ R[\rho' = r] \models \delta' \text{ and } v : \tau' \text{ is partially consistent with } H \text{ under } R[\rho' = r]\end{array}}{< H, R, v, \tau, \exists\rho'/\delta'\sigma'.\tau' > \rightsquigarrow R} \text{(a_inst_unsafe)}$$

$$\frac{}{< H, R, v, \texttt{region}@\sigma, \texttt{region}@\sigma' > \rightsquigarrow R[\sigma = R\sigma']} \text{(a_region)}$$

$$\frac{R_1 = R[\sigma = R\sigma'] \qquad R_{i+1} = R_i[\sigma_i = R_i\sigma_i']}{< H, R, v, T[\sigma_1, \ldots, \sigma_m]@\sigma, T[\sigma_1', \ldots, \sigma_m']@\sigma' > \rightsquigarrow R_{m+1}} \text{(a_struct)}$$

Figure 5.5: Semantic assignment rules

simplicity of notation the source language names of the region constants are reused as names for the corresponding runtime regions, so $C_R \subseteq A_H$.

Our semantics will use two environments during evaluation:

- An environment E mapping variables to values.
- An *abstract region map* R *over abstract regions* X, $R : X \cup C_R \rightarrow A_H$ (in a heap H), mapping region expressions to regions. We assume that $R\sigma = \sigma$ for all $\sigma \in C_R$.

We say that a region property δ is valid under abstract region map R, $R \models \delta$, if:

$$\frac{R \not\models \delta}{R \models \neg\delta} \qquad \frac{R \models \delta_1 \text{ or } R \models \delta_2}{R \models \delta_1 \vee \delta_2} \qquad \frac{R\sigma_1 \preceq R\sigma_2}{R \models \sigma_1 \preceq \sigma_2}$$

The natural operational semantics (Figure 5.4) rules take the form

$$< H, E, R, s > \rightsquigarrow < H', E', R' >$$

meaning that evaluation of s with heap H, environment E and abstract region map R produces a heap H', environment E' and abstract region map R'. The rules for assignment (s_assign), field read (s_read) and write (s_write) are straightforward: they apply sub-rules for assignment (Figure 5.5, detailed below) to update the abstract region map, then modify the environment or heap as necessary. Creation of an object (s_new) is similar, but must

pick an unique value (v) for the pointer to the new object. Assignment of `null` (s_null) is a little strange as it does not update R for the abstract regions (used in μ_0) bound as a result of this assignment. These newly bound abstract regions have no meaningful value as x_0 does not point to an object after being assigned `null`.

The rule (s_chk) for `chk` statements simply check that the asserted relation is valid at runtime. The rules for statement sequencing, `if` and `while` are standard. Function calls (s_fncall) assign the function arguments to the instantiated types of the function's arguments (so as to update the abstract region map), then evaluates the function's body in a new environment (with the function's arguments) and new abstract region map (with the function's region arguments). The function's result is assigned to the result variable.

Except for the `chk` operations, an implementation of rlang does not need the abstract region map. The heap for rlang need only contain the field values and the r_0 field of the region tuple (which is necessary for implementing the `regionof` function and reference counting). Our translation from RC to rlang (Chapter 5.6) uses `chk` statements that can be verified using only these r_0 fields.

Assignment (Figure 5.3) binds abstract regions, so can update the abstract region map R. Hence assignment reduction rules (Figure 5.5) take the form

$$< H, R, v, \tau_1, \tau_2 > \leadsto R'$$

to represent assignment of a value v of type τ_2 to a location of type τ_1 with heap H and abstract region map R. These rules return an updated abstract region map R'.

Except for instantiation of existential types, these rules are simple: assignment of `region` (a_region) and structured types (a_struct) updates R so that the region expressions of the target type are equal to the corresponding region expressions of the source type. The proof of soundness will show that this is correct even if the target region expression is a live region expression. Existential generalisation (a_gen) simply substitutes the region expression σ' used when type checking this assignment to allow the rest of the assignment derivation to see the same types as the type checking derivation.

The instantiation of existential types is more complex as the reduction rules must pick a region r for the instantiated abstract region ρ. This is straightforward if $v \neq 0$ and ρ' is used in the base type of τ' or if ρ' is the region of v. In this case, r can be found in the object stored at $H(v)$. But if $v = 0$, if ρ' is used only as a bound in subsequent existential quantifiers in τ', or is not used at all in τ' then there is no value for r that can be read

directly from the heap. Rather than enumerate all the cases that must be considered, we instead pick an arbitrary r that matches the existential quantifier's bound and is consistent with τ' and $H(v)$, as specified by *partial consistency.* Partial consistency asserts that the regions stored in the heap object for v match those specified in v's type (we write $\mathrm{fv}(\tau)$ for the free abstract regions of a type τ):

Definition 5.3.1 *$v : \tau$ is partially consistent with H under R* (with $\mathrm{fv}(\tau) \subseteq \mathrm{dom}(R)$) if it is not partially inconsistent with H under R.[2] $v : \tau$ is *partially inconsistent with H under R*:

- if $v = 0$ and $\tau = \mu@\sigma$ then $R\sigma \neq 0$.
- if $\tau = \texttt{region}@\sigma$ and $H(v) = ((r), ())$ then $r \neq R\sigma$
- if $\tau = T[\sigma_1, \ldots, \sigma_m]@\sigma$, T is defined by $\texttt{struct}\ T[\rho_1, \ldots, \rho_m]\{f_1 : \tau_1, \ldots, f_n : \tau_n\}$ and $H(v) = ((r_0, r_1, \ldots, r_m), _)$. The property holds if $(r_0 \neq R\sigma) \vee (\exists j.R\sigma_j \neq r_j)$.
- if $\tau = \exists\rho/\delta.\tau'$ then for all $r \in A_H$ such that $R[\rho = r] \models \delta$, we have $v : \tau'$ partially inconsistent with H under $R[\rho = r]$

The rule for existential type assignment (a_inst) then simply assigns a value r to ρ that allows $v : \tau'$ to be partially consistent with H. When there does not exist such a region r, evaluation of the assignment can continue with the (a_inst_unsafe) rule which aborts the update of the abstract region map. We will show in the proof of soundness that the (a_inst_unsafe) rule is never used by a well-typed program. It is easy to see that an r that matches the constraints of (a_inst) can be found by simple enumeration in time proportional to $|A_H|^{(n+1)}$ where n is the number of existential quantifiers in τ'. With a little more care, such an r can be found in time at worst proportional to n.

5.4 Soundness

We express the soundness of our type system as the preservation by reductions of the consistency of typed values with the heap and of constraint sets with the abstract region map. These definitions of consistency are as follows (consistency of values is defined as a lack of inconsistency to allow for consistent and circular data structures):

[2] We choose to make partial inconsistency the primary definition to match Definition 5.4.1.

Definition 5.4.1 *$v : \tau$ is consistent with H under R* (with $\mathrm{fv}(\tau) \subseteq \mathrm{dom}(R)$) if it is not inconsistent with H under R. We say *$v : \tau$ is inconsistent with H under R*:

- if $v = 0$ and $\tau = \mu @ \sigma$ then $R\sigma \neq 0$.
- if $\tau = \texttt{region}@\sigma$ and $H(v) = ((r), ())$ then $r \neq R\sigma$
- if $\tau = T[\sigma_1, \ldots, \sigma_m]@\sigma$, T is defined by $\texttt{struct}\ T[\rho_1, \ldots, \rho_m]\{f_1 : \tau_1, \ldots, f_n : \tau_n\}$ and $H(v) = ((r_0, r_1, \ldots, r_m), (v_1, \ldots, v_n))$. The property holds if $(r_0 \neq R\sigma) \vee (\exists j. R\sigma_j \neq r_j) \vee (\exists i. v_i : \tau_i$ is inconsistent with H under $[\rho_1 = r_1, \ldots, \rho_m = r_m])$.
- if $\tau = \exists \rho / \delta . \tau'$ then for all $r \in A_H$ such that $R[\rho = r] \models \delta$, we have $v : \tau'$ inconsistent with H under $R[\rho = r]$

Definition 5.4.2 A *set of values $v_1 : \tau_1, \ldots, v_n : \tau_n$ is consistent with H under R* if each $v_i : \tau_i$ is consistent with H under R.

Note that consistency of a value is a generalisation to all objects reachable from a value of the partial consistency relation used by rlang's semantics.

The main soundness theorem is as follows:

Theorem 5.4.3 *Soundness:* If

- $\delta, L \vdash s, \delta'$
- $< H, E, R, s > \rightsquigarrow < H', E', R' >$
- Variables $x_1 : \tau_1, \ldots, x_n : \tau_n$ are live before s
- Variables $x'_1 : \tau'_1, \ldots, x'_m : \tau'_m$ are live after s
- $R \models \delta$
- $E(x_1) : \tau_1, \ldots, E(x_n) : \tau_n$ are consistent with H under R
- $w_1 : \alpha_1, \ldots, w_l : \alpha_l$ are consistent with H under Q; the w_i are used to guarantee that the consistency of local variables are preserved during function calls

then

- $R' \models \delta'$

- $E'(x'_1) : \tau'_1, \ldots, E'(x_m) : \tau'_m$ are consistent with H' under R'
- $w_1 : \alpha_1, \ldots, w_l : \alpha_l$ are consistent with H' under Q
- The (a_inst_unsafe) assignment rule is not used in the semantic reduction

Proof: See below.

5.5 Soundness Proof

We start with some simple lemmas, then prove the soundness of the assignment rules (Lemma 5.5.11) and the main soundness theorem by induction on the structure of evaluations.

Lemma 5.5.1 If $R|_{\mathrm{fv}(\tau)} = R'|_{\mathrm{fv}(\tau)}$ then $v : \tau$ is consistent with H under R iff $v : \tau$ is consistent with H under R'.
Proof: obvious from the definition of consistency.

Lemma 5.5.2 $v : \tau$ is consistent with H under $R[\rho = R\sigma]$ iff $v : \tau[\sigma/\rho]$ is consistent with H under R.
Proof: obvious from the definition of consistency.

Lemma 5.5.3 Let τ be a type and $\rho_1, \ldots, \rho_m$ be distinct abstract regions with $\mathrm{fv}(\tau) \subseteq \{\rho_1, \ldots, \rho_m\}$. Then $v : \tau$ is consistent with H under $[\rho_1 = R\sigma_1, \ldots, \rho_m = R\sigma_m]$ iff $v : \tau[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m]$ is consistent with H under R.
Proof: follows from Lemmas 5.5.2 and 5.5.1.

The proofs of the following simple properties of $\models$ are easy:

Lemma 5.5.4 Let R be an abstract region map, δ a region property, $\mathrm{fv}(\delta) = \{\rho_1, \ldots, \rho_m\}$. Then $R \models \delta[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m]$ iff $[\rho_1 = R\sigma_1, \ldots, \rho_m = R\sigma_m] \models \delta$.

Lemma 5.5.5 If $R \models \delta$ and $R \models \delta'$ then $R \models \delta \wedge \delta'$.

Lemma 5.5.6 If $R \models \delta$ and $\delta \Rightarrow \delta'$ then $R \models \delta'$.

Lemma 5.5.7 If $R \models \delta$ and $R|_{\mathrm{fv}(\delta)} = R'|_{\mathrm{fv}(\delta)}$ then $R' \models \delta$.

Lemma 5.5.8 If r is some region, $R[\rho' = r] \models \delta$ and $\rho \notin \text{fv}(\delta)$ then $R[\rho = r] \models \delta[\rho/\rho']$.

Lemma 5.5.9 If $v : \tau$ is consistent with H under R then $v : \tau$ is partially consistent with H under R.

Lemma 5.5.10 If $v : \exists\rho/\delta.\tau$ is consistent with H under R, and r is such that $R[\rho = r] \models \delta$ and $v : \tau$ is partially consistent with H under $R[\rho = r]$, then $v : \tau$ is consistent with H under $R[\rho = r]$.

Proof: From definition of consistency and partial consistency we see that the abstract region map used for checking the consistency of fields of objects does not depend on the instantiation of quantified variables.

Lemma 5.5.11 *Assignability* If:

- $\delta, L \vdash \tau_1 \leftarrow \tau_2, \delta', L'$
- $< H, R, v, \tau_1, \tau_2 > \leadsto R'$,
- $v : \tau_2$ is consistent with H under R
- $R \models \delta$

then $v : \tau_1$ is consistent with H under R', $R' \models \delta'$ and $R|_L = R'|_L$. Additionally, the (a_inst_unsafe) assignment rule is not used in the reduction.

Proof: By induction on the structure of the evaluation of the type assignment. The proof considers each reduction rule in turn, and each case starts with the reduction and type checking rules used.

- $\dfrac{< H, R, v, \tau[\sigma/\rho], \tau' > \leadsto R'}{< H, R, v, \exists\rho/\delta'.\tau, \tau' > \leadsto R'}$

  $$\frac{\sigma \in L \cup C_R \qquad \text{fv}(\delta'[\sigma/\rho]) \subseteq L \qquad \delta \Rightarrow \delta'[\sigma/\rho] \qquad \delta, L \vdash \tau[\sigma/\rho] \leftarrow \tau', \delta'', L'}{\delta, L \vdash \exists\rho/\delta'.\tau \leftarrow \tau', \delta'', L'}$$

  First we note that $R \models \delta$ and $\delta \Rightarrow \delta'[\sigma/\rho]$ implies that $R[\rho = R\sigma] \models \delta'$ (Lemma 5.5.6).

  By induction $v : \tau[\sigma/\rho]$ is consistent with H under R', $R' \models \delta''$ and $R|_L = R'|_L$. By Lemma 5.5.2 $v : \tau$ is consistent with H under $R'[\rho = R'\sigma]$. As $R|_L = R'|_L$, $\sigma \in L \cup C_R$, $\text{fv}(\delta'[\sigma/\rho]) \subseteq L$ and $R[\rho = R\sigma] \models \delta'$ we conclude that $R'[\rho = R'\sigma] \models \delta'$. Therefore by the definition of consistency for existential types, $v : \exists\rho/\delta.\tau$ is consistent with H under R'.

- $$\frac{\begin{array}{c}\text{there exists } r \in A_H \text{ such that} \\ R[\rho' = r] \models \delta' \text{ and } v : \tau' \text{ is partially consistent with } H \text{ under } R[\rho' = r] \\ < H, R[\rho = r], v, \tau, \tau'[\rho/\rho'] > \leadsto R'\end{array}}{< H, R, v, \tau, \exists\rho'/\delta'.\tau' > \leadsto R'}$$

$$\frac{\rho \notin L \qquad \delta \Rightarrow \delta'' \qquad \mathrm{fv}(\delta'') \subseteq L \qquad \delta'' \wedge \delta'[\rho/\rho'], L \cup \{\rho\} \vdash \tau \leftarrow \tau'[\rho/\rho'], \delta''', L'}{\delta, L \vdash \tau \leftarrow \exists\rho'/\delta'.\tau', \delta''', L'}$$

By the hypothesis, $v : \exists\rho'/\delta'.\tau'$ is consistent with H under R, so by Lemma 5.5.10 $v : \tau'$ is consistent with H under $R[\rho' = r]$. By Lemmas 5.5.2 and 5.5.1, $v : \tau'[\rho/\rho']$ is consistent with H under $R[\rho = r]$. From $R \models \delta$, $\delta \Rightarrow \delta''$ and $\rho \notin L$, $\mathrm{fv}(\delta'') \subseteq L$ we get $R[\rho = r] \models \delta''$ (Lemmas 5.5.6 and 5.5.7). By Lemma 5.5.8 $\rho \notin L$, $R[\rho' = r] \models \delta'$ implies that $R[\rho = r] \models \delta'[\rho/\rho']$. Therefore $R[\rho = r] \models \delta'' \wedge \delta'[\rho/\rho']$ so, by induction, $v : \tau$ is consistent with H under R', $R' \models \delta'''$ and $R|_L = R'|_L$.

- $$\frac{\begin{array}{c}\text{there does not exist } r \in A_H \text{ such that} \\ R[\rho' = r] \models \delta' \text{ and } v : \tau' \text{ is partially consistent with } H \text{ under } R[\rho' = r]\end{array}}{< H, R, v, \tau, \exists\rho'/\delta'.\tau' > \leadsto R}$$

By the hypothesis, $v : \exists\rho'/\delta'.\tau'$ is consistent with H under R so (Lemma 5.5.9) $v : \exists\rho'/\delta'.\tau'$ is partially consistent with H under R. Therefore there does exist $r \in A_H$ such that $R[\rho' = r] \models \delta'$ and $v : \tau'$ is partially consistent with H under $R[\rho' = r]$, a contradiction. Therefore this (a_inst_unsafe) rule can never be applied.

- $$\frac{}{< H, R, v, \texttt{region@}\sigma, \texttt{region@}\sigma' > \leadsto R[\sigma = R\sigma']}$$

$$\frac{\delta, L \vdash \sigma \leftarrow \sigma', \delta', L'}{\delta, L \vdash \texttt{region@}\sigma \leftarrow \texttt{region@}\sigma', \delta', L'}$$

$$\frac{\sigma \in L \cup C_R \qquad \delta \Rightarrow \sigma = \sigma'}{\delta, L \vdash \sigma \leftarrow \sigma', \delta, L} \qquad \frac{\sigma \notin L \qquad \delta \Rightarrow \delta' \qquad \mathrm{fv}(\delta') \subseteq L}{\delta, L \vdash \sigma \leftarrow \sigma', \delta' \wedge \sigma = \sigma', L \cup \{\sigma\}}$$

If $\sigma \in L$, then from $R \models \delta$ and $\delta \Rightarrow \sigma = \sigma'$, $R\sigma = R\sigma'$ so $R[\sigma = R\sigma'] = R$. Therefore $v : \texttt{region@}\sigma'$ is consistent with H under $R[\sigma = R\sigma']$.

If $\sigma \notin L$, then $R[\sigma = R\sigma']|_L = R|_L$. So from $\delta \Rightarrow \delta'$, $\mathrm{fv}(\delta') \subseteq L$ and Lemmas 5.5.6 and 5.5.7 $R[\sigma = R\sigma'] \models \delta'$. And obviously, $R[\sigma = R\sigma'] \models \sigma = \sigma'$. So $R[\sigma = R\sigma'] \models \delta' \wedge \sigma = \sigma'$ and $v : \texttt{region@}\sigma'$ is consistent with H under $R[\sigma = R\sigma']$.

- $$\frac{R_1 = R[\sigma = R\sigma'] \qquad\qquad R_{i+1} = R_i[\sigma_i = R_i\sigma_i']}{< H, R, v, T[\sigma_1, \ldots, \sigma_m]@\sigma, T[\sigma_1', \ldots, \sigma_m']@\sigma' > \rightsquigarrow R_{m+1}}$$

  $$\frac{\delta, L \vdash \sigma \leftarrow \sigma', \delta_1, L_1 \qquad\qquad \delta_i, L_i \vdash \sigma_i \leftarrow \sigma_i', \delta_{i+1}, L_{i+1}}{\delta, L \vdash T[\sigma_1, \ldots, \sigma_m]@\sigma \leftarrow T[\sigma_1', \ldots, \sigma_m']@\sigma', \delta_{m+1}, L_{m+1}}$$

  The argument from the previous case is repeated $m + 1$ times.

Theorem 5.4.3: *Soundness*: If

- $\delta, L \vdash s, \delta'$
- $< H, E, R, s > \rightsquigarrow < H', E', R' >$
- Variables $x_1 : \tau_1, \ldots, x_n : \tau_n$ are live before s
- Variables $x_1' : \tau_1', \ldots, x_m' : \tau_m'$ are live after s
- $R \models \delta$
- $E(x_1) : \tau_1, \ldots, E(x_n) : \tau_n$ are consistent with H under R
- $w_1 : \alpha_1, \ldots, w_l : \alpha_l$ are consistent with H under Q. The w_i are used to guarantee that the consistency of local variables are preserved during function calls (see the function call case below).

then

- $R' \models \delta'$
- $E'(x_1') : \tau_1', \ldots, E'(x_m) : \tau_m'$ are consistent with H' under R'
- $w_1 : \alpha_1, \ldots, w_l : \alpha_l$ are consistent with H' under Q
- The (a_inst_unsafe) assignment rule is not used in the semantic reduction

Proof: By induction on the structure of the evaluation of s. The proof considers each reduction rule in turn (each case starts with the reduction and type checking rules). In rules where $H = H'$ we can immediately conclude that:

- $w_1 : \alpha_1, \ldots, w_l : \alpha_l$ are consistent with H' under Q.
- If $R|_L = R'|_L$, then all variables live after s that are not assigned in s are consistent with H' under R'.

In these rules we will thus only show that $R' \models \delta'$, $R|_L = R'|_L$, and live assigned variables are consistent with H under R'.

The fact that (a_inst_unsafe) is not used in the reduction follows from Lemma 5.5.11, used in all cases where semantic reduction rules invoke the semantic assignment rules. This fact is not repeated in the cases below.

- $$\frac{< H, E, R, s_1 > \leadsto < H', E', R' > \qquad < H', E', R', s_2 > \leadsto < H'', E'', R'' >}{< H, E, R, s_1; s_2 > \leadsto < H'', E'', R'' >}$$

  $$\frac{\delta, L \vdash s_1, \delta' \qquad \delta', L_{s_2} \vdash s_2, \delta''}{\delta, L \vdash s_1; s_2, \delta''}$$

  The live variables before $s_1; s_2$ are the same as those before s_1 so by induction, we conclude that $R' \models \delta'$, that the variables live after s_1 are consistent with H' under R' and that $w_1 : \alpha_1, \ldots, w_l : \alpha_l$ are consistent with H' under Q. The variables live after s_1 are the variables live before s_2 so by induction, we conclude that $R'' \models \delta''$, that variables live after s_2 (which are the same as those live after $s_1; s_2$) are consistent with H'' under R'' and that $w_1 : \alpha_1, \ldots, w_l : \alpha_l$ are consistent with H'' under Q.

- $$\frac{E(x) \neq 0 \qquad < H, E, R, s_1 > \leadsto < H', E', R' >}{< H, E, R, \texttt{if } x \ s_1 \ s_2 > \leadsto < H', E', R' >} \qquad \frac{\delta, L_{s_1} \vdash s_1, \delta' \qquad \delta, L_{s_2} \vdash s_2, \delta''}{\delta, L \vdash \texttt{if } x \ s_1 \ s_2, \delta' \vee \delta''}$$

  The live variables before s_1 are a subset of those before the `if`, so by induction, we conclude that $R' \models \delta'$, that variables live after s_1 (which are the same as those after the `if`) are consistent with H' under R', and that $w_1 : \alpha_1, \ldots, w_l : \alpha_l$ are consistent with H' under Q. By Lemma 5.5.6 and $\delta' \Rightarrow \delta' \vee \delta''$ we get $R' \models \delta' \vee \delta''$. The $E(x) = 0$ case is symmetric.

- $$\frac{< H, E, R, s > \leadsto < H', E', R' > \qquad \begin{array}{c} E(x) \neq 0 \\ < H', E', R', \texttt{while } x \ s > \leadsto < H'', E'', R'' > \end{array}}{< H, E, R, \texttt{while } x \ s > \leadsto < H'', E'', R'' >}$$

  $$\frac{\delta \vee \delta'', L_s \vdash s, \delta''}{\delta, L \vdash \texttt{while } x \ s, \delta \vee \delta''}$$

  The live variables before s are a subset of those before the `while` and $\delta \Rightarrow \delta \vee \delta''$ so by induction and Lemma 5.5.6, $R' \models \delta''$, the live variables after s are consistent with H' under R and $w_1 : \alpha_1, \ldots, w_l : \alpha_l$ are consistent with H' under Q. The variables live after s are a superset of those before the `while` and $\delta'' \Rightarrow \delta \vee \delta''$ so by induction and Lemma 5.5.6 $R'' \models \delta'$, the live variables after the `while` are consistent with H'' under R'' and $w_1 : \alpha_1, \ldots, w_l : \alpha_l$ are consistent with H'' under Q.

- $$\frac{E(x) = 0}{< H, E, R, \texttt{while}\ x\ s > \leadsto < H, E, R >} \qquad \frac{\delta \vee \delta'', L_s \vdash s, \delta''}{\delta, L \vdash \texttt{while}\ x\ s, \delta \vee \delta''}$$
The live variables after the `while` are a subset of those before it, and $\delta \Rightarrow \delta \vee \delta''$ so this case concludes.

- $$\frac{v_0 : \tau_0 \quad v_1 : \tau_1 \quad < H, R, E(v_1), \tau_0, \tau_1 > \leadsto R'}{< H, E, R, v_0 = v_1 > \leadsto < H, E[v_0 = E(v_1)], R' >} \qquad \frac{\delta, L \vdash \tau_0 \leftarrow \tau_1, \delta', L'}{\delta, L \vdash v_0 = v_1, \delta'}$$
By assumption, $E(v_1) : \tau_1$ is consistent with H under R and $R \models \delta$, so by Lemma 5.5.11, $E(v_1) : \tau_0$ is consistent with H under R', $R' \models \delta'$ and $R'|_L = R|_L$.

- $$\frac{\begin{array}{c} x_0 : \tau_0 \quad x_1 : T[\sigma_1, \ldots, \sigma_m]@\sigma \quad \texttt{struct}\ T[\rho_1, \ldots, \rho_m]\{\ldots, f_i : \tau_i, \ldots\} \\ H(E(x_1)) = ((r_0, \ldots), (v_1, \ldots, v_n)) \quad < H, R, v_i, \tau_0, \tau_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m] > \leadsto R' \end{array}}{< H, E, R, x_0 = x_1.f_i > \leadsto < H, E[x_0 = v_i], R' >}$$
$$\frac{\delta \wedge \sigma \neq \top, L \vdash \tau_0 \leftarrow \tau_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m], \delta', L'}{\delta, L \vdash x_0 = x_1.f_i, \delta'}$$
From the definition of a heap, $r_0 \neq 0$ so by consistency of $E(x_1)$ with H under R, $R\sigma = r_0 \neq 0$. Therefore by Lemma 5.5.5 $R \models \delta \wedge \sigma \neq \top$. Also $v_i : \tau_i$ is consistent with H under $[\rho_1 = R\sigma_1, \ldots, \rho_m = R\sigma_m]$, so (Lemma 5.5.3) $v_i : \tau_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m]$ is consistent with H under R. By Lemma 5.5.11 $v_i : \tau_0$ is consistent with H under R', $R' \models \delta'$ and $R'|_L = R|_L$.

- $$\frac{\begin{array}{c} x_1 : T[\sigma_1, \ldots, \sigma_m]@\sigma \quad \texttt{struct}\ T[\rho_1, \ldots, \rho_m]\{\ldots, f_i : \tau_i, \ldots\} \quad x_2 : \tau_2 \\ H(E(x_1)) = ((r_0, \ldots, r_m), (v_1, \ldots, v_n)) \\ < H, R, E(x_2), \tau_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m], \tau_2 > \leadsto R' \\ o = ((r_0, \ldots, r_m), (v_1, \ldots, v_{i-1}, E(x_2), v_{i+1}, \ldots, v_n)) \quad H' = H[E(x_1) = o] \end{array}}{< H, E, R, x_1.f_i = x_2 > \leadsto < H', E, R' >}$$
$$\frac{\delta \wedge \sigma \neq \top, L \vdash \tau_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m] \leftarrow \tau_2, \delta', L'}{\delta, L \vdash x_1.f_i = x_2, \delta'}$$
From the definition of a heap, $r_0 \neq 0$ so by consistency of $E(x_1)$ with H under R, $R\sigma = r_0 \neq 0$. Therefore by Lemma 5.5.5 $R \models \delta \wedge \sigma \neq \top$. Also $E(x_2) : \tau_2$ is consistent with H under R, so by Lemma 5.5.11 $E(x_2) : \tau_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m]$ is consistent with H under R', $R' \models \delta'$ and $R|_L = R'|_L$. By Lemma 5.5.3 $E(x_2) : \tau_i$ is consistent with H under $[\rho_1 = R'\sigma_1, \ldots, \rho_m = R'\sigma_m]$. Also $L = L'$ as $\{\sigma_1, \ldots, \sigma_m\} \subseteq L$, so $R = R'$.

We must show the consistency of $E(x'_1), \ldots, E(x'_m)$ (the values of the live variables after the field assignment) with H' under R and of $w_1, \ldots, w_l$ with H' under Q. The live variables after this statement are a subset of those live before it so we can replace the $E(x'_1), \ldots, E(x'_m)$ by $E(x_1), \ldots, E(x_n)$ (the values of the live variables before the field assignment). We consider these variables and the w_i's together by showing that there

is no value $v : \tau$ consistent with H under some abstract region map P and inconsistent with H' under P.

First we note that if $v : \tau$ is consistent with H' under P then it is not partially inconsistent with H' under P. Also $A_H = A_{H'}$, the regions of heap objects are unchanged in H and H' and $\text{dom}(H) = \text{dom}(H')$. Thus partial inconsistency of $v : \tau$ with H' under P is equivalent to partial inconsistency of $v : \tau$ with H under P.

Assume there exists some value $v : \tau$ consistent with H under P and inconsistent with H' under P. Any proof of inconsistency can be reduced to one of the two following cases:

- $v : \tau$ is partially inconsistent with H' (so also with H) under P. But $v : \tau$ is consistent with H under P, a contradiction.
- There exists $w : U[\ldots]@_$ reachable in H' from v such that
  - $\texttt{struct}\ U[\rho'_1, \ldots, \rho'_p]\{f_1 : \tau'_1, \ldots, f_q : \tau'_q\}$
  - $H'(w) = ((s_0, \ldots, s_p), (w'_1, \ldots, w'_q))$
  - $w'_k : \tau'_k$ is partially inconsistent with H' under $P' = [\rho'_1 = s_1, \ldots, \rho'_q = s_q]$.

  So $w'_k : \tau'_k$ is partially inconsistent with H under P'. Note also that $H(w) = ((s_0, \ldots, s_p), \ldots)$ and w must be reachable in H from some some $v' : \tau'$ (either the value of a live variable or one of $w_1, \ldots, w_l$), with $v' : \tau'$ consistent with H under P. There are again two cases:
  - If $w \neq E(x_1)$ or $k \neq i$ (i.e., y_k is not the assigned field) then $w'_k : \tau'_k$ is not partially inconsistent with H under P', a contradiction.
  - If $w = E(x_1)$ and $k = i$ (i.e., we are considering the assigned field) we saw above that $E(x_2) = w'_k : \tau_i = \tau'_k$ is consistent with H under $[\rho_1 = R\sigma_1, \ldots, \rho_m = R\sigma_m]$. By the consistency of $E(x_1) : T[\sigma_1, \ldots, \sigma_m]@\sigma$ with R we conclude $P' = [\rho_1 = R\sigma_1, \ldots, \rho_m = R\sigma_m]$ so $w'_k : \tau'_k$ is not partially inconsistent with H under P', a contradiction.

- $$\frac{x_0 : \mu_0@\sigma_0}{< H, E, x_0 = \texttt{null} > \rightsquigarrow < H, E[x_0 = 0], R[\sigma_0 = 0] >} \qquad \frac{\delta, L \vdash \mu_0@\sigma_0 \leftarrow \mu_0@\top, \delta', L'}{\delta, L \vdash x_0 = \texttt{null}, \delta'}$$

We show that $R|_L = R[\sigma_0 = 0]|_L$:

- $\sigma_0 \notin L$: obvious.
- $\sigma_0 \in L$: from the assignment rules we get $\delta \Rightarrow \sigma_0 = \top$ so from $R \models \delta$ and Lemma 5.5.6 we get $R\sigma_0 = 0$. Therefore $R|_L = R[\sigma_0 = 0]|_L$.

Also x_0: $E[x_0 = 0](x_0) = 0 : \mu_0@\sigma_0$ is consistent with H under $R[\sigma_0 = 0]$.

Next we show that $\delta, L \vdash \sigma_0 \leftarrow \top, \delta_1, L_1$ (the first step of the assignment) implies that $R[\sigma_0 = 0] \models \delta_1$:

- $\sigma_0 \in L$: from the assignment rules $\delta \equiv \delta_1$ so $R[\sigma_0 = 0] = R \models \delta_1$.
- $\sigma_0 \notin L$: from Lemmas 5.5.6, 5.5.7 and 5.5.5 we conclude that $R[\sigma_0 = 0] \models \delta_1$.

If $\mu_0 = \texttt{region}$ we have shown $R[\sigma_0 = 0] \models \delta'$. If $\mu_0 = T[\sigma_1, \ldots, \sigma_m]$ then we have $\delta_i, L_i \vdash \sigma_i \leftarrow \sigma_i, \delta_{i+1}, L_{i+1}$. By induction on i. we show that $R[\sigma_0 = 0] \models \delta_i$ for $i = 1 \ldots m+1$:

- $i = 1$: see above.
- $i \neq 1, \sigma_i \in L$: then $\delta_{i+1} \equiv \delta_i$ so by induction, $R[\sigma_0 = 0] \models \delta_{i+1}$.
- $i \neq 1, \sigma_i \notin L$: by Lemmas 5.5.6 and 5.5.5 $R[\sigma_0 = 0] \models \delta_{i+1}$.

So $R[\sigma_0 = 0] \models \delta'$.

- $$\frac{\begin{array}{c} x_0 : \tau_0 \qquad x_i : \tau_i \qquad x' : \texttt{region}@\sigma' \qquad \texttt{struct}\ T[\rho_1, \ldots, \rho_m]\{f_1 : \tau'_1, \ldots, f_n : \tau'_n\} \\ < H, R_i, E(x_i), \tau'_i[\sigma_1/\rho_1, \ldots, \sigma_m, \rho_m], \tau_i > \rightsquigarrow R_{i+1} \\ < H[v = o], R_{n+1}, x, \tau_0, T[\sigma_1, \ldots, \sigma_m]@\sigma' > \rightsquigarrow R' \\ v \notin \mathrm{dom}(H) \wedge v \neq 0 \qquad\qquad H(E(x')) = (\texttt{region}, r, _) \\ o = ((r, R_{n+1}\sigma_1, \ldots, R_{n+1}\sigma_m), (E(x_1), \ldots, E(x_n))) \end{array}}{< H, E, R_1, x_0 = \texttt{new}\ T[\sigma_1, \ldots, \sigma_m](x_1, \ldots, x_n)@x' > \rightsquigarrow < H[v = o], E[x_0 = v], R' >}$$

$$\frac{\begin{array}{c} \delta_i, L_i \vdash \tau'_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m] \leftarrow \tau_i, \delta_{i+1}, L_{i+1} \\ \delta_{n+1}, L_{n+1} \vdash \tau_0 \leftarrow T[\sigma_1, \ldots, \sigma_m]@\sigma', \delta', L' \end{array}}{\delta_1, L_1 \vdash x_0 = \texttt{new}\ T[\sigma_1, \ldots, \sigma_m](x_1, \ldots, x_n)@x', \delta'}$$

By induction on $1 \ldots n$ and Lemmas 5.5.11, 5.5.3 and 5.5.1 we conclude that $E(x_i) : \tau'_i$ are consistent with $[\rho_1 = R_{n+1}\sigma_1, \ldots, \rho_m = R_{n+1}\sigma_m]$, $R_{n+1} \models \delta_{n+1}$ and $R_{n+1}|_L = R$. As $v \notin \mathrm{dom}(H)$ (so v is not a value in the environment or in any object in H), $v : T[\sigma_1, \ldots, \sigma_m]@\sigma$ is consistent with $H[v = o]$ under R_{n+1}, and all live variables after this statement are consistent with $H[v = o]$ under R_{n+1}. Similarly $w_1 : \alpha_1, \ldots, w_l : \alpha_l$ are consistent with $H[v = 0]$ under Q.

The assignment to x_0 is a special case of the $x_0 = x_1$ rule seen above (replacing $E(x_1)$ by v).

- $$\frac{R \models \delta'}{< H, E, R, \texttt{chk}\ \delta' > \rightsquigarrow < H, E, R >} \qquad \frac{\mathrm{fv}(\delta') \subseteq L}{\delta, L \vdash \texttt{chk}\ \delta', \delta \wedge \delta'}$$

We only need to show that $R \models \delta \wedge \delta'$. This follows immediately from Lemma 5.5.5.

$$
\frac{\begin{array}{c} f[\rho_1,\ldots,\rho_m]/\delta'(y_1:\tau_1',\ldots y_n:\tau_n'):\tau',\delta'' \texttt{ is } [\rho_1',\ldots,\rho_p']y_1':\tau_1'',\ldots,y_q':\tau_q'',s,y \\ E_f=[y_1=E(x_1),\ldots,y_n=E(x_n),y_1'=0,\ldots,y_q'=0] \\ R_f=[\rho_1=R_{n+1}\sigma_1,\ldots,\rho_m=R_{n+1}\sigma_m,\rho_1'=0,\ldots,\rho_p'=0] \\ <H,E_f,R_f,s>\leadsto<H',E_f',R_f'> \\ x_i:\tau_i \qquad <H,R_i,E(x_i),\tau_i'[\sigma_1/\rho_1,\ldots,\sigma_m,\rho_m],\tau_i>\leadsto R_{i+1} \\ <H',R_{n+1},E_f'(y),\tau_0,\tau'[\sigma_1/\rho_1,\ldots,\sigma_m,\rho_m]>\leadsto R' \end{array}}{<H,E,R_1,x_0=f[\sigma_1,\ldots,\sigma_m](x_1,\ldots,x_n)>\leadsto<H',E[x_0=E_f'(y)],R'>}
$$

$$
\frac{\begin{array}{c} \delta_i,L_i\vdash\tau_i'[\sigma_1/\rho_1,\ldots,\sigma_m/\rho_m]\leftarrow\tau_i,\delta_{i+1},L_{i+1} \qquad \delta_{n+1}\Rightarrow\delta'[\sigma_1/\rho_1,\ldots,\sigma_m/\rho_m] \\ \delta_{n+1}\wedge\delta''[\sigma_1/\rho_1,\ldots,\sigma_m/\rho_m],L_{n+1}\vdash\tau_0\leftarrow\tau'[\sigma_1/\rho_1,\ldots,\sigma_m/\rho_m],\delta''',L' \\ \delta',L_s\vdash s,\delta^{iv} \qquad \delta^{iv}\Rightarrow\delta'' \qquad \mathrm{fv}(\delta')\cup\mathrm{fv}(\delta'')\subseteq\{\rho_1,\ldots,\rho_m\} \qquad y_1',\ldots,y_q' \text{ dead before } s \end{array}}{\delta_1,L_1\vdash x_0=f[\sigma_1,\ldots,\sigma_m](x_1,\ldots,x_n),\delta'''}
$$

By induction on $1\ldots n$ we conclude that $\forall i.E(x_i):\tau_i'[\sigma_1/\rho_1,\ldots,\sigma_m/\rho_m]$ is consistent with H under R_{n+1}, and $R_{n+1}\models\delta_{n+1}$. Therefore (Lemma 5.5.6) $R_{n+1}\models\delta'[\sigma_1/\rho_1,\ldots,\sigma_m/\rho_m]$. By Lemmas 5.5.1 and 5.5.3 $\forall i.E(x_i)=E_f(y_i):\tau_i'$ is consistent with H under R_f and by Lemma 5.5.4 $R_f\models\delta'$. By assumption, $y_1',\ldots,y_q'$ are dead before s, so all live variables before s are consistent with H under R_f.

Let N be the abstract regions in L_{n+1} ($N=L_{n+1}-C_R$). For simplicity of exposition, we assume $N\cap\mathrm{dom}(Q)=\emptyset$ (this is easily achieved by suitable renaming). We define Q' as $Q'\sigma=Q\sigma$ if $\sigma\in\mathrm{dom}(Q)$ and $Q'\sigma=R_{n+1}\sigma$ if $\sigma\in N$. Let the live variables before the function call be $x_1'':\tau_1'',\ldots,x_p'':\tau_p''$.

By induction, with extra values $w_1:\alpha_1,\ldots,w_l:\alpha_l,E(x_1''):\tau_1'',\ldots,E(x_p''):\tau_p''$ consistent with H under Q', we conclude that $R_f'\models\delta^{iv}$ (and by Lemma 5.5.6 $R_f'\models\delta''$), $E_f'(y):\tau'$ is consistent with H' under R_f' and $w_1:\alpha_1,\ldots,w_l:\alpha_l,E(x_1''):\tau_1'',\ldots,E(x_p''):\tau_p''$ consistent with H' under Q'. From this we conclude that $w_1:\alpha_1,\ldots,w_l:\alpha_l$ are consistent with H' under Q.

The typechecking rules specify that the L sets of f all contain $\{\rho_1,\ldots,\rho_m\}$ so $\forall i.R_f'\rho_i=R_f\rho_i=R_{n+1}\sigma_i$. Therefore R_{n+1} (Lemma 5.5.4) is consistent with $\delta''[\sigma_1/\rho_1,\ldots,\sigma_m/\rho_m]$. By Lemmas 5.5.1 and 5.5.3 $E_f'(y):\tau'[\sigma_1/\rho_1,\ldots,\sigma_m/\rho_m]$ is consistent with H' under R_{n+1}. By Lemma 5.5.5 $R_{n+1}\models\delta_{n+1}\wedge\delta''[\sigma_1/\rho_1,\ldots,\sigma_m/\rho_m]$. By Lemma 5.5.11, $E[x_0=E_f'(y)](x_0)=E_f'(y):\tau_0$ is consistent with H' under R' and $R'\models\delta'''$.

Finally, $\forall\sigma\in L_{n+1}.R'\sigma=R_{n+1}\sigma=Q'\sigma$, so all live variables other than x_0 are also consistent with H' under R'.

5.6 Translating RC to rlang

There are severals ways RC can be translated to rlang. For instance, one could apply a "region inference"-like algorithm [55] to RC programs, representing the results in rlang, in an attempt to find a very precise description of the program's region structure. Our goal is different: we want to translate an RC program P into an rlang program P' that faithfully matches P, then analyse P' to verify the correctness of assignments to pointers annotated with `sameregion`, `parentptr` and `traditional`. We therefore perform a straightforward translation, while guaranteeing the following properties of P':

- There is one region constant, R_T, for the "traditional region".

- For every structured type X in P there is a structured type $X[\rho]$ in P'. The abstract region ρ represents the region in which the structure is stored. So pointers to X in P' are always of the form $X[\sigma]@\sigma$.

  A field f in $X[\rho]$ of type T which is not `sameregion`, `parentptr` or `traditional` in P can point to any region. So its type in P' is $\exists\rho'.T[\rho']@\rho'$. A `traditional` f can be `null` or point to the traditional region so its type is $\exists\rho'/\rho' = \top \vee \rho' = R_T.T[\rho']@\rho'$. A `sameregion` f can be `null` or point to an object in ρ, so its type is $\exists\rho'/\rho' = \top \vee \rho' = \rho.T[\rho']@\rho'$. Finally, a `parentptr` f can point upwards in the region hierarchy (which includes being `null` as the region of `null` values is $\top$), so its type is $\exists\rho'/\rho \preceq \rho'.T[\rho']@\rho'$ For example,

```
struct L {
  region v;
  L *sameregion n;
};
```

  $\Rightarrow$

  $$\begin{array}{l} \texttt{struct}\ L[\rho]\ \{ \\ \quad v : \exists\rho'.\texttt{region}@\rho', \\ \quad n : \exists\rho'/\rho' = \top \vee \rho' = \rho.L[\rho']@\rho \\ \} \end{array}$$

  Global variables are represented as fields of a `Global` structure, stored in the traditional region, which is passed to every function.

- Every local variable and function argument x in P' is associated with a distinct abstract region ρ_x. If x is of type T in P, its type becomes $T[\rho_x]@\rho_x$ in P'. Function arguments are never assigned or used directly as the function result, and the destination of an assignment is not used elsewhere in the assignment statement. [3]

[3] This last restriction is due to the rules for handling liveness in Figure 5.3.

- Every field assignment $x_1.f = x_2$ (with x_1, x_2 assumed local) is immediately preceded by an appropriate runtime check: `chk` $\rho_{x_2} = \top \vee \rho_{x_2} = \rho_{x_1}$ if f is `sameregion` in P; `chk` $\rho_{x_1} \preceq \rho_{x_2}$ if f is `parentptr`; `chk` $\rho_{x_2} = \top \vee \rho_{x_2} = R_T$ if f is `traditional`. This matches the model for these annotations given in Chapter 3.2.4: assignments will abort the program if the requirements of `sameregion`, `parentptr` or `traditional` are not met.

- We always represent the result of a function as an existential type. Combined with the rules above, a function f with one argument of type T and result of type T' always has signature

$$f[\rho_x]/\delta(x : T[\rho_x]@\rho_x) : \exists\rho/\delta'.T'[\rho]@\rho, \delta''$$

for some boolean expressions $\delta, \delta', \delta''$. This representation allows us to have the same type (ignoring the boolean expressions) for a function returning the region of its argument (`myregionof`) and for a function returning a new region (`mynewregion`):

$$\texttt{myregionof}[\rho_x]/\texttt{true}(x : T[\rho_x]@\rho_x) : \exists\rho/\rho = \rho_x.\texttt{region}@\rho, \texttt{true}$$

$$\texttt{mynewregion}[\rho_x]/\texttt{true}(x : T[\rho_x]@\rho_x) : \exists\rho/\texttt{true}.\texttt{region}@\rho, \texttt{true}$$

It is easy to verify that an rlang program with these properties can be type checked, under the assumption that all function input, output and result properties sets are `true`. The implementation of RC infers better properties than this simple approximation by casting the inference of input, output and result properties as a dataflow problem:

- The set of facts we consider in our analysis of a function f with abstract regions $\{\rho_1, \ldots, \rho_m\}$ are: $\sigma = \top$, $\sigma \neq \top$, $\sigma_1 \preceq \sigma_2$, $\sigma_1 = \top \vee \sigma_1 = \sigma_2$ for all $\sigma, \sigma_1, \sigma_2 \in \{\rho_1, \ldots, \rho_m\} \cup C_R$. We call each of these facts a *constraint*. A constraint set C corresponds to the boolean expression $\bigwedge_{\delta \in C} \delta$. Our inference system replaces all boolean expressions by these constraint sets.

- We conservatively approximate the type checking rules for `if` and `while` by constraint set intersection. This is safe as

$$(\bigwedge_{\delta \in C} \delta) \vee (\bigwedge_{\delta \in C'} \delta) \Rightarrow \bigwedge_{\delta \in (C \cap C')} \delta$$

- Constraint sets form a finite-height lattice under set inclusion. The operations in the type checking rules are all monotonic when expressed in terms of constraint sets and

there is a least solution (all properties set to `true`, i.e., all constraint sets empty). Therefore it is possible to find the best collection of constraint sets using a greatest-fixed-point-seeking dataflow analysis of the whole program. This greatest-fixed-point for constraint sets is also the most precise typing possible (using these constraint sets).

- RC restricts this dataflow analysis to a single source file by assuming that any non-static C function and any function called via a function pointer has empty input, output and result constraint sets. The complexity of this analysis is $O(kSn^4)$, where k is the number of functions in a file, S the number of statements, and n the greatest number of local variables in a single function. We keep the analysis tractable by ignoring local variables that are effectively temporaries (all uses have a single reaching definition). The largest analysis time on any file in our benchmarks is 30s, with all other times being less than 10s. The analysis completes in less than 1s for 96% of files.

Once the inference is complete, we can safely eliminate any `chk` statement that asserts a property that is implied by its input constraint set. Results of this analysis are presented in Chapter 6.8.

5.7 Alternate Translations of RC to rlang

We have considered alternate ways of translating RC programs to rlang. Possible changes come in three categories, which are discussed at greater length below:

- Use of the extensions to RC discussed in Chapter 3.4 will lead to more precise rlang types, and therefore allow the static verification of more runtime checks (Chapter 5.7.1).
- The model for runtime checks in RC could be changed, e.g., by requiring that all assignments to annotated fields be statically verifiable (Chapter 5.7.2).
- The region aspects of rlang types could be inferred from the RC source rather than relying on the user to provide annotations in the form of type qualifiers (Chapter 5.7.3).

5.7.1 Extensions to RC

The addition of a new pointer type qualifier such as `childptr` is easy to incorporate into the translation to rlang. A field f in $X[\rho]$ of type T that is `childptr` can be `null` or point to an object in a subregion of ρ, so its type is $\exists\rho'/\rho' = \top \vee \rho' \preceq \rho$. The constraint sets used in type inference extend naturally to include the `childptr` fact $\rho' = \top \vee \rho' \preceq \rho$. Other potential pointer type qualifiers for RC can most likely be handled in a similar fashion.

The function qualifiers proposed in Chapter 3.4 are also easily modeled in rlang. It is sufficient to add appropriate runtime checks at every call site: in a call $f(a_1, \ldots, a_n)$ (we assume all arguments are local variables for simplicity) to a function f annotated with (p_n is the nth parameter to f):

- `sameregion(`p_i, p_j`)`: precede the call with

$$\texttt{chk}\ \rho_{a_i} = \top \vee \rho_{a_j} = \top \vee \rho_{a_i} = \rho_{a_j}$$

- `parentptr(`p_i, p_j`)`: precede the call with

$$\texttt{chk}\ \rho_{a_j} = \top \vee \rho_{a_j} \preceq \rho_{a_i}$$

As with `childptr` is is necessary to extend constraint sets to represent the facts verified by these runtime checks. If the type inference is performed on the whole program, these runtime checks are sufficient to guarantee that the properties of an annotated function f will hold in f: every call to f will be preceded by the necessary checks, so the greatest fixed point solution of the type inference for f will include the properties asserted by these runtime checks. In the presence of separate compilation we must change the assumption that any non-static C function and any function called via a function pointer has empty input, output and result constraint sets. Instead, we can assume that the properties of an annotated function f always hold inside f and give f the appropriate input constraint set.

A similar process can be used to model the annotations that describe `sameregion` and `parentptr` properties of a function's result.

5.7.2 Runtime Checks

There are three reasonable models for runtime checks for the RC annotations:

- Perform a runtime check at all uses of the annotations, i.e., at all assignments to annotated types (and, if added to RC, at all calls to annotated functions as described above). This is the approach we selected. The type inference process is used to eliminate unnecessary checks.

- Optimise the placement of runtime checks: allow runtime checks to be moved from their natural placement, under the condition that a moved check must not cause a program that would have succeeded to fail. For instance, a check on both branches of an `if` could be moved before the `if`, or a check executed on all calls to a function f could be moved before all calls to f (it might be possible to show that the check is safe at some call sites). This appears to be a hard optimisation problem (requiring at the very least an execution profile estimate), requires whole program analysis to do much movement of checks out of functions, and could produce unintuitive behaviour as a failed check would not occur where the programmer expects it.

- Require that all annotations be statically verifiable. This corresponds to translating RC programs to rlang programs with no uses of the `chk` statement. The type inference process can then be used to see if the resulting program is type-correct. To be useful this requires either whole program analysis of RC programs or the use (and static verification) of the proposed function annotations. In the spirit of RC's preference of greater program flexibility via dynamic rather than static safety, we chose to keep runtime checks.

5.7.3 Annotation Inference

An analysis that infers `sameregion` fields in an RC program (with no annotations) could be built as follows:

- Translate the RC program to rlang as in Chapter 5.6. Note that there will be no `chk` statements as there are no type annotations.

- Extend rlang's region properties to include boolean variables with global scope. For every field f in $X[\rho]$ of type T add a boolean variable s_{X_f} and give f the type $\exists\rho'/s_{X_f} \Rightarrow (\rho' = \top \vee \rho' = \rho).T[\rho']@\rho'$. This variable, if true, will require that f be `sameregion`.

- Find, using an appropriate type inference algorithm, a set of values for the s_{X_f} variables that produces a legal typing for the rlang program. Note that is always at least one legal typing where all boolean variables are false (no fields are `sameregion`).

This scheme can obviously be extended to the `parentptr` and `traditional` annotations at the expense of greater complexity.

We did not choose to investigate this path further for several reasons. First, it requires a complex, unpredictable whole program analysis. Small changes to the code may prevent a field from being `sameregion`. Secondly, we have found in our benchmarks that in many cases many, but not all, assignments to a field declared `sameregion` can be statically verified. This inference scheme would not be able to infer that such a field is `sameregion`. Finally, and most importantly, this approach would lose the documentation of programmer intent that is implicit in the `sameregion` declarations.

Chapter 6

Results

We designed and evaluated RC with the help of eight allocation-intensive C programs. We briefly present these programs in Chapter 6.1, describe how they use regions in Chapter 6.2 and summarise the changes necessary to run these programs with RC in Chapter 6.3.

The second half of this chapter discusses the results of running these benchmarks with RC and other allocators. Chapter 6.4 presents these other allocators and our test environment. Chapter 6.5 gives an overview of each benchmark's allocation and pointer-write behaviour. We compare RC's memory consumption (Chapter 6.6) and performance (Chapter 6.7) with the other allocators. In these chapters we also compare RC to our old system C@ and to the alternative reference-counting schemes of Chapter 4.3.5. We then investigate in more detail the performance of our type qualifiers (Chapter 6.8) and our schemes of Chapter 4.3.4 for reducing the cost of reference-counting local variables (Chapter 6.9). Finally, we measure the overhead RC imposes (due to compiling to C and our special region library) in Chapter 6.10 and approximate the cost of a multi-threaded reference-counting implementation in Chapter 6.11. We end with a summary of our results (Chapter 6.12).

6.1 Benchmarks

Our benchmarks, and their respective inputs are:

- *cfrac*: A program to factor large integers using the continued fraction method. The original application used explicit reference counting to reclaim storage. We factor

417576463441248601459380302877 1.

- *gröbner*: Find the Gröbner's basis of a set of polynomials. This program was originally written using `malloc` and `free`. The input is nine nine-variable polynomials.
- *mudlle*: A byte-code compiler for a scheme-like language. The original version of this program uses unsafe regions. We compile a 500-line file 50 times, a 1700-line file 30 times and a 1000-line file 30 times.
- *lcc*: Our modified version of the lcc [24] C compiler. The original program also uses unsafe regions (Hanson's arenas [32]). The input is a 6000-line C file.
- *moss*: A software plagiarism detection system, written originally using `malloc` and `free`. The input is 180 student compiler projects (about 10MB).
- *tile*: Automatically partitions a set of text files into subsections based on frequency and grouping of words in the text. This program originally used `malloc` and `free`. Two copies of a draft of a 56k character paper [27] are given as input.
- *rc*: The RC compiler, written with RC's regions. The input is a 1202 line C file (2179 non-blank lines after preprocessing).
- *apache*: The Apache web server v1.3.12, originally written with unsafe regions. We request 2045 pages from the web server. The client requesting the pages runs on the same machine as apache as performance is otherwise completely dominated by transmission of pages over the network.

6.2 Region Structure

In this section, we present how each benchmark uses regions. For *mudlle*, *lcc*, *rc* and *apache* this is the benchmark's original region structure; for *cfrac*, *gröbner*, *moss* and *tile* it is the region structure we picked when converting these programs to use regions. In the descriptions of these structures below we will often refer to the example regions structures presented in Chapter 3.2.9.

The main loop of *cfrac* is an iterative computation (Chapter 3.2.9) whose main data structure is a multi-precision integer. Iteration n of this loop needs access to the integers created in iterations $n-1$ and $n-2$. In *cfrac*, every k iterations we create a region

for integers allocated in the next k iterations and delete the region created $2k$ iterations ago. As we need access to integers created two iterations ago, the minimum value for k is 2. Larger values of k decrease region creation and deletion overhead but increase memory usage. We found that $k = 10$ was a good compromise. There are also a few loops in `cfrac` which create a region at the start of each loop iteration and delete it at the end (the phase-based style of Chapter 3.2.9).

The main part of *gröbner* is a set of two nested loops, and the main data structure is a polynomial. The inner loop is an iterative computation like *cfrac*, but only needs access to polynomials created on the previous iteration. We create a region on every iteration for the polynomials that will be allocated in the current iteration, and delete the region created two iterations ago. The outer iteration maintains a region for the polynomials that form the basis set (the results region) and for the polynomials that are candidates for the basis set (the candidates region). When the inner loop finds (and returns) a new polynomial P for the basis set, the outer loop: copies P to the result region and builds a new candidates set in a new candidates region incorporating P and filtering out now-irrelevant polynomials. The old candidates region is then deleted. Finally, *gröbner* also uses a temporary region (Chapter 3.2.9) for parsing polynomials.

The two main data structures in *mudlle* are a parse tree representing a file being compiled, and a `fncode` structure used to compile a single function. The `fncode` structure is relatively complicated containing lists of the function's variables, of the byte-code instructions for the function, etc. A region is created for the parse tree, which is freed at the end of compilation. A region is created for each `fncode` structure, which is freed when the corresponding function has been compiled. The mudlle language has nested functions, the region for the `fncode` of a function g nested inside a function f is a child region of the `fncode` region of f. The `fncode` region of top-level functions is a child of the parse tree's region. Thus the region structure mimics the structure of the file being compiled and allows `parentptr` annotations to be used in a natural manner. As in *gröbner*, *mudlle* uses a temporary region during parsing.

There are always exactly three live regions in *lcc*. The permanent region holds data that lives for the whole duration of compilation, e.g., the objects representing global variables. The per-function region, a child of the permanent region, is deleted and recreated after every function is compiled and holds data necessary to compile a single function, e.g., the directed acyclic graph representing a C basic block. The per-statement region, a child

of the per-function region, is deleted and recreated after every C statement is compiled and holds data that is not needed after a single statement has been compiled, e.g., the types of that statement's expressions. Both the per-function and per-statement regions are examples of phase-based computation (Chapter 3.2.9).

A number of different permanent regions are used for different data structures in *moss*. By putting different data structures in different regions we express some of the locality properties of *moss*: objects in separate regions are less likely to conflict in the cache. Thus placing infrequently-accessed in a separate region than frequently-accessed objects can lead to locality improvements. This improvement can be seen in the performance results and L2 cache misses for *moss* in Chapter 6.7. In addition to these permanent regions, *moss* also builds a hash table for every function in the input programs. This hash-table region can be deleted at the end of each function (another example of phase-based computation) though we choose to delete this region every 10 functions to reduce region creation and deletion overhead.

Each data structure in *tile* (two hash-tables, a set of five arrays and the partitioning results) has its own regions. These are deleted at the end of partitioning or when the results are no longer needed. The size of the five arrays may need to be increased during partitioning, this is achieved by creating bigger arrays in a new array region, copying the old array contents and deleting the old array region.

The *rc* compiler has a similar structure to *mudlle*: a region for the parse tree for the file being compiled, and one region per function for all data needed while compiling that function. The per-function region is a child of the parse tree region and deleted when that function has been compiled. There are also a few other regions: one for allocating the structures representing C types and some temporary regions used while parsing C code and while printing the regular C code that is RC's output.

In *apache* there are five main regions: `pglobal` for permanent data, `pchild` for per-server data (this is only different from `pglobal` when running multiple server threads which we do not support), `plog` for logging-related allocations, `pconf` for configuration-related allocations and `ptrans` for per-TCP-connection allocations. The `plog` and `pconf` regions are deleted and recreated when the apache configuration changes, `ptrans` is created before a TCP connection is accepted and deleted after it is closed (each server handles one connection at a time). The `plog` and `pconf` regions are children of `pglobal`, `ptrans` is a child of `pconf`. Each request received by the server gets its own region, which is a

child of `ptrans`. This region is deleted when the request has been fully processed. Some requests cause the creation of subrequests, which again get their own region (a child of the creating request's regions). Subrequests can either complete (and be deleted) before their parent, or at the same time. In this latter case, we use `deleteregion_array` to delete a subrequest's region at the same time as the parent's region.[1] Finally, some *apache* functions use temporary or permanent regions.

6.3 Changes to Benchmarks

Except for *rc*, our benchmarks required changes to use RC's regions. The changes fall into the the following categories:

- *Regions*: The `malloc` and `free` based programs were changed to region-based allocation, using the region structure described above. Code is added to create and delete the appropriate regions, calls to `malloc` were replaced by region allocation and calls to `free` are removed. Some benchmarks required additional changes to keep track of when to delete regions, add additional copies, etc.
- *Regions*: The region-based programs are changed to use RC's region API. These changes are straightforward. Some additional changes are also required, e.g., to clear some pointers before deleting a region or to write `rc_adjust_x` functions for unions containing pointers.
- *Deletes*: The `deletes` qualifier must be added to any function that may delete a region. This is also straightforward (there is a compile-time error if any `deletes` qualifiers are forgotten).
- *Qualifiers*: Some `traditional`, `sameregion` and `parentptr` qualifiers were added to each benchmark. For some benchmarks, it was sufficient to add these qualifiers to appropriate types, for others small code changes were also necessary.
- *Performance*: We made a small number of changes to the benchmarks to improve performance, e.g., copying global variables to local variables to help qualifier runtime check elimination (Chapter 5.6).

[1] *apache*'s region model implicitly deletes child regions with their parents.

Name	Lines	Changed	Regions	Deletes	Qualifiers	Performance	Other
cfrac	3792	220 / 216	134 / 134	4 / 0	7	8	67
gröbner	2728	415 / 410	167 / 167	5 / 0	243	0	0
mudlle	4859	300 / 136	135 / 52	81 / 0	84	0	0
lcc	12827	438 / 272	129 / 55	92 / 0	166	51	0
moss	2673	105 / 97	58 / 58	8 / 0	21	18	0
tile	1319	150 / 138	111 / 111	12 / 0	19	8	0
rc	28434						
apache	52644	599 / 287	336 / 192	168 / 0	61	13	21

Table 6.1: Complexity of benchmark changes, in number of lines changed.

- *Other*: Some benchmarks required a few other changes.

Table 6.1 summarises the size of these changes in lines of code, broken down into the categories above. For *Changed* (the total number of lines changes), *Regions* and *Deletes* we include two figures: before the / is the number of lines changed while the number after the / excludes *mechanical changes*. *Mechanical changes* are the addition of the `deletes` qualifier and, for the programs that were originally region-based, the change to use RC's region API. For reference, we include the number of lines in *rc*. We also gives details on each benchmark's non-mechanical changes, by category:

- *cfrac*: For *regions*, we must copy solution to the results region. We also add some `sameregion` type qualifiers. For *performance* we avoid taking the address of a local variable (optimisation of reference counting is not done on variables whose address is taken). Under *other* changes we remove *cfrac*'s original reference counting.

  When running *cfrac* with the Boehm-Weiser conservative garbage collector, we also disable the original reference counting code.

- *gröbner*: For *regions*, we must copy solution to the results region and we use a temporary region for allocations during polynomial parsing rather than explicitly tracking allocated objects in an array. We changed the polynomial data structure to use `sameregion` pointers internally which required that some objects be region rather than stack allocated.

- *mudlle*: For *regions* we clear some global variables before deleting regions, use subregions and `regionof` and change the implemented language's generic value type from

`void *` to `unsigned long` to avoid storing integers in a pointer type. We add many `sameregion`, `parentptr` and `traditional` qualifiers, one of which requires a small (6 line) code rewrite.

- *lcc*: For *regions* we add some fields to an object to identify which member of a `union` is in use, eliminate uses of `memset` and `memcpy`, clear some global variables and object fields before deleting a region and use subregions. We add many `sameregion`, `parentptr` and `traditional` qualifiers and allocate some arrays in regions rather than as local or global variables to allow more use of `sameregion`. For *performance* we copy some global variables to locals to help qualifier runtime check elimination, remove clears or fields and local variables which are made unnecessary by RC's implicit initialisations and allocate some arrays in a region rather than on the stack.

- *moss*: We add some `sameregion` and `traditional` type qualifiers. For *performance* we copy some global variables to locals and use `regionof` to help qualifier runtime check elimination and we recreate a hash table's region every 10 functions rather than every function to reduce region creation and deletion overhead.

- *tile*: We add some `sameregion` and `traditional` type qualifiers. For *performance* we copy some global variables to locals and use `regionof` to help qualifier runtime check elimination.

- *apache*: For *regions* we must change *apache*'s growable array abstraction to incorporate type information, clear a few variables, replace the 'clear a region' function with a 'destroy region and create new region' function, eliminate uses of `memcpy` and slightly delay deleting some regions until all pointers to these regions have been overwritten. We add many `sameregion`, `parentptr` and `traditional` qualifiers. For *performance* we remove `memset`'s rendered unnecessary by RC's implicit initialisations and allocate an array with `alloca` rather than on the stack. Under *other* changes we work around the limitation on `strtol` (see Appendix A), move a structure declaration out of a function (as required by RC), and make an initialised local array `static`.

  There is one significant limitation in our port of *apache* to RC: the support for timeouts does not work as it uses `longjmp` which does not work in RC (see Chapter 3.2.5).

6.4 Allocators and Test Environment

We compare our RC-based programs with two other memory managers. The first is Doug Lea's malloc/free replacement library v2.7.0[2]. This is an improved version of the allocator used in some previous surveys of memory allocation costs [21, 56]. In those surveys this allocator exhibited good performance overall. In particular it has much better performance than Sun's default `malloc` library. The second allocator is the Boehm-Weiser conservative garbage collector [13] v6.0[3].

All benchmarks are compiled with `gcc` v2.95.2 and run on a 300MHz UltraSPARC-II with 128MB of memory and 512kB of L2 cache.

In the tables and figures below, we systematically use "RC" for RC with the standard options, "lea" for Doug Lea's `malloc/free` implementation and "bwgc" for the Boehm-Weiser conservative garbage collector. As in the implementation chapter, we will say "local variable" when we mean "local variable whose address is not taken". We will also use "pointer-free object (or page, or block)" as a shorthand for "object (or page, or block) containing only qualified pointers or non-pointer data".

Except where otherwise noted, all results below (except those marked as "lea" or "bwgc") are based on the region-based version of each application compiled by RC with the standard options. All execution times are wall-clock times and based on the best of five runs.

6.5 Benchmark Behaviour

The memory allocation characteristics of our benchmarks are summarised in Table 6.2 and Figures 6.1 and 6.2. The table shows the total number of bytes, objects and regions allocated, and the rate of these allocations. Figure 6.1 shows the distribution of object sizes in each application: the "32" column shows the number of allocated objects whose size is 32 bytes or less, the "128" column is for objects between 33 and 128 bytes, the "512" column for objects between 129 and 512 bytes and the "> 512" column for objects greater than 512 bytes. Figure 6.2 presents the same data, except that the height of each column is the number of bytes allocated for objects of that size, rather than the total number of objects.

[2] Obtainable at ftp://g.oswego.edu/pub/misc/malloc.c
[3] Obtainable at http://www.hpl.hp.com/personal/Hans_Boehm/gc

Name	kB Allocated		Objects Allocated		Regions Allocated	
	Total	Rate	Total	Rate	Total	Rate
cfrac	62987	10654/s	3812424	644861/s	23382	3955/s
grobner	327272	32451/s	5796711	574785/s	73435	7281/s
mudlle	27145	5874/s	1594630	345083/s	6729	1456/s
lcc	56852	7858/s	991527	137064/s	50878	7033/s
moss	9317	1288/s	554133	76643/s	1899	262/s
tile	381	72/s	10465	1992/s	12	2/s
rc	5703	2036/s	88159	31485/s	61	21/s
apache	31467	4009/s	175972	22419/s	4347	553/s

Table 6.2: Memory Allocation Rates

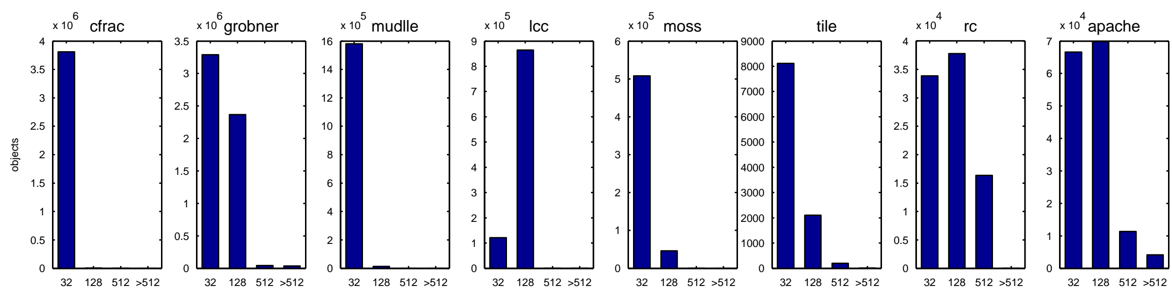


Figure 6.1: Objects allocated, distributed by object size

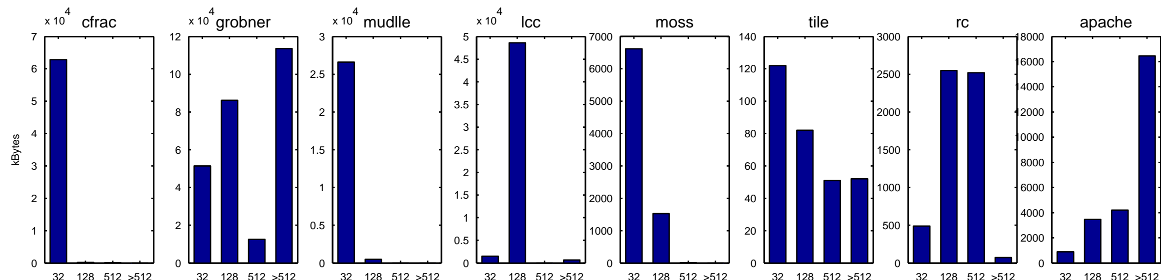


Figure 6.2: Bytes allocated, distributed by object size

Name	Blocks Allocated				Avg. Objects	Avg. kBytes.
	Total	Rate	8kB	Pointer-free	per Region	per Region
cfrac	23431	3963/s	100.0%	100.0%	163	2.7
grobner	87010	8627/s	90.6%	100.0%	78	4.5
mudlle	15424	3337/s	100.0%	99.1%	236	4.0
lcc	58995	8155/s	100.0%	55.0%	19	1.1
moss	4169	576/s	100.0%	86.5%	291	4.3
tile	49	9/s	91.8%	100.0%	871	27.8
rc	815	291/s	99.4%	37.8%	1444	93.2
apache	10748	1369/s	81.0%	59.3%	37	7.2

Table 6.3: Region Statistics

These results show that all programs, except *tile*, are allocation intensive, and all programs, except *tile* and *rc*, allocate a lot of regions. However the rate for allocating regions is much lower than for allocating objects. The two figures show that in all benchmarks most allocated objects are small, however for half the benchmarks (*gröbner*, *tile*, *rc*, *apache*) a significant fraction of the bytes are allocated in medium to large objects (> 128 bytes).

Table 6.3 gives some statistics on regions and the underlying region implementation. We report the number of blocks allocated (see Chapter 4.2.1), the rate of these allocations, the percentage of blocks allocated that are 8kB (a single page) and the percentage of blocks allocated that are for pointer-free objects. We also report the average number of objects per region and the average size of a region.

We observe that the rate of block allocation (Table 6.3) is much lower than the rate of object allocation (Table 6.2), between 16x lower for *apache* to 221x lower for *tile*. This is significant as block allocation is expensive (comparable to a `malloc` call) and object allocation is cheap as long as no new block is needed. The ratio of block to object allocations is lowest for the programs that have on average few objects per region (*lcc* and *apache*).

Table 6.3 also confirms that most allocated blocks are single 8kB pages (between 81% and 100% of all blocks allocated). This justifies our special support in the region allocator for allocating these blocks (the `single_blocks` list described in Chapter 4.2.1). The percentage of pointer-free pages varies substantially, from 37.8% for *rc* to a 100% for *cfrac* and *gröbner*.

Table 6.4 gives details on the number of writes to pointers ("Total") and the "Rate" of these writes, for local variables and for all other pointer writes. These figures

Name	Local Variable Writes		Other Writes	
	Total	Rate	Total	Rate
cfrac	76384647	12920271/s	778300	131647/s
grobner	89601466	8884627/s	6465365	641087/s
mudlle	41758171	9036609/s	14308483	3096404/s
lcc	45558781	6297868/s	9935789	1373484/s
moss	43074278	5957714/s	16213087	2242473/s
tile	4302005	819117/s	192023	36561/s
rc	31047240	11088300/s	1870773	668133/s
apache	4481649	570983/s	738990	94150/s

Table 6.4: Pointer write statistics

confirm that writes to local variables are the most important (their lowest percentage is 73% for *lcc*). Chapter 6.9 shows that we can avoid reference count operations for most of these local variable writes.

We see that a benchmark's rate of object allocation is not correlated with the rate at which it writes pointers. For instance, *cfrac* writes a pointer every 4.8 object allocations while *rc* allocates an object every 21.2 pointer writes.

6.6 Memory Usage

Table 6.5 shows the maximum memory usage for four allocators: our three standard allocators, "RC", "lea" and "bwgc", and "p" which is RC-pairs (Chapter 4.3.7). This table has two parts for the two versions of each benchmark (original and region-based) The "user" column is the maximum amount of memory in use at any time, from the application writer's perspective. For the region-based version we also include the maximum number of regions in existence at any time. We report the amount of memory requested from the operating by each allocator to satisfy the application's memory allocations, and the corresponding overhead ($\frac{\text{requested}}{\text{user}} - 1$). Figure 6.3 presents the same data in graphical form for easier comparison.

Excepting *apache*, RC has a memory usage and overhead which are comparable with Doug Lea's malloc/free implementation. The large memory overhead for RC in *apache* is due to the large number of simultaneous regions live (25): we examined these 25 regions and found that most needed a pointer-free and a not-pointer-free page, so take at least

Name	region-based user	RC	p	original user	lea	bwgc
cfrac	123(4)	136 / 11%	136 / 11%	101	112 / 11%	880 / 772%
grobner	87(4)	148 / 71%	155 / 79%	83	101 / 22%	488 / 489%
mudlle	283(12)	399 / 41%	341 / 21%	272	365 / 34%	1576 / 478%
lcc	4370(3)	5096 / 17%	4581 / 5%	4364	5187 / 19%	36392 / 734%
moss	2366(7)	2775 / 17%	2414 / 2%	2356	2998 / 27%	5008 / 113%
tile	191(6)	240 / 26%	241 / 26%	172	204 / 19%	264 / 54%
rc	4647(6)	5053 / 9%	4758 / 2%	4648	4946 / 6%	6680 / 44%
apache	108(25)	437 / 305%	295 / 174%	98	111 / 14%	360 / 267%

Table 6.5: Maximum Memory Usage (in kB) and Overheads

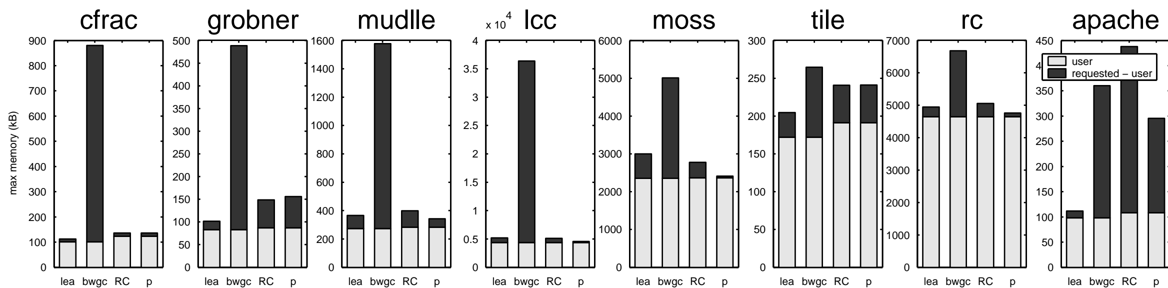


Figure 6.3: Maximum Memory Usage (in kB) and Overheads

16kB of memory. Thus the minimum amount of memory *apache* can use is approximately 400kB. This drawback of our region implementation only affects programs which use a large number of regions but only need little memory. On all benchmarks except *apache*, the Boehm-Weiser conservative collector uses most memory, in some cases (*cfrac*, *gröbner*, *mudlle*, *lcc*) by a wide margin.

RC-pairs sometimes uses substantially less memory than RC: for instance, 13% less for *moss* and 32% less for *apache*. In the case of *apache* this is due to the large number of small regions: with RC-pairs these regions take only 8kB rather than 16kB as we do not have separate pointer-free pages. Even with this improvement, the memory usage of RC-pairs is still high compared to Doug Lea's malloc/free implementation.

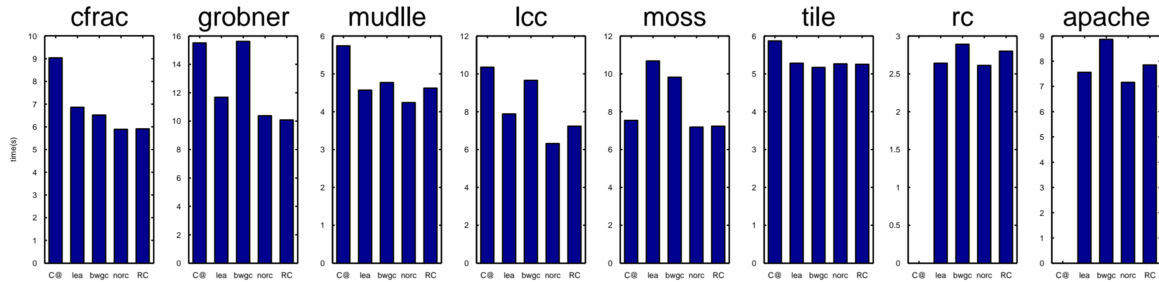


Figure 6.4: Execution time

6.7 Performance

In this section, we compare the performance of RC with our old system C@, Doug Lea's malloc/free and the Boehm-Weiser conservative garbage collector (Chapter 6.7.1) and with our alternative reference-counting schemes (Chapter 6.7.2). We also measure the overhead of reference-counting and show that compared to C@ we have reduced this overhead both in absolute time and percentage of runtime (Chapter 6.7.3).

These results, and the results in the next sections show the occasional performance anomaly. For instance, *gröbner* is faster with reference-counting enabled than without. In *rc*, disabling qualifier runtime checks increases execution time (see Figure 6.10). The code changes when compiling an application with different options (e.g., with or without reference-counting) change the code size and alignment, and the pattern of data accesses. This can lead to, for instance, changes in cache misses with small, unpredictable effects on performance. As an example, *gröbner* with reference-counting disabled has 1 million more L2 cache misses than with reference-counting enabled. At 230ns per cache miss on our test machine, this accounts for .23s of the .29s time difference between these two versions of *gröbner*.

6.7.1 Performance vs Other Allocation Techniques

The performance of our three standard allocators, of our old language C@, and of RC with reference-counting disabled ("norc") is presented in Figure 6.4. We did not port *rc* or *apache* to C@.

On these benchmarks, RC is always faster than our old system C@. This improvement is due both to a C compiler (`gcc`) that produces better code than `lcc` [24] (on which

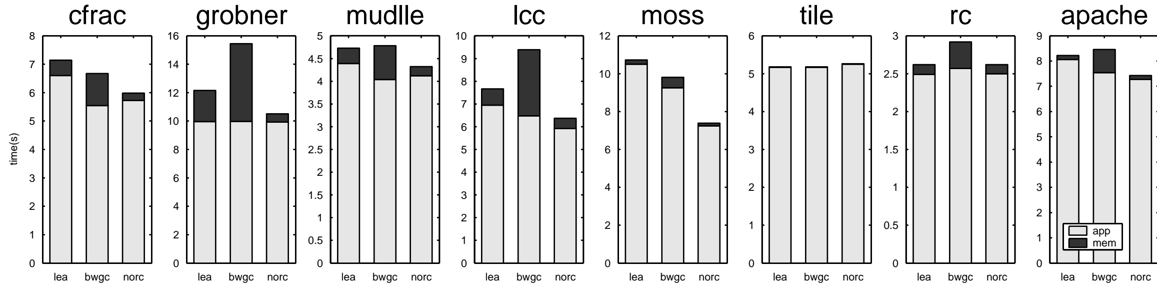


Figure 6.5: Execution time, showing time spent in memory management

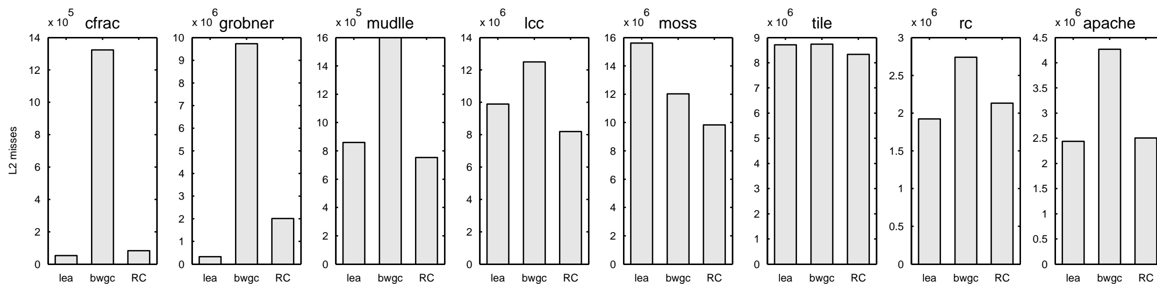


Figure 6.6: L2 cache misses

C@ is based) and to a reduced reference count overhead in RC (see Chapter 6.7.3). RC is competitive with `malloc` and `free` on all benchmarks, from 6% slower on *rc* to 48% faster on *moss*. RC is faster (up to 55% on *gröbner*) than the Boehm-Weiser conservative garbage collectors on all benchmarks except *tile* (2% slower).

We also show, in Figure 6.5, the breakdown between time spent in memory allocation ("mem") and in the rest of the application ("app"). We measure the memory management time by instrumenting the code using the UltraSPARC's cycle counters. This instrumentation slows down the code a little, and the code changes lead to small performance changes, hence the execution times are not perfectly consistent between the two figures.

These figures show that the cost of memory allocation is always lowest for regions, and always highest for conservative garbage collection. These figures also show that the Boehm-Weiser conservative garbage collector and RC gain some of their performance in *cfrac* because they can disable that application's own hand-written reference-counting code.

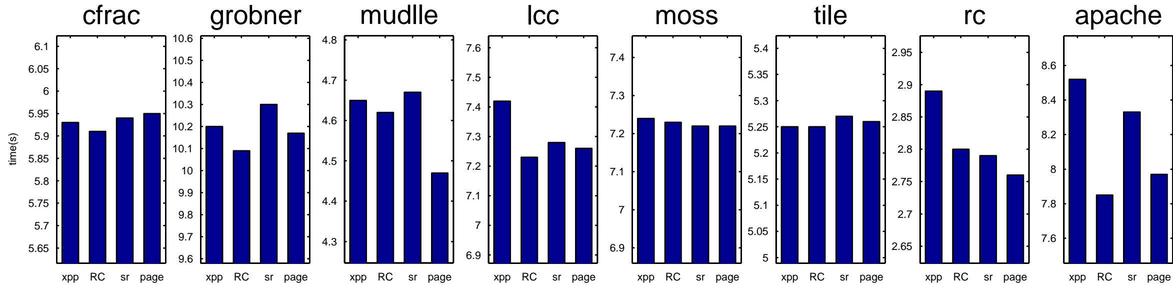


Figure 6.7: Execution time with alternative reference-counting (non-zero time origin)

This shows up as a lower application time. Similarly, in *moss* we see that RC spends less time in the application as well as in memory allocation.

In Figure 6.6, we show the number of L2 cache misses for each of the allocators and applications. This confirms that *moss* has better locality with RC. We also see smaller locality improvements for RC with *mudlle* and *lcc*, which explains some of the improved application time from Figure 6.5. We also see that applications compiled with the Boehm-Weiser have the most cache misses (except on *moss*), which is presumably due to L2 misses during garbage collection.

6.7.2 Performance of Alternative Reference-counting Implementations

Figure 6.7 shows the performance of the variations on standard reference counting presented in Chapter 4.3.6. The *excluding parent pointers* scheme is "xpp", the *include same-region references* scheme is "sr", "page" is the version of *include same-region references* that keeps a reference count per page.

The "xpp" approach is always slower (up to 8% on *apache*, but 3% or less on the other benchmarks) than the standard approach. Thus it is only worthwhile in an implementation of regions where the test that a region and its children can be deleted must be efficient. The other two approaches to reference-counting from Figure 6.7 ("sr" and "page") are either slightly faster (up to to 3%) or slightly slower (up to 6%) than the standard approach. The "page" approach takes memory for the per-page reference counts and produces a clear performance benefit for *mudlle* only.

The performance of RC-pairs (Chapter 4.3.7) is presented in Figure 6.8. The "p" column represents the *split array* approach, "psr" is the *split array* approach extended to

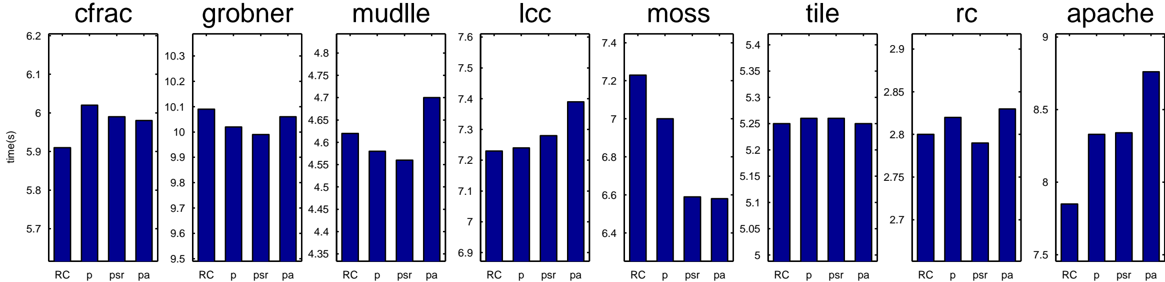


Figure 6.8: Execution time with RC-pairs (non-zero time origin)

include same-region references. The “pa” column represents the *flat array* approach.

There is little performance difference between all these pair-based approaches and the standard reference-counting scheme (2% slower to 3% faster) on all benchmarks except *moss* and *apache*. For *moss*, both “psr” and “pa” are 8% faster than *moss* compiled with reference-counting disabled, and *moss* executes very few reference-count updates per second (see the “rcupdate” column of Table 6.9 below). We thus do not attribute this speedup to improved reference-counting performance. The situation is similar for *apache*, except that the RC-pairs approach is slower rather than faster.

6.7.3 Reference-counting Overhead

Table 6.6 shows the reference counting cost for C@ and RC. This cost is presented as absolute time in seconds, and as a percentage of execution time. These figures are obtained by running our benchmarks with reference-counting enabled and disabled and subtracting the execution times. For RC, we also show time spent removing references from deleted regions (“Region unscan”). This last cost is measured using the UltraSPARC’s cycle counters. The largest reference counting overhead is for *lcc* at 12.6% of execution time. The region unscan accounts for 2% or less of execution time. This table also shows that the better performance of RC over C@ is due not only to a better base compiler (`gcc` vs `lcc`) but also to a reduction in the reference counting overhead (which is not affected by the C compiler used). As we discussed earlier, the negative reference-counting time for *gröbner* is mostly due to less L2 cache misses when reference-counting is enabled.

Name	C@		RC		Region
	(s)	(%)	(s)	(%)	unscan (s)
cfrac	0.52	5.8%	0.02	0.4%	0.01
grobner	1.13	7.3%	-0.29	-2.9%	0.02
mudlle	0.69	12.0%	0.38	8.3%	0.01
lcc	1.52	14.7%	0.91	12.6%	0.12
moss	0.04	0.6%	0.04	0.6%	0.01
tile	0.01	0.1%	-0.01	-0.2%	< 0.01
rc			0.19	6.7%	< 0.01
apache			0.69	8.8%	0.12

Table 6.6: Reference counting overhead in RC and C@

Name	Number of assigns	% safe assigns
cfrac	12	50.0%
grobner	105	80.0%
mudlle	569	88.0%
lcc	300	32.7%
moss	55	83.6%
tile	43	83.7%
rc	350	12.3%
apache	177	30.5%

Table 6.7: `sameregion`, `parentptr` and `traditional`: static statistics

6.8 Qualifiers

In this section, we will use the term *static pointer writes* to mean the number of assignment statements of pointer type in a benchmark's source code, and *runtime pointer writes* for the number of assignments executed by a benchmark on its test input. In both cases, we exclude writes to local variables. We refer to the qualifier-runtime-check-elimination system of Chapter 5.6 as the *check-elimination system.*

Table 6.7 shows the number of static pointer writes that assign a qualified type. The second column, "% safe" is the percentage of these writes for which our check-elimination system can eliminate the runtime check. Except for *rc*, we can remove a substantial fraction (30%+) of runtime checks from the generated code. For *gröbner*, *mudlle*, *moss* and *tile* we eliminate more than 80% of the checks.

Table 6.8 shows the percentage of runtime pointer writes that were of qualified types. These percentages are then broken down for each of the three qualifiers. We see that

Name	Qualified	`sameregion`	`parentptr`	`traditional`
cfrac	0.7%	0.7%	0.0%	0.0%
grobner	98.0%	98.0%	0.0%	0.0%
mudlle	81.9%	32.5%	2.1%	47.3%
lcc	73.8%	49.2%	16.4%	8.2%
moss	99.4%	5.4%	0.0%	94.0%
tile	100.0%	7.0%	0.0%	93.0%
rc	35.2%	31.7%	2.4%	1.2%
apache	38.9%	16.0%	9.3%	13.6%

Table 6.8: `sameregion`, `parentptr` and `traditional`: dynamic statistics

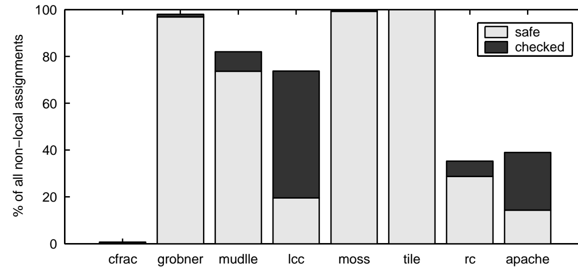


Figure 6.9: Effectiveness of qualifiers and qualifier check removal

different qualifiers are important in different benchmarks.

Figure 6.9 breaks the runtime pointer writes in our benchmarks into three categories. The "safe" category is the percentage of runtime pointers writes that were of qualified type and shown to be statically safe by our check-elimination system. These require no runtime work. The next category, "checked", is the percentage of runtime pointer writes that were of qualified type and that required a runtime check. The final category is the difference between the top of the bar and 100% is the percentage of runtime pointer writes that required a reference count update. The goal of our qualifiers is to reduce this percentage; the goal of our check-elimination system is to reduce the number of "checked" pointer writes.

Table 6.9 presents the same three categories in pointer writes per second. We call the final category "rcupdate". The overhead of reference counting is primarily dependent on the rate of reference-count operations, and secondarily on the rate of runtime checks.

From the "rcupdate" rate, we can expect that *mudlle*, *lcc* and *rc* will have the highest reference-counting overhead. This is confirmed by the reference-counting cost figures

Name	Writes	rcupdate	checked	safe
cfrac	131647/s	130711/s	737/s	198/s
grobner	641087/s	12756/s	7784/s	620545/s
mudlle	3096404/s	559623/s	257918/s	2278862/s
lcc	1373484/s	360471/s	743974/s	269039/s
moss	2242473/s	12421/s	6084/s	2223967/s
tile	36561/s	8/s	1/s	36551/s
rc	668133/s	432727/s	43279/s	192126/s
apache	94150/s	57488/s	23196/s	13465/s

Table 6.9: Reference count and runtime check rates

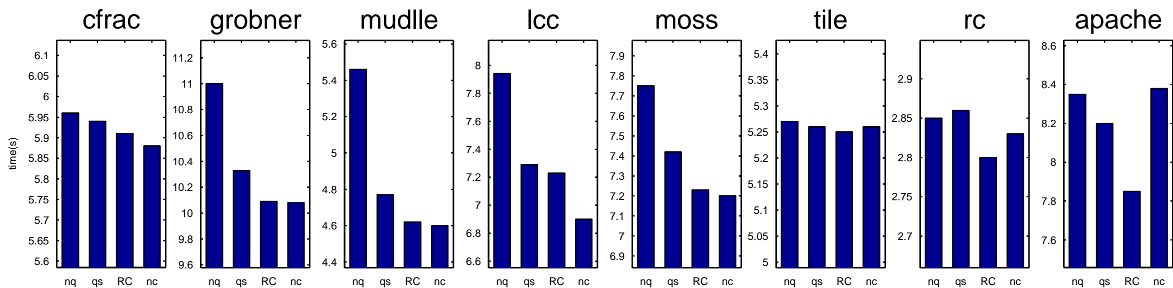


Figure 6.10: Execution time with `sameregion`, `parentptr` and `traditional` (non-zero time origin)

of Table 6.6.

The effects on execution time of `sameregion`, `parentptr` and `traditional` annotations and of our check-elimination system are shown in Figure 6.10. In the “nq” column, the annotations are ignored; in “qs” the annotations are used and checked at runtime; in “RC” the check-elimination system has removed provably safe runtime checks; in “nc” all runtime checks are (unsafely) removed (“nc” thus bounds the maximum improvement our check-elimination system can provide).

The rate of pointer-writes (“Writes” column in Table 6.9) is high for *gröbner* and *moss*, but most of these writes are of qualified pointers that require no runtime check. Without qualifiers, the reference-counting overhead of these benchmarks would be significant, as shown by the “nq” column of Figure 6.10.

From Figures 6.10 and 6.9 we conclude that our type annotations are important to the performance of *gröbner*, *mudlle*, *lcc*, *moss* and to a lesser extent *rc*. The

check-elimination system provides useful reductions in reference count overhead in *gröbner*, *mudlle*, *lcc* and *moss*. For instance, without any qualifiers the reference count overhead of *lcc* would be 20.4% instead of 12.6%, and the overhead of *mudlle* would be 22.3% instead of 8.3%. The anomalous performance results for *rc* and *apache* prevent any useful conclusion.

The programs (*gröbner*, *mudlle*, *tile*, *moss*) where the percentage of qualified assignments is highest are dominated by one or two data structures which use qualified types for their internal pointers (large integers in *gröbner*, an instruction list in *mudlle* and the input buffer used by code produced by the flex lexical analyser generator in *tile*, *moss* and *mudlle*). In *cfrac* essentially all pointer assignments are of pointers to local variables used for by-reference parameters in functions with signatures such as

```
int *pdivmod(int *u, int *v, int **qp, int **rp)
```

The effectiveness of our check-elimination system in verifying the safety assignments to `sameregion` and `traditional` pointers, and hence eliminating runtime checks, is also variable. Most checks remain in *lcc*, while virtually all are eliminated in *gröbner*, *tile* and *moss*. We illustrate here, using a simple linked list type, the kinds of code whose safety our system successfully or unsuccessfully verifies. The examples will assume the following type and local variable declarations:

```
struct rlist {
  struct rlist *sameregion next;
  struct finfo *sameregion data;
} *x, *y;
region r;
struct rlist **objects[100];
```

A simple idiom that is successfully verified is the creation of the contents of x after x itself exists:

```
x = ralloc(r, ...);
x->next = ralloc(regionof(x), ...);
```

Similar situations often arise with imperative data structures such as hash tables (as in *moss*). The large integers in *gröbner* also follow this pattern.

Our check-elimination system remains successful on fairly complex loops as long as all the variables are locals or function parameters. For instance, we can successfully verify all the assignments in Figure 1.1. A more elaborate version of this loop (involving inter-procedural analysis) is found in *moss* and is also verified.

The sameregion, parentptr and traditional annotations allow verification of some code that accesses data from the heap (or from global variables), e.g.:

```
x = ralloc(regionof(y), ...);
x->next = y->next;
```

The traditional annotations in the code generated by the flex lexical analyser generator used by *tile, moss* and *mudlle* are more complex examples (also involving inter-procedural analysis) of this.

Other constructions do not work so well. Nothing is known about objects accessed from arbitrary arrays, e.g.:

```
x = ralloc(r, ...);
x->next = objects[23];
```

The parse stack used in the code generated by the bison parser generator is like the objects array and prevents verification of the construction of parse trees in *mudlle* and *rc* (which use sameregion pointers).

Most of the benchmarks allocate memory in a region stored in a global variable, partly as an artifact of converting the programs to use regions (adding a region argument to every function would have been painful), and partly as a result of using bison generated parsers (the parsing actions only have access to the parsing state and to global variables). Our region type system does not represent the region of global variables, so verification of annotations often fails in these programs. Where possible, we changed these programs to keep regions in local variables, or used regionof to find the appropriate region in which to allocate objects.

The final case which our system does not handle well is hand-written constructors such as:

```
rlist *new_rlist(region r, rlist *next)
{
  rlist *new = ralloc(r, ...);
  new->next = next;
  return new;
}
```

To verify the assignment to next, our system must verify that at every call to new_rlist, next is null or in the same region as r. This is often not possible, e.g., in *rc* where these functions are called from a bison generated parser. It is not possible to apply a technique similar to the first idiom and replace the allocation with:

```
rlist *new = ralloc(regionof(next), ...);
```

because `next` may be `null`.[4]

6.9 Local Variables

Table 6.4 shows that writes to local variables are the most frequent, and therefore most important for reference-counting performance. Table 6.10 presents the effectiveness of various strategies for eliminating reference count operations for these writes to local variables. The results are presented as the rate of reference count operations. The *assignment* scheme of Chapter 4.3.4 is "asgn", "RC" uses the default *function* scheme, and "opt" is the *optimal* scheme. The second part of the table ("ndopt" and "nda" columns) is discussed below.

The effects of these three schemes on execution time are shown in Figure 6.11. We include a "none" bar which shows the performance of our benchmarks when references from local variables are ignored. The "none" bar is a lower-bound on how much we can improve reference-count performance for local variables. We observe that the performance of "RC" and "opt" is nearly identical, and close to "none" on all benchmarks. The *assignment* scheme is noticeably worse. Thus the *function* scheme is clearly the best solution for reference-counting local variables as it is both faster and easier to implement than the *optimal* scheme. We again notice a few anomalous results ("none" costing more than any of the other schemes, "opt" slower than "asgn" and "RC" for *rc*).

The need for the `deletes` qualifier on functions can be obviated if we assume that all functions may delete a region. However, this significantly increases the cost of reference counting local variables as shown by Figure 6.12. Here we show the time without `deletes` qualifiers for the *assignment* scheme ("nda") and the *optimal* scheme ("ndopt"). For reference, we include the standard "RC" time. The rate of local variable reference count operations for "ndopt" and "nda" are both given in Table 6.10.

These results show that the `deletes` qualifier is necessary for good performance: even with our best scheme ("ndopt"), the overhead of reference-counting can be as high as 25% (*lcc*). The highest overhead with the `deletes` qualifier is 12.6% (*lcc* again).

[4] In a new language it would be possible to have a separate `null` value for each region, which would allow this idiom to work. It is not clear whether this would be otherwise desirable.

Name	asgn	RC	opt	nda	ndopt
cfrac	476988/s	60814/s	22419/s	12920271/s	2660809/s
grobner	370487/s	14346/s	7390/s	8884627/s	1915792/s
mudlle	372861/s	196954/s	148357/s	9036609/s	2769986/s
lcc	233677/s	59634/s	39607/s	6297868/s	2288225/s
moss	1896163/s	5314/s	39/s	5957714/s	379510/s
tile	9422/s	2/s	1/s	819117/s	390507/s
rc	59235/s	2758/s	2730/s	11088300/s	1359263/s
apache	71451/s	17109/s	10258/s	570983/s	278673/s

Table 6.10: Local variable reference count operation rates

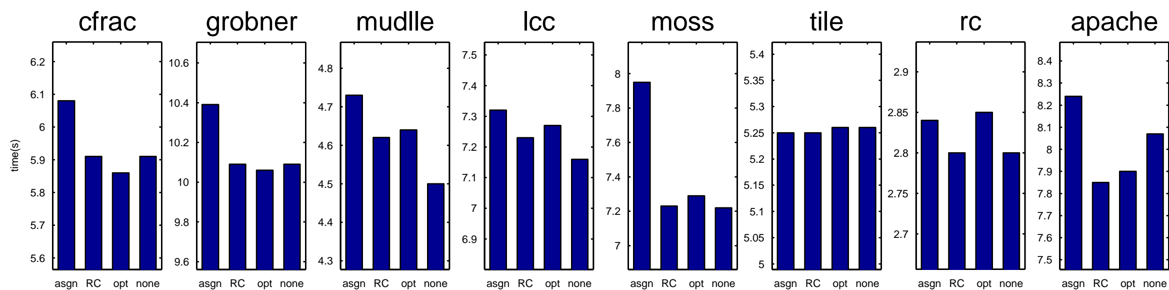


Figure 6.11: Cost of reference-counting local variables (non-zero time origin)

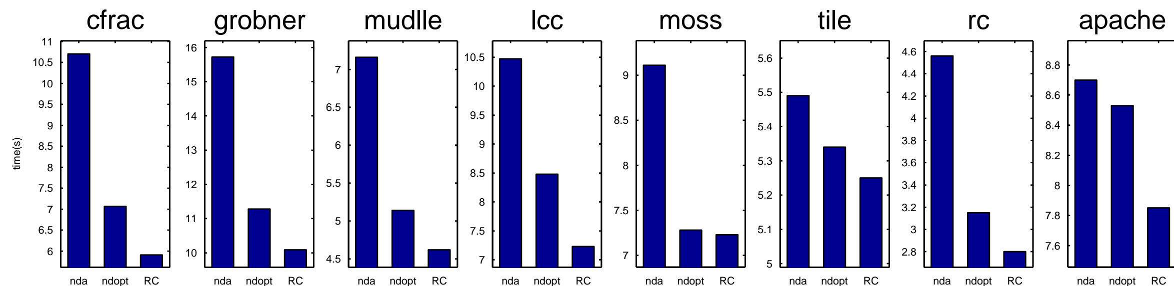


Figure 6.12: Cost of not using `deletes` qualifier (non-zero time origin)

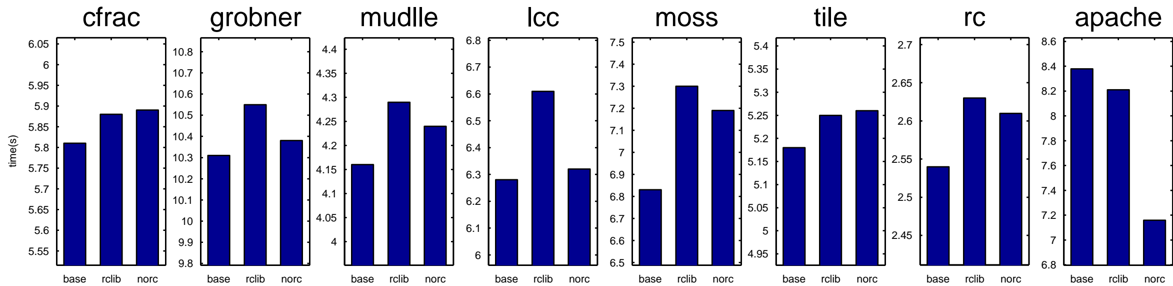


Figure 6.13: Other RC overheads (non-zero time origin)

6.10 Other Overheads

There are two further costs to using RC beyond reference counting: the C code output by RC has small changes which can affect performance; using separate pointer-free blocks increases memory usage and affects performance. Figure 6.13 shows these two costs: "base" is the runtime when each application is compiled with `gcc` (as described in Chapter 3.2.7) and a region library that does not use separate blocks for pointer-free objects; "rclib" is the runtime when compiled with `gcc` and using the standard region library; "norc" is the runtime when compiled with RC and reference-counting disabled.

The cost of using RC and it's region library, with reference-counting disabled, is 5% (on *moss*), but mostly around 1-2%. The cost in memory usage of RC can be seen in Chapter 6.6. The memory usage of unsafe regions which do not use separate blocks for pointer-free objects is the same as that of RC-pairs. Thus we can see the cost in memory of using RC by comparing the "p" and "RC" columns in Figure 6.3 and Table 6.5. As we saw in Chapter 6.6, this overhead can be significant.

6.11 Atomic Swaps

To get an idea of the cost of reference-counting for a multi-threaded system, we replaced the memory writes for unqualified pointers in our benchmarks by the UltraSPARC `swap` instruction which is atomic. This change approximates the cost of the reference-count operation of Figure 4.9. The effect of this change on execution time are shown in Figure 6.14. The highest overhead is 10% for *lcc*, and the results for *apache* are anomalous.

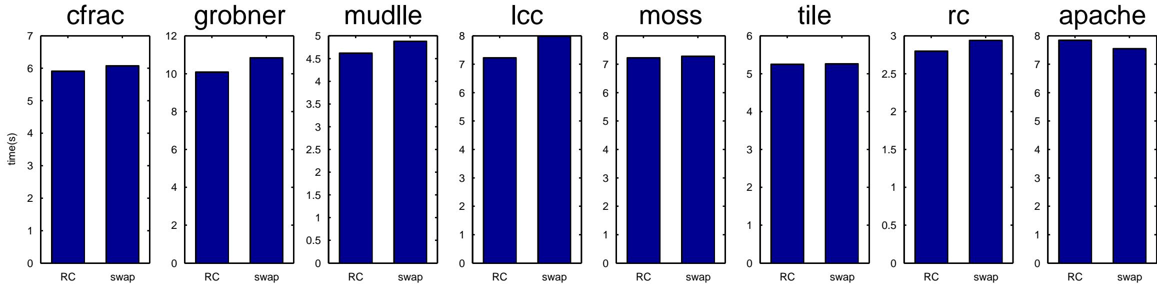


Figure 6.14: Overhead of using atomic `swap` for pointer writes

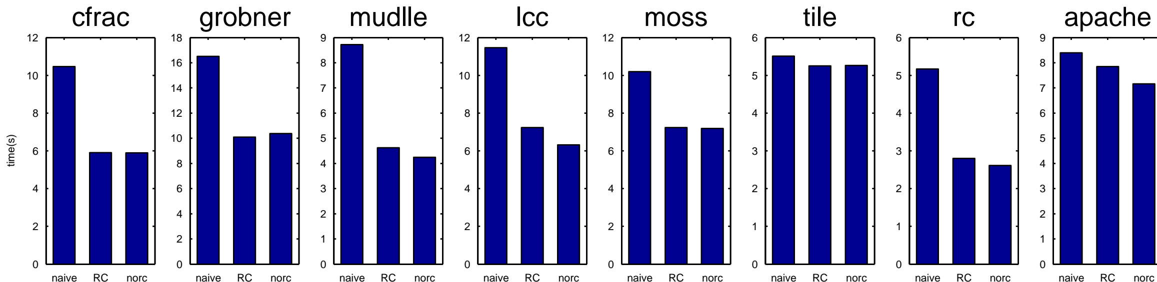


Figure 6.15: RC vs naïve reference-counting

6.12 Summary

In summary, the performance of RC is good. On all benchmarks, RC is from 6% slower to 55% faster than either Doug Lea's high-quality malloc/free implementation or the Boehm-Weiser conservative garbage collector. Memory usage is generally good, though programs like *apache* with many small regions pay a higher memory overhead. This overhead could be reduced by using pages smaller than 8kB.

On all benchmarks except *cfrac*, our type qualifiers capture a substantial fraction of non-local pointer assignments (from 35% to 99.98%). Again excluding *cfrac*, our qualifier-runtime-check-elimination system is effective at eliminating runtime checks for these qualifiers (between 37% and 99.99% of checks eliminated).

We investigated several variations on reference-counting: none had compelling performance advantages, though keeping reference-counts between pairs of regions does significantly reduce memory usage on some benchmarks (e.g., *moss* and *apache*). This improvement comes at the cost of a reference-counting scheme that does not scale to a large

number of regions n as it requires space proportional to n^2 to store the reference counts. Finally, the `deletes` qualifier is necessary to keep the cost of reference-counting for local variables low.

To conclude, Figure 6.15 shows the performance difference between a "naïve" region reference-counting implementation (no qualifiers, a reference-count operation on every pointer write) and RC. The reference-count overhead for "naïve" is as high as 51.4% (on *mudlle*), compared to a maximum of 12.6% for RC (on *lcc*).

Chapter 7

Conclusion

In summary, the performance of regions when compared to malloc/free and conservative garbage collection is good:

- On most applications with a small memory footprint (a few 100kB), unsafe regions tend to use somewhat more memory than malloc/free (from 6% less to 47% more). On one application (the *apache* web server) our region library uses substantially more memory (nearly 2.7x (184kB) more). This is due to the number of regions used simultaneously in *apache* and the large minimum size of a region (8kB) in our implementation. Our region library would benefit from better support for small regions (e.g., making the minimum region size 4kB as in our previous system C@).

- On applications with a larger memory footprint (a few megabytes), unsafe regions use less memory than malloc/free (from 19% less to 4% less).

- The conservative garbage collector uses more memory than unsafe regions or malloc/free, up to 8x more.

- RC's safe regions pay a generally small memory-usage price for safety: this overhead is 17% or less on all benchmarks except *apache*. On *apache* RC's minimum region size is 16kB which leads to a 48% increase in memory usage (to 437kB). On the benchmarks with a larger memory usage, RC's regions tend to use less memory than malloc/free (from 6% less to 2% more).

- Unsafe regions are always faster than malloc/free (up to 49%), and from 2% slower to 51% faster than conservative garbage collection.

- The performance of RC's safe regions is also good: RC is from 6% slower to 48% faster than malloc/free, and from 2% slower to 55% faster [1] than conservative garbage collection.

These results show that regions are a viable alternative to the two traditional memory management styles, explicit deallocation and garbage collection. Regions do have specific strengths and weaknesses, we discuss these further below (Chapter 7.1).

The cost of reference counting, i.e., the overhead of RC over unsafe regions, is reasonable: at most 12.6% of execution time is dedicated to maintaining reference counts. This overhead is achieved with the help of type qualifiers that allow the programmer to easily declare some aspects of the program's region structure. We generalise these qualifiers into a type system for reference-counted region systems. Analysis of RC programs rewritten with these types allows us to eliminate a substantial fraction of the runtime checks implied by the type qualifiers (from 37% to 99.99%).

We end this dissertation with a couple of thoughts about improvements to RC, and to memory management in general (Chapter 7.2).

7.1 Strengths and Weaknesses of Regions

There are situations where regions are not a good memory management model, particularly when the programmer does not know enough about the lifetime of objects at allocation-time to place them in appropriate regions. One example we encountered is a game[2] where objects are allocated and deallocated as the result of the player's actions; there is no way to place objects with similar lifetimes in a common region. We leave generalizations of explicit regions to better handle such applications as future work.

Stoutamire [49] and our *moss* benchmark show that regions can be used to improve data locality by providing a mechanism for programmers to specify which values should be colocated in memory, as well as which values should be kept separated. Neither traditional garbage collection nor malloc/free provide any such mechanism. This advantage is not specific to reference-counted regions. For instance, Stoutamire's study was based on garbage-collected regions.

[1] On this benchmark, RC is faster than unsafe regions because of reduced L2 cache misses.
[2] MUME, see http://mume.org or telnet://mume.org for details.

Reference-counted regions provide safety with predictable performance, making them suitable for use in a safe, real-time language. The cost of all operations is constant (pointer writes) or bounded in an easily predictable way (object and region allocation, region deallocation), as discussed in Chapter 4.5. Of course, malloc/free implementations can also be real-time, but without safety. And as mentioned in the introduction, real-time garbage collectors have a significant performance penalty.

In summary, we see that malloc/free and garbage collection have nearly opposite tradeoffs: malloc/free is unsafe and hard to use, but provides good control over memory and a low space overhead. Garbage collection is safe and easy to use, but provides no control over object deallocation and has a high space overhead. Regions are a third alternative which avoids most of these disadvantages: regions are safe and reasonably easy to use, have reasonable space overhead and provide better control over memory than either malloc/free or garbage collection (as regions allow some control over locality).

From these considerations, we believe regions are best suited for high-performance applications that use a large fraction of machine memory and where the lifetimes of values can be statically predicted. Regions are also useful for writing software with more predictable performance than garbage-collection-based systems.

7.2 Extensions

To be useful for all applications, we believe that our region library and reference-counting must be extended to handle regions containing very few small-to-medium-size objects. For instance, we could provide a call to allocate a single-object-region. This would help applications like the game we mentioned above where the lifetimes of individual objects are not predictable. It is not immediately clear how support for such regions can be efficiently added to RC.

Most of today's languages are wedded to a single memory management paradigm, e.g., garbage collection. Most compilers for these languages are also restricted to a single implementation of that paradigm. Thus programmers are forced by their language choice into a haphazard selection of a memory management technique (garbage collection, region-based, etc) and algorithm (e.g., stop-and-copy garbage collection). Furthermore, once made this selection is hard to change as memory management assumptions are built deeply into languages, compilers and applications. We believe that applications would benefit from

more flexibility in the choice of memory management in languages and their implementation. This would allow programmers to use the style of memory management most suited to a particular application, or to mix different styles of memory management in different parts of a large program. Additionally, the programmer could choose an implementation of memory management suited to a particular application (e.g., a region library with good support for small objects at the cost of slightly slower execution time). RC is a step in this direction, as it allows mixing of two memory management styles: explicit deallocation and regions.

Bibliography

[1] A. Aiken, M. Fahndrich, and R. Levien. Better Static Memory Management: Improving Region-based Analysis of Higher-order Languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 174–185, La Jolla, CA, June 1995.

[2] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[3] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient Detection of all Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Languages Design and Implementation*, pages 290–301, Orlando, FL, June 1994. Also Lisp Pointers VIII 3, July–September 1994.

[4] D. Bacon, C. Attanasio, H. Lee, V. Rajan, and S. Smith. Java without the Coffee Breaks: a Nonintrusive Multiprocessor Garbage Collector. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 92–103, Snowbird, UT, June 2001.

[5] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: A Dialect of Java without Data Races. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Systems, Languages and Applications (OOPSLA '00)*, pages 382–400, Minneapolis, MN, Oct. 2000.

[6] H. G. Baker. List Processing in Real-Time on a Serial Computer. *Communications of the ACM*, 21(4):280–94, 1978.

[7] D. A. Barrett and B. G. Zorn. Using Lifetime Predictors to Improve Memory Allocation Performance. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming*

Language Design and Implementation, pages 187–196, Albuquerque, New Mexico, June 1993.

[8] J. F. Bartlett. Compacting Garbage Collection with Ambiguous Roots. Technical Report 88/2, DEC Western Research Laboratory, Palo Alto, CA, Feb. 1988. Also in Lisp Pointers 1, 6 (April–June 1988), pp. 2–12.

[9] J. F. Bartlett. Mostly-Copying Garbage Collection picks up Generations and C++. Technical Note, DEC Western Research Laboratory, Palo Alto, CA, Oct. 1989.

[10] L. Birkedal, M. Tofte, and M. Vejlstrup. From Region Inference to von Neumann Machines via Region Representation Inference. In *Conference Record of POPL '96: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg Beach, FL, Jan. 1996.

[11] D. G. Bobrow. Managing Re-entrant Structures using Reference Counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980.

[12] H.-J. Boehm. Simple GC-Safe Compilation. In P. R. Wilson and B. Hayes, editors, *OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems*, Oct. 1991.

[13] H.-J. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[14] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*^{TM}. Addison-Wesley, Reading, Mass., 2000.

[15] M. V. Christiansen, F. Henglein, H. Niss, and P. Velschow. Safe Region-based Memory Management for Objects. Technical Report D-397, DIKU, Department of Computer Science, University of Copenhagen, Oct. 1998.

[16] G. E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, 3(12):655–657, Dec. 1960.

[17] Compaq Computer Corporation. Alpha 21264 Microprocessor Hardware Reference Manual, July 1999.

[18] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, chapter 27. MIT Press, Cambridge, Mass., 1990.

[19] K. Crary, D. Walker, and G. Morrisett. Typed Memory Management in a Calculus of Capabilities. In *Conference Record of POPL '99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, Texas, Jan. 1999.

[20] R. Deline and M. Fähndrich. Enforcing High-Level Protocols in Low-Level Software. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, UT, June 2001.

[21] D. Detlefs, A. Dosser, and B. Zorn. Memory allocation costs in large C and C++ programs. *Software Practice and Experience*, 24(6), 1994.

[22] L. P. Deutsch and D. G. Bobrow. An Efficient Incremental Automatic Garbage Collector. *Communications of the ACM*, 19(9):522–526, Sept. 1976.

[23] J. R. Ellis and D. L. Detlefs. Safe, efficient garbage collection for C++. In USENIX Association, editor, *Proceedings of the 1994 USENIX C++ Conference: April 11–14, 1994, Cambridge, MA*, pages 143–177, Berkeley, CA, USA, Apr. 1994. USENIX.

[24] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1995.

[25] G. Morrisett et al. Cyclone: A Next-Generation Systems Language. Information at http://www.cs.cornell.edu/Projects/cyclone/.

[26] D. Gay and A. Aiken. Memory Management with Explicit Regions. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 313–323, Montréal, Canada, June 1998.

[27] D. Gay and A. Aiken. Language Support for Regions. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 70–80, Snowbird, UT, June 2001.

[28] A. Goldberg and R. Tarjan. A New Approach to the Maximum-Flow Problem. *Journal of the ACM*, 35(4):921–940, Oct. 1988.

[29] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, June 1996.

[30] D. Grunwald and B. Zorn. CustoMalloc: Efficient, Synthesised Memory Allocators. *Software Practice and Experience*, 23:851–869, 1993.

[31] D. Grunwald, B. Zorn, and R. Henderson. Improving the Cache Locality of Memory Allocation. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Languages Design and Implementation*, pages 177–186, Albuquerque, NM, June 1993.

[32] D. R. Hanson. Fast Allocation and Deallocation of Memory Based on Object Lifetimes. *Software Practice and Experience*, 20(1):5–12, Jan. 1990.

[33] R. Hastings and B. Joyce. Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter '92 USENIX conference*, pages 125–136. USENIX Association, 1992.

[34] Y. Ichisugi and A. Yonezawa. Distributed Garbage Collection Using Group Reference Counting. In *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, Oct. 1990.

[35] M. S. Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, Univesity of Texas at Austin, Austin, TX, Dec. 1997.

[36] R. W. M. Jones and P. H. J. Kelly. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[37] S. Kaufer, R. Lopez, and S. Pratap. Saber-C: an Interpreter-based Programming Environment for the C Language. In *Proceedings of Usenix Summer Conference*, pages 161–171, July 1988.

[38] B. Liblit. Type Systems for Distributed Data Sharing. Technical Report (in preparation), EECS Department, University of California, Berkeley, 2001.

[39] A. Loginov, S. H. Yong, S. Horwitz, and T. Reps. Debugging via Run-Time Type Checking. In *Proceedings of FASE 2001: Fundamental Approcges to Softare Engineering*, 2001.

[40] J. H. McBeth. On the Reference Counter Method. *Communications of the ACM*, 6(9):575, Sept. 1963.

[41] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Code. To appear in the *Conference Record of POPL '02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*

[42] H. Patil and C. N. Fischer. Efficient Run-time Monitoring Using Shadow Processing. In *Automated and Algorithmic Debugging*, pages 119–132, 1995.

[43] N. Röjemo and C. Runciman. Lag, Drag, Void, and Use: Heap Profiling and Space-efficient Compilation Revisited. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Languages Design and Implementation*, pages 34–41, June 1996.

[44] D. T. Ross. The AED Free Storage Package. *Communications of the ACM*, 10(8):481–492, Aug. 1967.

[45] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap Profiling for Space-Efficient Java. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 104–113, Snowbird, UT, June 2001.

[46] F. Smith and G. Morrisett. Comparing Mostly-copying and Mark-Sweep Conservative Collection. In *International Symposium on Memory Management*, pages 68–78, Vancouver, Canada, Oct. 1998.

[47] J. Steffen. Adding Run–time Checking to the Portable C Compiler. *Software Practice and Experience*, 22(4):305–316, 1992.

[48] J. M. Stichnoth, G.-Y. Lueh, and M. Cierniak. Support for Garbage Collection at Every Instruction in a Java Compiler. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 118–127, Atlanta, GA, May 1999.

[49] D. Stoutamire. *Portable, Modular Expression of Locality.* PhD thesis, University of California at Berkeley, 1997.

[50] D. Stoutamire and S. Omohundro. The Sather 1.1 Specification. Technical Report TR-96-012, International Computer Science Institute, Berkeley, CA, August 1996.

[51] W. R. Stoye, T. J. W. Clarke, and A. C. Norman. Some Practical Methods for Rapid Combinator Reduction. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 159–166, Austin, TX, Aug. 1984.

[52] A. Sălcianu and M. Rinard. Pointer and Escape Analysis for Multithreaded Programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '97)*, pages 12–23, Mar. 2001.

[53] D. Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, Dec. 1996. Available as Technical Report CMU-CS-97-108.

[54] D. Tarditi. Compact Garbage Collection Tables. In *International Symposium on Memory Management*, pages 50–58, Minneapolis, MN, Oct. 2000.

[55] M. Tofte and J.-P. Talpin. Region-Based Memory Management. *Information and Computation*, 132(2):109–176, Feb. 1997.

[56] K.-P. Vo. Vmalloc: A General and Efficient Memory Allocator. *Software Practice and Experience*, 26(3):357–374, Mar. 1996.

[57] D. Walker and G. Morrisett. Alias Types for Recursive Data Structures. Technical Report TR2000-1787, Cornell University, Mar. 2000.

[58] D. Weaver and T. Germond. *The SPARC Architecture Manual: Version 9.* Prentice-Hall, New Jersey, USA, 1994.

[59] J. William S. Beebee. Region-Based Memory Management for Real-Time Java. Master's thesis, Dept. of Electrical Engineering and Computer Science, Massachusettes Institute of Technology, Sept. 2001.

[60] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, Sept. 1992. Springer-Verlag.

[61] P. R. Wilson. Uniprocessor Garbage Collection Techniques. Technical report, University of Texas, Jan. 1994. Expanded version of the IWMM92 paper.

[62] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, Sept. 1995. Springer-Verlag.

[63] D. S. Wise and D. P. Friedman. The One-Bit Reference Count. *BIT*, 17(3):351–9, 1977.

[64] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-Performance Java Dialect. In *Proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 1–14, Palo Alto, CA, Feb. 1998.

[65] G. M. Yip. Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments. Technical Report 91/8, Digital, Western Research Laboratory, June 1991. Masters Thesis — MIT, Cambridge, MA, 1991.

Appendix A

Standard C Library Compatibility

This section details the rules that must be followed when calling functions in the standard C library. Any function that is not mentioned can be called without any restrictions.

- `scanf, fscanf, sscanf, scanf`: do not use the `%p` format specifier (it writes a pointer).
- `fread`: do not read into a type containing pointers.
- `memcpy, memmove, memset`: the destination argument must not be of a type containing pointers. You can use `rarraycopy` to replace many uses of `memcpy`.
- `strtod, strtol, strtoul`: the second argument (the address of a variable into which to store a pointer to the unconverted suffix) should be `NULL`, or the following workaround should be used:

  ```
  char *s, *t;
  strtol(s, &t, base);
  ```

  should be rewritten as:

  ```
  char *s, *t;
  t = s;
  strtol(s, &t, base);
  ```

- `malloc`: as was mentioned above, do not use `malloc` to allocate memory for a type containing unqualified pointers. Use `calloc` instead, or call `memset` to clear the memory before it is used.

- `realloc`: if using `realloc` to increase the size of an object containing unqualified pointers, call `memset` to clear the extra memory allocated to the object. For instance,

```
int **x = calloc(10, sizeof(int *));
...
x = realloc(x, 20 * sizeof(int *));
memset(x + 10, 0, 10 * sizeof(int)); /* clear extra 10 'int *'s */
```

- `qsort`: although this may write pointers (depending on the type being sorted) it can be used with no restrictions as it only permutes the pointers within the object being sorted.

- `setjmp, longjmp`: these functions cannot be used.