RC User's Manual v1.0

David Gay
University of California, Berkeley
dgay@cs.berkeley.edu

July 5, 2001

1 Introduction

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'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. Region-based memory management brings structure to memory management by grouping related objects, making programs clearer and easier to write and to understand (especially when compared to using `malloc` and `free`).

RC is a dialect of C that adds safe, region-based memory management to C. There are two main advantages to using RC over an existing region-based memory management package for C (such as arenas [3]):

- RC is *safe*: 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. Deleting a region with a non-zero reference count causes a runtime error (abort). Thus RC programs cannot access freed memory. The overhead for safety is low (less than 11% on a group of realistic benchmarks [2]).

  Because the reference counts are per-region rather than per-object, RC avoids two of the common problems with reference counting:

  - The space for reference counts is only four bytes per region, rather than a significant overhead per object (often four bytes) in traditional reference counted schemes. As regions contain many objects, this is a significant space savings (or alternately a trivial space cost over non-reference counted systems).
  - Traditional reference counted systems do not deal well with cycles between objects, e.g., where object A points to object B and B points to A. Both A and B have a reference count of one even when no other pointers remain, so neither A or B can be freed. In RC, if A and B are allocated in the same region then the pointers from A to B and B to A are not external, so are not included in the region's reference count and do not prevent the region from being deleted. The same argument applies to cycles that go through more than two objects. It has been our experience that most cycles occur within a single region in region-based programs. RC also includes a function (`deleteregion_array`) for deleting several regions simultaneously while allowing cycles between those regions.

- RC incorporates static information, in the form of three new type qualifiers, that helps document and enforce some aspects of an application's memory structure. These annotations, `sameregion`, `traditional` and `parentptr`, are based on our observations of common programming patterns in large region-based applications:

```
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: An example of region-based allocation.

- 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.

The correctness of assignments to qualified pointers is enforced by runtime checks. The RC compiler includes an option (`-frc-qopt`) which can often eliminate many of these checks.

Regions have some other advantages. Region-based allocation is faster than `malloc` and `free`. Even with the overhead of safety, this advantage remains (in [2] we found that safe regions were from 6% slower to 58% faster than `malloc` and `free` on our benchmarks). Also, 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 (see [4] and [1]).

RC is compatible with existing C source code as long as a few restrictions are met (Section 2.4). RC code can also be linked with existing C object code in most circumstances (Section 2.5).

RC currently runs on SPARC-based Solaris 2.6 and Solaris 7 machines, and on x86-based Linux machines. Porting it to other platforms should not pose any particular problems. RC outputs C code and invokes `gcc` to produce object files. RC's frontend, `rcc`, takes the same options as `gcc`. For instructions on installing RC, see Section 3.

While RC is a separate compiler, it is possible to compile RC code with a regular C compiler with a few preprocessor directives and an appropriate header file and library. In this case, all runtime checking (for deleting regions and for the type qualifiers) is of course lost. For details, see Section 2.8.

The copyright on RC is as follows:

2 RC

The basic concepts in RC are:

- *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.
- *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`, memory used for global and 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 is a pointer with neither of these qualifiers.

We start by describing the region API (Section 2.2) and the type qualifiers (Section 2.3). We then describe the restrictions on C that must be followed by RC programs (Section 2.4) and under which conditions RC programs can invoke C object code (Section 2.5). Section 2.6 gives some examples of RC code and Section 2.7 discusses debugging RC programs. Finally, Section 2.8 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).

2.1 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.

RC uses `malloc` to obtain memory for region allocation. Some implementations of `malloc` have various problems (e.g., slow performance, cannot allocate all of memory, etc). These can often be solved by using Doug Lea's high quality, public domain, `malloc` and `free` implementation that is available at `ftp://g.oswego.edu/pub/misc/malloc.c` (and `malloc.h`).

2.2 Region Library

The region API is summarised in Figure 2. 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 Section 2.2.8.

`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 2 for simplicity.

This API is defined by the `regions.h` header file. This file is automatically included at the start of every C 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 Section 2.8 for details.

2.2.1 `region_main`

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

```
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 2: Region API

2.2.2 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 Section 2.2.6 for how `newregion` and `newsubregion` behave if they cannot allocate memory for the new region.

2.2.3 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 you 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. See the discussion of cyclical data structures in the Examples section (2.6) for uses of `deleteregion_array`.

See Section 2.3.1 for details on the `deletes` function qualifier.

2.2.4 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 have to provide a reference-count-adjusting function. See Section 2.4.3 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 Section 2.2.6 for behaviour if these functions cannot allocate the requested memory.

2.2.5 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 for `old` for the new object.[1]

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 Section 2.2.6 for behaviour if these functions cannot allocate the requested memory.

2.2.6 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.

[1] The current implementation does not.

2.2.7 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 Section 2.4).

2.2.8 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` (Section 2.2.4).

The `typed_ralloc`, `typed_rarrayalloc`, `typed_rarrayextend` and `typed_rarraycopy` behave identically to `ralloc`, `rarrayalloc`, `rarrayextend` and `rarraycopy`, 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))
```

`rarrayalloc`, `rarrayextend` and `rarraycopy` are defined in similar fashion.

2.3 Type Qualifiers

RC introduces new pointer type qualifiers (`sameregion`, `traditional` and `parentptr`) and one 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;
}
```

2.3.1 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.

2.3.2 Pointer-type Qualifiers

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`.

The `traditional` type qualifier (`int *traditional x`) which 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 1). 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 external references). 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 qualifier 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`.

2.4 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`.
- 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.
- 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.
- 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.

2.4.1 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 support threads in RC.

2.4.2 `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. RC programs that use `setjmp` and `longjmp` will find that the reference counts for regions may become incorrect.

2.4.3 `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*.

2.4.4 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.

2.4.5 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.

2.4.6 `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 D for details.

2.4.7 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) */
  }
}
```

2.5 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 D summarises the compatibility rules for RC when calling the standard C library.

2.6 Examples

This section shows some data and control structures for programs using regions. These are mostly distilled from the example region-based programs in the `rc/tests` directory of the RC distribution. These test programs (`cfrac`, `grobner`, `tile`, `lcc`, `mudlle`, `apache`) can all be compiled by simply typing `make` in the appropriate directory.

2.6.1 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;
}
```

2.6.2 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.

2.6.3 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.

2.6.4 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.

2.6.5 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 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);
}
```

2.7 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:

- When a region r cannot be deleted, the program is aborted, leaving a core dump for examination. The value of `realrc` in `delregion` (where the abort occurs) is r's reference count. If the value of `realrc` is less than zero, there is a bug in RC (see Appendix B on reporting bugs).
- The region of a pointer p can be found in `gdb` with the following expression:
  `p __rcregionmap[(long)`p` >> 13]`
- If the core dump does not help, recompile your application with the `-frc-nolocals` option. This disables reference counting local variables whose address is not taken. 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.
- The following functions will scan an area of memory for a pointer to a particular region r:

  ```
  void findgrefs(region r);
  void findrrefs(region r, region from);
  void findrefs(region r, void *from, void *to);
  ```

  `findgrefs` checks the global variable area, `findrrefs` checks all the memory allocated to region `from`, and `findrefs` checks the memory from `from` to `to`. These functions can be called from inside `gdb` (as long as the application is still running, as is standard for calling any function in `gdb`).

  These functions print out `info symbol 0x...` for every address at which they find something that looks like a pointer to an object in region r. For global variables, pasting this line to the `gdb` command prompt will print the corresponding global variable. For `findrrefs` and `findrefs` you will still have to find the corresponding object in your program. It is possible (though not extremely likely) that a section of memory containing non-pointer data (integers, strings, etc) may look like a pointer to `findrefs`, `findrrefs` or `findgrefs`.
- Finally, if all else fails, you can use `gdb`'s `display` function to track a region's reference count as you step through a program. This works best when used in conjunction with the `-frc-nolocals` option as there are then less changes to a region's reference count. For instance:

  ```
  (gdb) n
  765       compile_block = newregion();
  (gdb) p compile_block
  $1 = 0x3ba010
  (gdb) display $1->rc
  1: $1->rc = 1
  (gdb) n
  770       component_true = new_component(compile_block, c_constant,
  1: $1->rc = 2
  ```

  It is best to print the region and use the resulting $$n$ variable in the `display` expression as the `display` will then remain valid if the variable is modified or goes out of scope (e.g., when stepping into another function).

2.8 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` file from the `rc/libcompat` directory must 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 `rcc`.
- 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 the `rc` directory for details.

3 Installing RC

RC has been tested on Solaris 2.6 and Solaris 7 machines with a SPARC processor, and on x86 machines running Linux. It should be easy to port to other machines and operating systems. RC's compiler is designed to integrate with `gcc` and therefore requires that a `gcc` installation be available. RC is designed to work with any C compiler, but switching C compilers will require changes in how RC integrates into the compilation process, and small changes in the generated code and region library. RC's frontend (`rcc`) is written in `perl` and therefore requires a `perl` installation. The `Makefile` for RC requires GNU `make`.

Assuming GNU's `make`, `gcc` and `perl` are available, installation of RC is straightforward:

- Unpack the distribution archive, and change into the created `rc` directory.
- Run `./configure --prefix=`*`installation directory`*. The default installation directory is `/usr/local`.
- Compile and install RC with `make install`.
- Assuming that *installation directory*`/bin` is in your path, `rcc` will now invoke RC's frontend.

Testing the installation (this assumes that `rcc` is in your path):

- Change to directory `rc/tests/cfrac`.
- Execute `make`
- When compilation finishes, execute `./cfrac 23551`.
- The answer should be `23551 = 11 * 2141`

4 Compiling RC Programs

RC's frontend, `rcc`, can be used as a direct replacement for `gcc`. It accepts all of `gcc`'s flags, plus a few extra options for RC-specific features.

RC programs use the same `.c` suffix as C programs, this reflects the fact that most C code can be compiled directly with RC, and that RC code can be compiled with a C compiler with appropriate options (see Section 2.8).

There is one minor restriction when compiling with `rcc` rather than `gcc`: `rcc` does not support compiling RC and non-RC code from in one invocation of `rcc`. The command

```
rcc -c foo.c bar.s
```

produces the message `rcc does not support mixed C and other language compilation.  Sorry.` The more usual sequence of commands

```
rcc -c foo.c
rcc -c bar.s
```

is accepted with no restrictions.

4.1 RC flags

`rcc` accepts the following options:

- `-frc-local-opt`: use the optimal algorithm for placing reference count operations for local variables. This produces slightly faster executables, but the compile-time penalty is sometimes significant.
- `-frc-nolocals`: disable reference counting references from local variables whose address is not taken. This is mostly useful when debugging. See the discussion in the Debugging section above.
- `-frc-norc`: disable reference counting completely. This is effectively the same as compiling with a regular C compiler, but the syntax extensions of RC are accepted transparently.
- `-frc-noqchecks`: disable the runtime checks for assignments to qualified pointers.
- `-frc-nodeletes`: remove the requirement for placing the `deletes` keyword on functions that may delete regions. This increases the cost of reference counting.
- `-frc-qopt` or `-frc-qopt-perfn`: optimise the runtime checks for assignments to qualified pointers. With `-frc-qopt` this optimisation is performed by analysing a whole file (slower, better results), with `-frc-qopt-perfn` the analysis is performed on each function individually (faster, less checks removed).

Note that all files in an application should be given the same `-frc-...` options, and that these options should also be included on the invocation of `rcc` that links the application. For instance, the following will produce an incorrect application:

```
rcc -c file1.c
rcc -c -frc-nodeletes file2.c  ; ERROR: different -frc-... options
```

as will

```
rcc -c file1.c
rcc -c file2.c
rcc -o app -frc-norc file1.o file2.o ; ERROR: files not compiled with -frc-norc
```

A Copyright

B Known Bugs and Bug Reports

The following known problems exist with RC:

- Copying `struct`s or `union`s does not perform the runtime checks for qualified pointers.
- Function pointers in local static initialisers may cause bugs with the `-frc-qopt` option.
- Complex initialisers (for arrays, `struct`s or `union`s) are not type checked in `rcc` (but are in subsequent C compilation, but this won't catch errors related only to RC's type qualifiers).
- Some warnings may be generated twice (once from `rcc`, then again when the code is passed through `gcc`).
- The `-Wparentheses` option does not work (no warnings are produced).
- The `rarrayextend`, `rstrextend` and `rstrextend0` functions always allocate a new object and never reuse the old memory. This will be fixed in a subsequent release.

Please email any bug reports to David Gay, dgay@cs.berkeley.edu. If possible, please include a short program that exhibits the bug, and report the version of `gcc` and details on your system (processor, OS, etc). The latest version of RC can be found at `http://www.cs.berkeley.edu/~dgay/rc`.

C Future Enhancements

The following enhancements will appear in a subsequent release:

- Some functions to return the child regions or the parent region of a region r.
- *Permanent* regions, a generalisation of the traditional region in which region allocation will be possible.
- The system will retry memory allocations after out-of-memory handlers return in some as-yet-to-be-determined fashion, rather than simply aborting the program.

D 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.

References

[1] D. Gay and A. Aiken. Memory Management with Explicit Regions. In *Proceedings of the ACM SIG-PLAN '98 Conference on Programming Language Design and Implementation*, pages 313–323, Montréal, Canada, June 1998.

[2] 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, Utah, June 2001.

[3] D. R. Hanson. Fast Allocation and Deallocation of Memory Based on Object Lifetimes. *Software Practice and Experience*, 20(1):5–12, Jan. 1990.

[4] D. Stoutamire. *Portable, Modular Expression of Locality.* PhD thesis, University of California at Berkeley, 1997.