# Language Support for Regions

David Gay and Alex Aiken*
EECS Department
University of California, Berkeley
{dgay,aiken}@cs.berkeley.edu

## ABSTRACT

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. Using type annotations that make the structure of a program's regions more explicit, we reduce the overhead of reference counting from a maximum of 27% to a maximum of 11% on a suite of realistic 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.tar.gz`.

## 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.

Traditional region-based systems such as *arenas* [8] are unsafe: deleting a region may leave dangling pointers that are subsequently accessed. In this paper we present *RC*, a dialect of C with regions that guarantees 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. Section 3 gives a short introduction to RC. RC compiles to C, so can be used with any C compiler on any platform. While our results are presented in the context of a C dialect, our techniques can be used to add support for regions to other languages (Section 3).

```
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.**

We believe that region-based programming has several advantages over other memory management techniques. First, it 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`). Second, regions provide safety with good performance: on our benchmarks, regions with reference counting are from 7% slower to 58% faster than the same programs using malloc/free or the Boehm-Weiser conservative garbage collector, and the overhead of reference counting is at most 11% of execution time. Furthermore, Stoutamire [11] and our earlier study of regions [6] 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.

Our paper makes three contributions. First, RC is a realistic proposal for adding language support for regions to mainstream languages. We have used RC in large applications and found programming with regions both straightforward and productive.

Our second contribution, and the major change in RC over our previous system C@ [6], is the addition of static information in the form of three 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

*This work was supported in part by NASA Contract No. NAG2-1210, NSF CCR-0085949, NSF infrastructure grant ACI-9619020 and DARPA contract F30602-95-C-0136. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

*PLDI* '01 Snowbird, Utah USA
Copyright 2001 ACM 0-89791-88-6/97/05 ..$5.00

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 39% and 99.98% of pointer assignments executed were to annotated types. The correctness of assignments to annotated pointers is enforced by runtime checks (Section 3.2).

Our third contribution is 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 (Section 4).

The combination of type annotations and static elimination of runtime checks reduces the largest reference counting overhead from 27% to 11% of runtime. On two benchmarks, more than 90% of the reference counting cost is eliminated, on three other benchmarks between 27% and 75% of the reference counting cost is removed. Two of the three other benchmarks already have very low reference counting overhead (less than 1% of total execution time). For a full discussion of the results, see Section 5.

## 2. RELATED WORK

The statically checked region-based systems proposed by Tofte and Talpin [13] and Crary, Walker and Morrisett [3] include type systems that are similar to the one used in rlang: all these systems annotate pointers with a name for the targeted region. Walker and Morrisett [15] have enriched these region type systems with a form of existentially quantified regions. Deline and Fähndrich [4] have designed a programming language, *Vault*, that incorporates Walker and Morrisett's type system and allows static verification of region and other resource usage. There are two important differences between the type system of Walker and Morrisett and rlang's:

- 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`.

In [6] 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. 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 [5] in [6]) and incorporates some static information about a program's region structure.

Regions were used for decades in practice, well before the current research interest. Ross [10] 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 [14] 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 [8] *arenas* are freed all at once. Barrett and Zorn [1] 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.

Stoutamire [11] adds *zones*, which are garbage-collected regions, to Sather [12] 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 [2] 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 in [9] for use in distributed systems. Neither of these papers includes any performance measurements.

Surveys of memory management can be found in [16] for garbage collection and [17] for explicit allocation and deallocation.

```
typedef struct region *region;

region newregion(void);
region newsubregion(region r);
void deleteregion(region r);
/* ralloc, rarrayalloc are not functions (they
    take a type as last argument) */
type *ralloc(region r, type);
type *rarrayalloc(region r, size_t n, type);
region regionof(void *x);
```

**Figure 2: Region API**

# 3. RC

From the programmer's point of view, RC is essentially C with a region library (Figure 2) and a few type annotations (Section 3.2). RC programs can reuse existing C code, and even in most cases object code (this is important as the C runtime library is not always available in source form), as long as the restrictions detailed in Section 3.1 are met. An overview of the implementation of RC is given in Section 3.3.

We stress that the ideas in RC are portable to other languages. In addition, different notions of memory safety can be realised in the RC framework. The option developed in this paper has `deleteregion` abort the program when there remain references to the region. A second option is to simply return a failure code from `deleteregion` when its use would be unsafe. A third choice is implicit region deletion: at various times, e.g., when memory is running out, the system deallocates any regions whose reference count has dropped to zero. This last option provides memory safety semantics similar to traditional garbage collection.

We choose to make `deleteregion` explicit as this makes RC a dialect of C: if the type annotations are removed (e.g., via the C preprocessor) and a region library is provided, any RC program can be compiled with a regular C compiler. Of course, `deleteregion` is then unsafe.

RC's reference counting scheme, which keeps a count of external references into each region, has two advantages over traditional reference counting: the space overhead is low (one integer per region) and cyclic data structures can be used transparently as long as the cycles are contained within a single region. When a cycle crosses regions, it is the programmer's responsibility to break it before attempting to delete any of the regions involved in the cycle.

## 3.1 RC Restrictions

RC imposes a number of restrictions on some unsafe, low-level features of the C language. None of these would be necessary if regions were added to, e.g., Java:

- Integers that do not correspond to valid pointers may not be cast to a pointer type.
- Region pointers must always be updated explicitly:
  - Copying objects containing region pointers byte-by-byte with `char *` pointers is not allowed.
  - Unions containing pointers are only partially supported: RC must be able to track these pointers, so the programmer must provide functions to copy such unions in a type-safe way (i.e., by copying pointers from within the union iff these pointers are valid).
- Object code compiled by compilers other than RC can be used so long as this code does not write or overwrite any region pointers in the heap or in global variables. For example, `printf` can be used with no problems while `memcpy` and `memset` functions can only be used on objects containing no pointers.
- RC does not currently support `setjmp` and `longjmp`. This restriction could be lifted in an implementation where reference-counting is integrated into the compiler.

Our current implementation does not detect these situations.

## 3.2 Type Annotations

Our previous version of C-with-regions, C@ [6] made a type distinction between pointers to objects in regions and traditional C pointers (to the stack, global data, or `malloc` heap). Any conversion between these two kinds of pointers was potentially unsafe and could lead to incorrect behaviour. We now find this approach too cumbersome: existing code cannot be used with regions without modification, and some code must be provided in both traditional pointer and region pointer versions. RC has one basic kind of pointer that can hold both region and traditional pointers. Traditional C pointers are viewed as pointers to a distinguished "traditional region" which contains the code, stack, global data and `malloc` heap.

Examination of our benchmarks shows that some pointers still 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). For example, in our `moss` benchmark 94% of runtime pointer assignments are of traditional pointers in code produced by the `flex` lexical analyser generator. RC has a `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`.

In our `lcc` benchmark, 56% of runtime pointer assignments write a pointer to an object in region $r$ into another object in region $r$. Similar percentages are found in several other benchmarks. This, combined with examination of our benchmarks' source code, led us to add a `sameregion` type qualifier for pointers that stay within the same region or are null. The `next` and `data` fields of Figure 1 are examples of this annotation. We have found that `sameregion` equates well with "part of the same data structure": data structures that are freed all at once can be allocated within the same region, and therefore all their internal pointers can be declared `sameregion`. 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.

The Apache web server uses subregions to handle subrequests created to handle an original request. On our test input, 10% of runtime pointer assignments in Apache are to pointers that always stay within the same region or point to a parent region. We capture these pointers with a `parentptr` type qualifier. Subregions and `parentptr` pointers are found in several other benchmarks. 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.

A final type qualifier, `deletes`, is used on function types to indicate functions that may delete regions (see Section 3.3.2).

## 3.3 Implementation

The implementation of RC is based on an RC-to-C compiler and a runtime library that together provide the region API of Figure 2 (Section 3.3.1) and maintain the region's reference counts (Section 3.3.2). By compiling to C, we are able to use RC with any C compiler, rather than being tied to a particular compiler as in our previous system C@ [6].

### *3.3.1 Region Library*

The implementation of the region library is similar to the one in [6]: a region, defined by

```
struct region {
  int rc, id, nextid;
  struct allocator normal;
  struct allocator pointerfree;
};
```

is composed of a reference count and two allocators, the `pointerfree` allocator for objects containing only non-pointer data or annotated pointers, and the `normal` allocator for all other objects. This distinction reduces the cost of updating reference counts when deleting a region (see below). The `id` and `nextid` fields are described with the `parentptr` runtime check implementation below.

Allocation of memory to regions is in *blocks* whose size is a multiple of the page size (currently 8KB[1]) and which are aligned on a page-size boundary. Each page belongs to one region only and the library maintains a map from pages to regions. This allows efficient implementation of the `regionof` function and of reference counting.

### *3.3.2 Maintaining Reference Counts*

Reference count updates may occur on any pointer assignment[2] and when a region is deleted. 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 3(a)) takes 23 SPARC instructions, so maintaining reference counts is potentially very expensive. RC reduces this cost through use of the type annotations of Section 3.2 and by eliminating most reference count operations for local variables.

Assignments to `sameregion`, `parentptr` and `traditional` pointers only need one of the runtime checks of Figure 3(b) rather than a reference count update. These checks take between 6 and 14 SPARC instructions and do not need to read the value being overwritten. Section 4.3 discusses how we eliminate a significant fraction of these runtime checks. The runtime check for `parentptr` relies on a depth-first numbering of the region hierarchy stored in the `id` and `nextid` fields of each region. Our current implementation updates this numbering every time a region is created, but this could easily be replaced by a more efficient scheme.

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

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

**Figure 3: Reference counting and annotation checking**

The references from local variables need only be included in a region's reference count when calling `deleteregion`. As we are compiling RC to C we cannot use C@'s [6] approach and have `deleteregion` scan the stack for pointers to regions from local variables. Instead, when calling a function that may delete a region, RC increments the reference count of all regions referred to by live local variables and decrements these reference counts on return. This approach works well as calls to functions that may delete regions are much rarer than pointer assignments to local variables. RC thus 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 (so must also be included in any prototype for the function). The compiler requires that any function that calls a function qualified with `deletes` be itself qualified with `deletes`.

We investigated a more elaborate (and optimal) scheme for placing reference count increments and decrements for local variables, but found it had little benefit (and sometimes a substantial compile-time cost) over the approach outlined above. Details can be found in [7].

When deleting a region, references from the now dead region to other regions are removed by scanning all the objects in the region, using type information recorded when

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

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

$$
\begin{array}{ll}
\tau = \mu @ \sigma \mid \exists\rho/\delta.\tau & \text{(types)} \\
\mu = \texttt{region} \mid T[\sigma_1, \ldots, \sigma_m] & \text{(base types)} \\
\sigma = \rho \mid R \mid \top & \text{(region expressions)} \\
\delta = \sigma \leq \sigma | \neg\delta | \delta \vee \delta | (\delta) & \text{(region properties)} \\
\multicolumn{2}{l}{\texttt{struct}\ T[\rho_1, \ldots, \rho_m]\{\mathit{field}_1 : \tau_1, \ldots, \mathit{field}_n : \tau_n\}} \\
 & \text{(structure declarations)}
\end{array}
$$

$T$: type names, $\rho$: abstract regions, $R$: region constants

**Figure 4: Region type language**

the objects were allocated. The pages of the `pointerfree` allocator need not be scanned as they do not contain pointers to other regions. We have found that the cost of this scan operation remains reasonable (2% or less on all benchmarks). However, we plan to investigate ways of reducing this cost further.

## 4. A REGION TYPE SYSTEM

The type annotations of Section 3.2 are a simple way for the user to specify types from a more general region type language (Section 4.1) which partially specifies the regions of pointers. This type language is used in a simple region-based language *rlang* (Section 4.2). By translating RC programs into rlang, our compiler for RC can check the correctness of some annotations and reduce the reference count overheads in some programs (Section 4.3).

### 4.1 Region 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 $A_H$: $r' \leq r$ 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 $r \leq \top$ for all regions $r$.

Figure 4 gives types for pointers reflecting this heap structure: there are pointers to regions (`region`), and pointers to named records with named fields. Each type is annotated with a *region expression* $\sigma$ which specifies the region to which values of that type point ($\ldots @\sigma$). Function and non-pointer types could be added easily to both the heap model and type language.

Region expressions are either *abstract regions* $\rho$ 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 $\rho$ is a region in $A_H$ that respects the property specified by boolean expression $\delta$. For instance, the type $\exists\rho/\top \leq \top.T[\ldots]@\rho$ 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 $\top \leq \top$ 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 Section 4.2).

$$
\begin{array}{l}
\text{program} ::= \text{fn}^* \\[4pt]
\begin{array}{rcl}
\text{fn} & ::= & f[\rho_1, \ldots, \rho_m]/\delta(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau, \delta' \\
 & & \texttt{is}\ [\rho'_1, \ldots, \rho'_p]x'_1 : \tau'_1, \ldots, x'_q : \tau'_q, s, x \\[4pt]
s & ::= & s_1; s_2 \\
 & | & \texttt{if}\ x\ s_1\ s_2 \\
 & | & \texttt{while}\ x\ s \\
 & | & x_0 = x_1 \\
 & | & x_0 = f[\sigma_1, \ldots, \sigma_m](x_1, \ldots, x_n) \\
 & | & x_0 = x_1.field \\
 & | & x_1.field = x_2 \\
 & | & x_0 = \texttt{null} \\
 & | & x_0 = \texttt{new}\ T[\sigma_1, \ldots, \sigma_m](x_1, \ldots, x_n)@x' \\
 & | & \texttt{chk}\ \delta
\end{array}
\end{array}
$$

Some predefined functions:
$\texttt{newregion}[]/\texttt{true}() : \exists\rho.\texttt{region}@\rho, \texttt{true}$
$\texttt{newsubregion}[\rho]/\texttt{true}() : \exists\rho'/\rho' \leq \rho.\texttt{region}@\rho', \texttt{true}$
$\texttt{deleteregion}[\rho]/\texttt{true}(r : \texttt{region}@\rho) : \texttt{region}@\top, \texttt{true}$
$\texttt{regionof\_}T[\rho, \rho_1, \ldots]/\texttt{true}(x : T[\rho_1, \ldots]@\rho) : \texttt{region}@\rho, \texttt{true}$

**Figure 5: *rlang*, a simple imperative language with regions**

If two values point to the same abstract region $\rho$ 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

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

$x$ is a list stored in region $\rho$ of arbitrary regions. Without the existentially quantified type 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).

### 4.2 Region Type Checking in rlang

We chose to define *rlang* (Figure 5) 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 $\delta$ that expresses requirements that must hold between the abstract region parameters at all calls to $f$. The output property $\delta'$ expresses properties that are known to hold between the abstract region parameters when $f$ returns.

The `chk` $\delta$ statement is a runtime check that the property specified by $\delta$ holds. If the check fails, the program is aborted. Instantiation and generalisation of existential types is implicit in the rules for assignment (Figure 6) rather than being done by explicit instantiate and generalise op-

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

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

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

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

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

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

$$\frac{\delta, L \vdash s_1, \delta' \quad \delta', L_{s_2} \vdash s_2, \delta''}{\delta, L \vdash s_1; s_2, \delta''} \qquad \frac{\delta, L_{s_1} \vdash s_1, \delta' \quad \delta, L_{s_2} \vdash s_2, \delta''}{\delta, L \vdash \texttt{if}\; x\; s_1\; s_2, \delta' \vee \delta''} \qquad \frac{\delta \vee \delta'', L_s \vdash s, \delta''}{\delta, L \vdash \texttt{while}\; x\; s, \delta \vee \delta''}$$

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

## Assignment

$$\frac{\begin{array}{c} \sigma' \in L \cup C_R \qquad \mathrm{fv}(\delta'[\sigma'/\rho]) \subseteq L \\ \delta \Rightarrow \delta'[\sigma'/\rho] \quad \delta, L \vdash \tau[\sigma'/\rho] \leftarrow \tau', \delta'', L' \end{array}}{\delta, L \vdash \exists\rho/\delta'.\tau \leftarrow \tau', \delta'', L'} \;(\exists\text{gen.}) \qquad \frac{\begin{array}{c} \rho \notin L \qquad \delta \Rightarrow \delta'' \qquad \mathrm{fv}(\delta'') \subseteq L \\ \delta'' \wedge \delta'[\rho/\rho'], L \cup \{\rho\} \vdash \tau \leftarrow \tau'[\rho/\rho'], \delta''', L' \end{array}}{\delta, L \vdash \tau \leftarrow \exists\rho'/\delta'.\tau', \delta''', L'} \;(\exists\text{inst.})$$

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

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

**Figure 6: Region Type Checking**

erations. 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 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 $x.\mathit{field} : \tau$ asserts that $x$, or a field of $x$, has type $\tau$. The set of free abstract regions of a boolean expression $\delta$ is $\mathrm{fv}(\delta)$.

Type checking for rlang (Figure 6) 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 $\delta$ describes the properties of $f$'s abstract regions before executing $s$, the output property $\delta'$ the properties of these abstract regions after executing $s$. Instead of an explicit binding construct for abstract regions, assignments may bind any abstract region of the assignment target which is not in the *live abstract region* set $L_s$. This set $L_s$ contains $f$'s abstract region parameters and the abstract regions used in any live variable's type. The output property $\delta''$ of $f$ describes properties of $f$'s abstract region parameters that hold when the function returns. If these parameters could be rebound, then $\delta''$ would describe properties of some arbitrary regions used inside $f$ rather than of $f$'s abstract region parameters. We assume that $L_s$ is precomputed for each statement $s$ using a standard liveness analysis.

The judgments $\delta, L \vdash \tau_1 \leftarrow \tau_2, \delta', L'$ of Figure 6 check that a value of type $\tau_2$ is assignable to a location of type $\tau_1$. These judgments take an input property $\delta$ and live abstract region set $L$ and produce an updated (as a result of binding abstract regions) output property $\delta'$ and live abstract region set $L'$. The ($\exists$gen.) rule allows assignment as long as $\tau_2$ can be existentially quantified to match $\tau_1$. The ($\exists$inst.) rule allows instantiation of an existentially quantified region into a dead abstract region $\rho$, and updates $\delta$ and $L$ to reflect $\rho$'s new properties. It is possible that $\delta$ described properties of the old value of $\rho$, these properties are removed by using a new property $\delta''$, implied by $\delta$, that does not have $\rho$ amongst its free variables. Base types are assignable if their region expressions match. Two region expressions match if $\delta$ implies they are equal or if the abstract region $\rho$ of the assignment target is dead. In this last case $\delta$ is updated to reflect $\rho$'s new properties.

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.

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.

We have proved the soundness of our type system, based on a natural operational semantics for rlang and a definition of consistency of typed values with the heap. The details and proof are in [7].

## 4.3 Translating RC to the Region Type System

There are severals ways RC can be translated to rlang. For instance, one could apply a "region inference"-like algorithm [13] 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 `sameregion`, `parentptr` and `traditional` annotations. 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 $\rho$ 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 $\rho$, 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 $\exists\rho'/\rho \leq \rho'.T[\rho']@\rho'$ For example,

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

  becomes

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

  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 $\rho_x$. 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]

- 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` $\rho_{x_1} \leq \rho_{x_2}$ 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 Section 3.2: 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 arguments 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`):

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

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

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: $\sigma = \top$, $\sigma \neq \top$, $\sigma_1 \leq \sigma_2$, $\sigma_1 = \top \vee \sigma_1 = \sigma_2$ 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

[3] This last restriction is due to the rules for handling liveness in Figure 6.

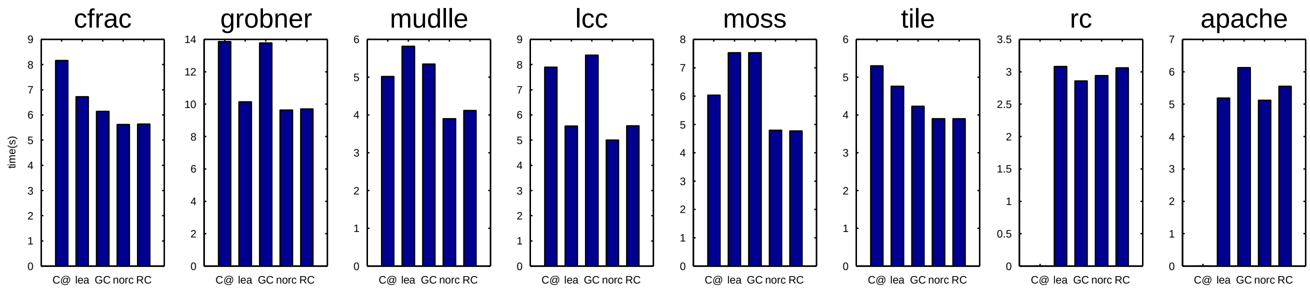

**Figure 7: Execution time**

| Name | Lines | Number allocs | Mem alloc (kB) | Max use (kB) |
|---|---|---|---|---|
| cfrac | 4203 | 3812425 | 56076 | 102 |
| gröbner | 3219 | 5971710 | 312992 | 474 |
| mudlle | 5078 | 1594372 | 22354 | 210 |
| lcc | 12430 | 1002210 | 55637 | 4121 |
| moss | 2675 | 553986 | 6312 | 2185 |
| tile | 926 | 10459 | 309 | 153 |
| rc | 22823 | 81093 | 4714 | 4214 |
| apache | 62289 | 164296 | 30806 | 78 |

**Table 1: Benchmark characteristics.**

| Name | C@ | | RC | | Region |
|---|---|---|---|---|---|
| | (s) | (%) | (s) | (%) | unscan (s) |
| cfrac | 0.48 | 6% | 0.02 | 0.4% | .01 |
| gröbner | 0.88 | 7% | 0.07 | 0.7% | .02 |
| mudlle | 0.56 | 13% | 0.23 | 6% | .01 |
| lcc | 1.14 | 17% | 0.56 | 11% | .07 |
| moss | 0.11 | 2% | -0.02 | <0% | <.01 |
| tile | 0.02 | 0.4% | 0.00 | 0% | <.01 |
| rc | | | 0.12 | 4% | <.01 |
| apache | | | 0.43 | 8% | .10 |

**Table 2: Reference counting overhead in RC and C@**

`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 largest 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 Section 5.2.

## 5. RESULTS

We use a set of eight small to large C benchmarks to analyse the performance of RC: `cfrac` and `gröbner` perform numeric computations using large integers, `mudlle`, `lcc` and `rc` are compilers, `tile` and `moss` process text and `apache` is a web server. Half of these programs (`mudlle`, `lcc`, `rc`, `apache`) were already region-based (using simple region libraries with no safety guarantees); the other half were converted to use regions (details can be found in [6]). The `cfrac` benchmark was written with explicit reference-counting; this hand-written reference counting is disabled when running `cfrac` with RC and conservative garbage collection. Table 1 reports the benchmarks' sizes (in lines of code) and summarises their memory allocation behaviour: "number allocs" is the number of objects allocated, "mem alloc" is the total amount of memory allocated during execution of the program, "max use" is the maximum amount of memory in use at any time.

### 5.1 Performance

We compared the performance of RC with our old system, C@, with conventional malloc/free-based memory management and with conservative garbage collection. Measurements were made on a Sun Ultra 10 with a 333Mhz UltraSparc II processor, a 2MB L2 cache and 256MB of memory.

Figure 7 reports elapsed time (from the best of five runs) for each benchmark for five compiler/allocator combinations: "C@" is our previous region compiler (we did not convert `rc` or `apache` to run under C@ as this would have required substantial effort); "lea" is gcc 2.95.2 with Doug Lea's malloc/free replacement library v2.6.6[4] (which has much better performance than Sun's default `malloc` library); "GC" is gcc 2.95.2 with the Boehm-Weiser conservative garbage collector v5.3; "norc" is gcc 2.95.2 with our RC compiler and reference counting disabled; "RC" is gcc 2.95.2 with our RC compiler and reference counting enabled. For the benchmarks which were originally not region-based (`cfrac`, `gröbner`, `tile`, `moss`), the "lea" column is the execution time obtained when running the original code. For those benchmarks which were region-based, the "lea" column uses a simple region-emulation library that uses `malloc` and `free` to allocate and free each individual object. The "GC" column uses the same code as "lea", except that calls to `malloc` are replaced by calls to garbage collected allocation and calls to `free` are removed. RC with reference counting always per-

[4] Obtainable at ftp://g.oswego.edu/pub/misc/malloc.c

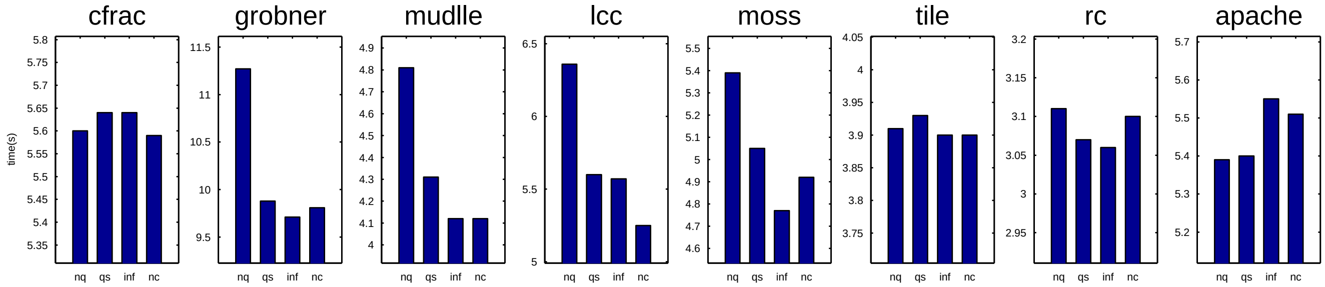

**Figure 8: Execution time with `sameregion`, `parentptr` and `traditional` (non-zero time origin)**

forms better than C@ and is faster than malloc/free or the Boehm-Weiser garbage collector on `cfrac`, `gröbner`, `mudlle`, `moss`, and `tile` (up to 58%). At worst, RC is 7% slower (on `rc`).

Table 2 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. For RC, we also show time spent removing references from deleted regions ("Region unscan"). The largest reference counting overhead is for `lcc` at 11% of execution time. The region unscan accounts for 2% or less of execution time on all other benchmarks. 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). We discuss the performance anomalies (negative time for reference counting) below.

## 5.2 Region Type System Results

We added `sameregion`, `parentptr` and `traditional` annotations to all our benchmarks. Table 3 reports the number of annotations we added, the number of lines of code we had to change to allow annotations (excluding the lines with the annotations themselves) and the percentage of assignment statements of annotated types whose safety we were able to check statically.

On most benchmarks the only changes were the addition of the `sameregion`, `parentptr` and `traditional` keywords. In `gröbner`, which represents large integers as a structure with a pointer to an array, we allocated some of these structures in a region rather than on the stack and explicitly allocated the array in the same region as the structure. This allowed us to declare the pointer to the array as `sameregion`. We perform a similar change in `lcc`. In `moss` and `lcc` we improve the results of constraint inference by replacing some uses of global variables (whose region is not tracked in our region type system) by parameters, local variables and calls to `regionof` (whose region is tracked).

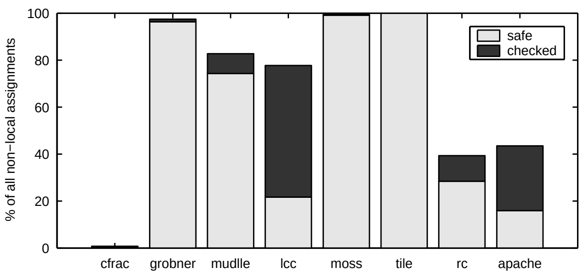

**Figure 9: Details of reference count operations**

| Name | Keywords added | Lines changed | % safe assigns |
|---|---:|---:|---:|
| cfrac | 8 | 0 | 50 |
| gröbner | 4 | 217 | 80 |
| mudlle | 75 | 0 | 88 |
| lcc | 81 | 62 | 31 |
| moss | 20 | 22 | 89 |
| tile | 21 | 0 | 84 |
| rc | 331 | 0 | 11 |
| apache | 64 | 0 | 31 |

**Table 3: `sameregion`, `parentptr` and `traditional`: static statistics**

The effects on execution time of `sameregion`, `parentptr` and `traditional` annotations and of our constraint inference system are shown in Figure 8. In the "nq" column, the annotations are ignored; in "qs" the annotations are used and checked at runtime; in "inf" the constraint inference system has removed provably safe runtime checks; in "nc" all runtime checks are (unsafely) removed ("nc" thus bounds the maximum improvement our inference system can provide). Some of these results are anomalous, showing increases in execution time as less work is performed. This is particularly obvious in `apache` ("inf" and "nc" columns), but is also visible in `moss` and `rc` ("nc" column). Our conclusion is that our performance measurements are affected by noise (due to minor changes in code and the process's environment) whose amplitude is hard to quantify, but that this noise does not affect overall conclusions when examining a sufficiently large set of benchmarks. The negative reference count time above is another instance of this phenomenon.

Figure 9 presents the runtime frequencies of several categories of pointer assignments (excluding assignments to local variables) in our benchmarks. The "safe" category is the percentage of pointer assignments to `sameregion`, `parentptr` or `traditional` pointers that were shown to be statically safe by our constraint inference. These require no runtime work. The next category, "checked", is the percentage of assignments to `sameregion`, `parentptr` or `traditional` pointers that required a runtime check. The final category is the difference between the top of the bar and 100% is the percentage of assignments that required reference counting work. The goal of our annotations is to reduce this percentage; the goal of our constraint inference system is to reduce the number of "checked" pointer assignments.

From figures 8 and 9 we conclude that our type annotations are important to the performance of `gröbner`, `mudlle`, `lcc`, `moss` and to a lesser extent `rc`. The constraint inference 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 27% instead of 11%, and the overhead of mudlle would be 23% instead of 6%. The anomalous performance results for apache prevent any useful conclusion. In all these benchmarks at least 39% of pointer assignments are of annotated types. The programs (gröbner, mudlle, tile, moss) where the percentage of annotated assignments is highest are dominated by one or two data structures which use annotated 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)
```

We do not think this is representative of typical programs.

The effectiveness of our constraint inference 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 the linked list type of Figure 1, the kinds of code whose safety our system successfully or unsuccessfully verifies. The examples will assume the following local variables are declared:

```
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 constraint inference 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. 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.[5]

# 6. CONCLUSION AND FUTURE WORK

We have designed and implemented RC, a dialect of C extended with safe regions. RC programs perform competitively with malloc/free or garbage collection based programs (from 7% slower to 58% faster) on our benchmarks. The overhead of safety is low (11% or less) on all benchmarks. This overhead is achieved with the help of type annotations that allow the programmer to easily declare some aspects of the program's region structure. We generalise these annotations 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 annotations (from 21% to 99.99%).

The current translation from RC into our region type system is very simple. There is scope for both a more elaborate translation and for more annotations in RC to make a program's region structure more explicit.

[5] 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.

## 7. REFERENCES

[1] 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 Languages Design and Implementation*, pages 187–196, Albuquerque, New Mexico, June 1993.

[2] D. G. Bobrow. Managing Re-entrant Structures using Reference Counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980.

[3] 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.

[4] 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*, Snowbird, Utah, June 2001.

[5] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1995.

[6] 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.

[7] D. Gay and A. Aiken. Language Support and Compilation Techniques for Regions. Technical Report UCB//CSD-00-1115, EECS Department, University of California, Berkeley, Nov. 2000.

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

[9] 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.

[10] D. T. Ross. The AED Free Storage Package. *Communications of the ACM*, 10(8):481–492, Aug. 1967.

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

[12] D. Stoutamire and S. Omohundro. The Sather 1.1 Specification. Technical Report TR-96-012, International Computer Science Institute, Berkeley, CA, August 1996.

[13] M. Tofte and J.-P. Talpin. Region-Based Memory Management. *Information and Computation*, 132(2):109–176, Feb. 1997.

[14] K.-P. Vo. Vmalloc: A General and Efficient Memory Allocator. *Software Practice and Experience*, 26(3):357–374, Mar. 1996.

[15] D. Walker and G. Morrisett. Alias Types for Recursive Data Structures. Technical Report TR2000-1787, Cornell University, Mar. 2000.

[16] 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.

[17] 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.