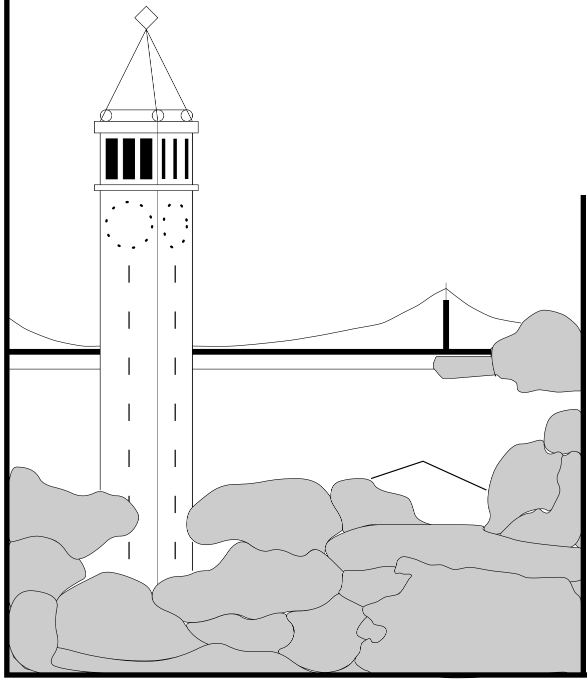
Language Support and Compilation Techniques for Regions

David Gay and Alex Aiken
{*dgay,aiken*}*@cs.berkeley.edu*



Report No. UCB//CSD-00-1115

November 2000

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Language Support and Compilation Techniques for Regions

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

Abstract

Region-based memory management systems structures memory by grouping objects in regions under program control. Memory is reclaimed by deleting regions, freeing all objects stored therein. Our compiler for C with regions, RC, prevents unsafe region deletions by keeping a count of references to each region. RC is compiled to C. Using type annotations that make the structure of a program's regions more explicit and a reference counting scheme that optimises reference count operations on local variables, we reduce the overhead of reference counting from a maximum of 27% to a maximum of 18% on our benchmarks. We generalise these annotations in a region type system whose main novelty is the use of existentially quantified abstract regions to represent pointers to objects whose region is partially or totally unknown.

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` qualifier is discussed below.

```
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, finfo);
  ... /* fill in data */
  rl->next = last; last = rl;
}
output_rlist(l);
deleteregion(r);
```

Figure 1: An example of region-based allocation.

Traditional region-based systems such as arenas [Han90] 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 languages other than C (Section 3).

We believe that region-based programming has several advantages over traditional 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 malloc and free). Second, regions provide safety with good performance: on our benchmarks, regions with reference counting are from 12% slower to 59% faster than the same programs using malloc/free or the Boehm-Weiser conservative garbage collector, and the overhead of reference counting is at most 18% of execution time. Furthermore Stoutamire [Sto97] and our earlier study of regions [GA98] show that regions can be used to improve 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 language RC makes four 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.

The second contribution of RC, and the main change over our previous system C@ [GA98], is the addition of static information in the form of two type annotations, `sameregion` and `traditional`. These annotations are based on our observations about programming patterns in large region-based applications.

- A pointer declared `sameregion` is never external, i.e., 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 is always the address of a local variable. Traditional pointers improve the in-

[*]This work was supported in part by NASA Contract No. NAG2-1210 and NSF CCR-0085949. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

teroperation of RC with legacy code that does not use regions.

The 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` or `traditional` pointers never update reference counts. Excepting one benchmark in which reference counting overhead was negligible, we found that between 31% and 99.93% 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` and `traditional`. 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 compiler for RC generates ANSI C code, allowing its use on any platform with any C compiler. Our previous system, C@, relied for performance on being able to scan stack frames to find local variables of pointer type, but this is impossible when compiling to a high-level language. Our last contribution is an algorithm that places reference count operations for local variables in the generated C code in a fashion that minimises the runtime cost of reference counting local variables. In practice we have found that this optimal algorithm is little better than a simple heuristic and has significant compile-time cost (Section 3.4).

The combination of the type annotation and runtime check elimination reduces the largest reference counting overhead from 27% to 18% of runtime. On two benchmarks, more than 90% of the reference counting cost is eliminated, on three other benchmarks between 20% and 57% of the reference counting cost is removed. Two of the three other benchmarks already have very low reference counting overhead (less than 2% 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 [TT97] and Crary, Walker and Morrisett [CWM99] include type systems that are similar to the one used in rlang: all these systems annotate pointers with a name for the targeted region. The main innovation in rlang is the introduction of existentially quantified region names that can denote any existing region. This leads to two important differences with the type systems of [TT97] and [CWM99]:

- RC loses the ability to check statically the safety of `deleteregion`, as values stored in types with existentially quantified regions are not tracked by the type system.
- RC gains the ability to represent programs such as:
```
region r[10];
for(i = 0; i < 10; i++) r[i] = newregion();
x = ralloc(r[random(0, 10)], ...);
```

There is no type for `r` in the type systems of [TT97] or [CWM99]. This code is not very useful, but similar examples are found in real programs, e.g., one of our benchmarks contains a list of environments with each environment allocated in its own region.

Our system preserves the safety of `deleteregion` by using reference counting. We believe that the gain in expressivity from the use of existential types, 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 [GA98] 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), allows use of any C compiler rather than requiring modification of an existing compiler (lcc [FH95] in [GA98]) and incorporates some static information about a program's region structure.

Regions have been used for decades. Ross [Ros67] 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 [Vo96] 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 [Han90] *arenas* are freed all at once. Barrett and Zorn [BZ93] 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 [Sto97] adds *zones*, which are garbage-collected regions, to Sather [SO96] 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 [Bob80] 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 [IY90] for use in distributed systems. Neither of these papers include any performance measurements.

Surveys of memory management can be found in [Wil92] for garbage collection and [WJNB95] for explicit allocation and deallocation.

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. The implementation of RC is given in Sections 3.3 and 3.4.

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. First, as in our system, `deleteregion` can abort the program when there remain references to the region. The second option is to simply return a failure code from `deleteregion` when its use would be unsafe. A final choice is implicit region deletion: at var-

```
typedef struct region *region;

region newregion(void);
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

ious 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.

RC is 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 cyclical data structures can be used transparently as long as the cycles are contained within a single region.

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.

Our current implementation does not detect these situations. We expect to add some support for detecting and working around these situations in later versions of RC.

3.2 Type Annotations

Our previous version of C-with-regions, C@ [GA98] 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 found this approach too restrictive: existing code cannot be used with regions without modification; 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 particular 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, 49% of runtime pointer assignments write a pointer to an object in region r into another object in the same region. Similar percentages are found in several other benchmarks. This, combined with examination of our benchmarks' source code, lead 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.

A third type qualifier, `deletes`, is used on function types to help with separate compilation (see Section 3.4).

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). Updates to local variables are particularly common, so we investigated three approaches for minimising the cost of this operation (Section 3.4). 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@ [GA98].

3.3.1 Region Library

The implementation of the region library is similar to the one in [GA98]: a region, defined by

```
struct region {
  int rc;
  struct allocator normal;
  struct allocator pointerfree;
};
```

(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 (regionof(newval) != regionof(p))
    abort();
traditional:
  if (regionof(newval) != traditional_region)
    abort();
```

Figure 3: Reference counting and annotation checking

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).

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` and `traditional` pointers only need one of the runtime checks of Figure 3(b) rather than a reference count update. These checks take only 8 SPARC instructions (and do not need to read the value being overwritten). Section 4.5 discusses how we eliminate a significant fraction of these runtime checks.

The references from local variables are discussed in Section 3.4.

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 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 most benchmarks, 6% in one case). However, we plan to investigate ways of reducing this cost further.

[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.

3.4 Local Variable Reference Counts

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 [GA98] approach and have `deleteregion` scan the stack for pointers to regions from local variables. Instead we place operations to increment and decrement the reference counts of regions referred to from local variables in positions that guarantee that the reference count is correct when calling `deleteregion` but that allows the count to be incorrect at other times. We investigated three placement schemes. For each scheme, we will write `incrc`(v) and `decrc`(v) for the operation that increments or decrements the reference count of the region referred to by local pointer variable v:

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

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

3.4.1 Minimising local variable reference counting

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

- S: The statements where the reference counts must take v into account. The variable v is said to be *counted*. These statements are all calls to functions that may delete a region.
- U: The statements where the reference counts must not take v into account. The variable is said to be *uncounted*. These statements are assignments to v and all points where v is dead.
- O: All other statements.

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

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

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

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

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

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

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

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

An optimal assignment of white and black nodes for G' is the same as a partition of G' into two graphs, with sb and sw separated, which minimises the weight of the cut edges. In other words, the optimal assignment is a minimum cut of G' viewed as an undirected graph and with sb, sw separated by the cut. This minimum cut can be found by finding the maximum flow [CLR90] on G' (viewed as an undirected graph) with source sw and sink sb. The actual cut is found by disconnecting all edges saturated in the maximum flow. All nodes reachable from sb are black, all other nodes are white.

There is an $O(|V'||E'|\log(|V'|^2/|E'|))$ algorithm for maximum flow [GT88]. In control-flow graphs $|E| \leq 2|V|$, so for our problem the complexity is $O(|V|^2)$. A separate minimum cut problem must be solved for each local variable, so the total complexity is $O(|V|^3)$ (assuming the number of local variables is proportional to the function size).

3.4.2 Implementation

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

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

$\tau = \mu@\sigma \mid \exists\rho \leq \sigma.\tau$ (types)
$\mu = \texttt{region} \mid T[\sigma_1, \ldots, \sigma_m]$ (base types)
$\sigma = \rho \mid R \mid \bot$ (region expressions)
$\texttt{struct}\ T[\rho_1, \ldots, \rho_m]\{field_1 : \tau_1, \ldots, field_n : \tau_n\}$
(structure declarations)

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

Figure 4: Region type language

4 A Region Type System

The type annotations of Section 3.2 can be viewed as a simple way for the user to specify types from a more general region type language (Section 4.1) which partially tracks the regions of pointers. This type language is used in a simple region-based language *rlang* (Section 4.2). We use a simple semantics for rlang (Section 4.3) to show the soundness of our type checking rules (Section 4.4). 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 4.5).

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 $A_H = \{\bot, r_1, \ldots, r_n\}$ for the set of regions of H. We define a partial order on A_H: $r \preceq r'$ if $r = \bot \vee r = r'$. The region of an object pointer is the region of the targeted object. The region of a pointer v is $\bot$ iff $v = \texttt{null}$.

Figure 4 gives types for pointers that reflect this heap structure: there are pointers to regions (`region`), and pointers to named records with named fields. All types are annotated with a *region expression* σ 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* ρ or elements of the set $C_R = R \cup \{\bot\}$ 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 $\exists\rho \leq \sigma.\tau$ construct, which means that ρ is a region in A_H that meets the specified constraint under the $\preceq$ relation. The scope of ρ includes the bound,[3] so the type $\exists\rho \leq \rho.T[\ldots]@\rho$ represents an object of type T in any region. To simplify notation, we write $\exists\rho$ as a shorthand for $\exists\rho \leq \rho$. 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).

[3]This avoids having separate existential constructs for bounded and unbounded regions and so simplifies the type-checking rules.

If two values point to the same abstract region ρ then the values must specify objects in the same region. As a consequence, if one of the values is null then $\rho = \bot$ 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, the structure definition

$$\texttt{struct}\ L[\rho]\{v : \exists\rho'.\texttt{region}@\rho', next : \exists\rho'' \leq \rho.L[\rho'']@\rho''\}$$

is a list stored in region ρ 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 Typechecking 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 x after s has executed. The set of simple region expressions valid in the argument and result types of f is $\{\rho_1, \ldots, \rho_m\} \cup C_R$. The set of region expressions valid in the local variables of f is $\{\rho_1, \ldots, \rho_m, \rho'_1, \ldots, \rho'_p\} \cup C_R$. The local variables $x'_1, \ldots, x'_q$ must be dead before s. The meaning of the input and output *constraint sets* $\textsc{c}$ and $\textsc{c}'$ is given below. The `chk` $x_0 \leq x_1$ (or R) statement is a runtime check that x_0 is null or in the same region as x_1 (or in the region denoted by region constant R). 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 operations. The rest of the language is straightforward: `if` and `while` statements assume `null` is false and everything else is true; `new` statements specify values for the structure's fields; the program is executed by calling a function called `main` with no arguments. Figure 5 also gives signatures for the predefined `newregion`, `deleteregion` and `regionof_`T (one for each structure type T) functions.

A *constraint set* ($\textsc{c}, \textsc{d}$, etc) specifies properties of a set of region expressions. The *domain* of a constraint set, written $\text{dom}(\textsc{c})$ is the set of region expressions appearing in $\textsc{c}$. A constraint set can specify:

- That σ is not the $\bot$ region. We write $\textsc{c} \vdash \sigma \neq \bot$.
- That region $\sigma_1 \leq \sigma_2$. We write $\textsc{c} \vdash \sigma_1 \leq \sigma_2$.
  $\textsc{c} \vdash \sigma_1 = \sigma_2$ is shorthand for $\textsc{c} \vdash \sigma_1 \leq \sigma_2 \wedge \textsc{c} \vdash \sigma_2 \leq \sigma_1$.

A function f parameterised over abstract regions $\rho_1, \ldots, \rho_m$ has an input constraint set $\textsc{c}$ with domain $\{\rho_1, \ldots, \rho_m\} \cup C_R$ which expresses requirements on the regions of f's arguments that the callers of f must respect, and an output constraint set $\textsc{c}'$ (with the same domain) which expresses the constraints that are known to hold when f exits.

A constraint set $\textsc{c}$ must respect the following properties ($\sigma_1, \sigma_2, \sigma_3$ are region expressions from $\text{dom}(\textsc{c})$):

$$\begin{array}{ll}
\textsc{c} \vdash \sigma_1 \leq \sigma_1 & \text{(reflexivity)} \\
\textsc{c} \vdash \sigma_1 \leq \sigma_2 \wedge \textsc{c} \vdash \sigma_2 \leq \sigma_3 \Rightarrow \textsc{c} \vdash \sigma_1 \leq \sigma_3 & \text{(transitivity)} \\
\textsc{c} \vdash \bot \leq \sigma_1 & \\
\textsc{c} \vdash \sigma_1 \neq \bot \wedge \textsc{c} \vdash \sigma_1 \leq \sigma_2 \Rightarrow \textsc{c} \vdash \sigma_1 = \sigma_2 \wedge \textsc{c} \vdash \sigma_2 \neq \bot &
\end{array}$$

$$\text{program} ::= \text{fn}^*$$

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

Some predefined functions:
$\texttt{newregion}[][\emptyset]() : \exists\rho.\texttt{region}@\rho, \emptyset$
$\texttt{deleteregion}[\rho][\emptyset](r : \texttt{region}@\rho) : \texttt{region}@\bot, \emptyset$
$\texttt{regionof\_}T[\rho, \rho_1, \ldots][\emptyset](x : T[\rho_1, \ldots]@\rho) : \texttt{region}@\rho, \emptyset$

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

Constraint sets form a lattice with the following partial order:

$$\begin{array}{ll}
\textsc{c} \leq \textsc{c}' \text{ if } \forall\sigma_1, \sigma_2 \in \text{dom}(\textsc{c}) & (\textsc{c} \vdash \sigma_1 \neq \bot \Rightarrow \textsc{c}' \vdash \sigma_1 \neq \bot) \wedge \\
& (\textsc{c} \vdash \sigma_1 \leq \sigma_2 \Rightarrow \textsc{c}' \vdash \sigma_1 \leq \sigma_2)
\end{array}$$

The least constraint set is $\emptyset$.

We define some transformations on constraint sets. In each case the resulting set is the smallest constraint set that meets the specified constraints (σ_1, σ_2 are any region expression from $\text{dom}(\textsc{c})$):

- $\textsc{c}[\neg\rho]$ (kill an abstract region):
  $(\sigma_1 \neq \rho \wedge \textsc{c} \vdash \sigma_1 \neq \bot \Rightarrow \textsc{c}[\neg\rho] \vdash \sigma_1 \neq \bot) \wedge$
  $(\sigma_1 \neq \rho \wedge \sigma_2 \neq \rho \wedge \textsc{c} \vdash \sigma_1 \leq \sigma_2 \Rightarrow \textsc{c}[\neg\rho] \vdash \sigma_1 \leq \sigma_2$
- $\textsc{c}[\sigma_1 \leq \sigma_2]$ (assert order of abstract regions):
  $(\textsc{c}[\sigma_1 \leq \sigma_2] \vdash \sigma_1 \leq \sigma_2) \wedge \textsc{c} \leq \textsc{c}[\sigma_1 \leq \sigma_2]$
- $\textsc{c}[\sigma \neq \bot]$ (assert an abstract region is not $\bot$):
  $\textsc{c}[\sigma \neq \bot] \vdash \sigma \neq \bot \wedge \textsc{c} \leq \textsc{c}[\sigma \neq \bot]$.
- $\textsc{c}[\sigma_1 = \sigma_2]$ is shorthand for $\textsc{c}[\sigma_1 \leq \sigma_2][\sigma_2 \leq \sigma_1]$.
- $\textsc{c}//S$ restricts $\textsc{c}$'s domain to S.

We write $X[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m]$ for substitution of region expressions for (free) abstract regions in region expressions, types and constraint sets. The notation $x : \tau$ and $x.field : \tau$ asserts that x, or a field of x, has type τ.

Type checking for rlang (Figure 6) relies extensively on constraint sets. Statements of a function f are checked by the judgment $\textsc{c}, L \vdash s, \textsc{c}'$. These judgments take an input constraint set $\textsc{c}$ with domain $\{\rho_1, \ldots, \rho_m, \rho'_1, \ldots, \rho'_p\} \cup C_R$ (describing the properties of all arguments and live local variables) and produce an output constraint set $\textsc{c}'$. Instead of an explicit binding construct for abstract regions, assignments may bind any abstract region of the assignment target which is not used in any function argument or live variable. This set of region expressions that cannot be bound at entry to s is called the *live region expression* set and is denoted by L (or L_s where necessary for clarity). We assume that L is precomputed for each statement using a standard liveness

$$\dfrac{\textsc{c}, L_s \vdash s, \textsc{c}' \qquad x : \tau \qquad \textsc{c}'' \leq \textsc{c}' \qquad x'_1, \ldots, x'_q \text{ are dead before } s}{\vdash f[\rho_1, \ldots, \rho_m][\textsc{c}](x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau, \textsc{c}'' \ \mathtt{is}\ [\rho'_1, \ldots, \rho'_p] x'_1 : \tau'_1, \ldots, x'_q : \tau'_q, s, x} \quad \text{(fndef)}$$

$$\dfrac{x_0 : \tau_0 \quad x_1 : \tau_1 \quad \textsc{c}, L \vdash \tau_0 \leftarrow \tau_1, \textsc{c}', L'}{\textsc{c}, L \vdash x_0 = x_1, \textsc{c}'} \quad \text{(assign)}$$

$$\dfrac{x_0 : \tau_0 \quad x_1 : \mu_1 @ \sigma_1 \quad x_1.\mathit{field} : \tau'_1 \quad \textsc{c}[\sigma_1 \neq \bot], L \vdash \tau_0 \leftarrow \tau'_1, \textsc{c}', L'}{\textsc{c}, L \vdash x_0 = x_1.\mathit{field}, \textsc{c}'} \quad \text{(read)}$$

$$\dfrac{x_1 : \mu_1 @ \sigma_1 \quad x_1.\mathit{field} : \tau'_1 \quad x_2 : \tau_2 \quad \textsc{c}[\sigma_1 \neq \bot], L \vdash \tau'_1 \leftarrow \tau_2, \textsc{c}', L'}{\textsc{c}, L \vdash x_1.\mathit{field} = x_2, \textsc{c}'} \quad \text{(write)}$$

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

$$\dfrac{x_0 : \mu_0 @ \sigma_0 \quad \textsc{c}, L \vdash \mu_0 @ \sigma_0 \leftarrow \mu_0 @ \bot, \textsc{c}', L'}{\textsc{c}, L \vdash x_0 = \mathtt{null}, \textsc{c}'} \quad \text{(null)}$$

$$\dfrac{x_0 : \mu_0 @ \sigma_0 \qquad x_1 : \mu_1 @ \sigma_1}{\textsc{c}, L \vdash \mathtt{chk}\ x_0 \leq x_1, \textsc{c}[\sigma_0 \leq \sigma_1]} \quad \text{(check)} \qquad \dfrac{x_0 : \mu_0 @ \sigma_0}{\textsc{c}, L \vdash \mathtt{chk}\ x_0 \leq R, \textsc{c}[\sigma_0 \leq R]} \quad \text{(check const)}$$

$$\dfrac{\textsc{c}, L \vdash s_1, \textsc{c}' \quad \textsc{c}', L_{s_2} \vdash s_2, \textsc{c}''}{\textsc{c}, L \vdash s_1; s_2, \textsc{c}''} \qquad \dfrac{\textsc{c}, L_{s_1} \vdash s_1, \textsc{c}' \quad \textsc{c}, L_{s_2} \vdash s_2, \textsc{c}''}{\textsc{c}, L \vdash \mathtt{if}\ x\ s_1\ s_2, \textsc{c}' \sqcap \textsc{c}''} \qquad \dfrac{\textsc{c}', L_s \vdash s, \textsc{c}'' \quad \textsc{c}' = \textsc{c} \sqcap \textsc{c}''}{\textsc{c}, L \vdash \mathtt{while}\ x\ s, \textsc{c}'}$$

$$\dfrac{\begin{array}{c} f[\rho_1, \ldots, \rho_m][\textsc{d}](y_1 : \tau'_1, \ldots y_n : \tau'_n) : \tau', \textsc{d}' \\ x_i : \tau_i \quad \textsc{c}_i, L_i \vdash \tau'_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m] \leftarrow \tau_i, \textsc{c}_{i+1}, L_{i+1} \quad \textsc{d}[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m] \leq \textsc{c}_{n+1} \\ \sigma_i \in F_{\tau'} \iff \rho_i \text{ free in } \tau' \text{ and for all } k, \rho_i \text{ not free in } \tau'_k \qquad F_{\tau'} \cap L_{n+1} = \emptyset \\ \textsc{c}_{n+1} \sqcup \textsc{d}'[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m], L_{n+1} \cup F_{\tau'} \vdash \tau_0 \leftarrow \tau'[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m], \textsc{c}', L' \end{array}}{\textsc{c}_1, L_1 \vdash x_0 = f[\sigma_1, \ldots, \sigma_m](x_1, \ldots, x_n), \textsc{c}'} \quad \text{(fncall)}$$

<u>Assignment</u>

$$\dfrac{\begin{array}{c} \sigma' \in L \quad \sigma[\sigma'/\rho] \in L \quad \textsc{c} \vdash \sigma' \leq \sigma[\sigma'/\rho] \\ \textsc{c}, L \vdash \tau[\sigma'/\rho] \leftarrow \tau', \textsc{c}', L' \end{array}}{\textsc{c}, L \vdash \exists \rho \leq \sigma.\tau \leftarrow \tau', \textsc{c}', L'} \quad (\exists\text{gen.}) \qquad \dfrac{\begin{array}{c} \rho \notin L \qquad \rho \in \mathrm{dom}(\textsc{c}) \\ \textsc{c}[\neg\rho][\rho \leq \sigma'[\rho/\rho']], L \cup \{\rho\} \vdash \tau \leftarrow \tau'[\rho/\rho'], \textsc{c}', L' \end{array}}{\textsc{c}, L \vdash \tau \leftarrow \exists \rho' \leq \sigma'.\tau', \textsc{c}', L'} \quad (\exists\text{inst.})$$

$$\dfrac{\textsc{c}, L \vdash \sigma \leftarrow \sigma', \textsc{c}', L'}{\textsc{c}, L \vdash \mathtt{region} @ \sigma \leftarrow \mathtt{region} @ \sigma', \textsc{c}', L'} \qquad \dfrac{\textsc{c}, L \vdash \sigma \leftarrow \sigma', \textsc{c}_1, L_1 \qquad \textsc{c}_i, L_i \vdash \sigma_i \leftarrow \sigma'_i, \textsc{c}_{i+1}, L_{i+1}}{\textsc{c}, L \vdash T[\sigma_1, \ldots, \sigma_m] @ \sigma \leftarrow T[\sigma'_1, \ldots, \sigma'_m] @ \sigma', \textsc{c}_{m+1}, L_{m+1}}$$

$$\dfrac{\sigma \in L \quad \textsc{c} \vdash \sigma = \sigma'}{\textsc{c}, L \vdash \sigma \leftarrow \sigma', \textsc{c}, L} \qquad \dfrac{\rho \notin L}{\textsc{c}, L \vdash \rho \leftarrow \sigma', \textsc{c}[\neg\rho][\rho = \sigma'], L \cup \{\rho\}}$$

Figure 6: Region Typechecking

analysis. The abstract regions of a function's arguments and the constant regions C_R are included in all sets L and so cannot be rebound. This guarantees that the properties asserted in a function's output constraint set actually apply to the region expressions of the function's input arguments. Notice that abstract regions used only in the function's result type are not affected by this restriction (see the discussion of function calls below).

The judgments $\textsc{c}, L \vdash \tau_1 \leftarrow \tau_2, \textsc{c}', L'$ of Figure 6 check that a value of type τ_2 is assignable to a location of type τ_1. These judgments take an input constraint set $\textsc{c}$ and live region expression set L and produce an updated (as a result of binding abstract regions) output constraint set $\textsc{c}'$ and live region expression set L'. The ($\exists$gen.) rule allows assignment as long as τ_2 can be existentially quantified to match τ_1. The ($\exists$inst.) rule allows instantiation of an existentially quantified region into a dead abstract region ρ, and updates $\textsc{c}$ and L to reflect ρ's new properties. Base types are assignable if their region expressions match. Two region expressions match if they are equal according to $\textsc{c}$ or if the abstract region ρ of the assignment target is dead. In this last case $\textsc{c}$ is updated to reflect ρ'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 $\bot$. 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 $\bot$. After execution of a runtime check, the checked relation holds (check and check const).

The rules 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 constraints must be a subset of the function body's output constraints.

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 constraints at the call site match f's input constraint. The set $F_{\tau'}$ is the region arguments of the call to f that correspond to abstract regions mentioned solely in f's return type. Elements of $F_{\tau'}$ are bound on return from f, so we require that they be dead (not in L_{n+1}) before the call. After the call, f's output constraints are known to hold and f's result must be assignable to the call's destination. These special rules for elements of $F_{\tau'}$ make the translation of RC into rlang simpler (see Section 4.5).

4.3 Semantics

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

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

- A *value* (or *pointer*) is represented as a unique natural integer. `null` pointers are represented by 0.
- A *region* is represented as a unique natural integer. The $\perp$ region is represented by 0. We represent our partial order on regions with $\preceq$, so $n \preceq m$ if $n = 0 \lor n = m$.
- Given a type `struct` $T[\rho_1, \ldots, \rho_m]\{f_1 : \tau_1, \ldots, f_n : \tau_n\}$, an *object o of type T* is represented as a pair (R, P) containing a tuple of regions $R = (r_0, r_1, \ldots, r_m)$ and a tuple of values $P = (v_1, \ldots, v_m)$. The region of o is r_0, r_i is the value of ρ_i and v_i is the value of f_i. As the $\perp$ region contains no object $r_0 \neq 0$. The object representing region r is the pair $((r), ())$. Note that the $\perp$ region is represented by object $((0), ())$.
- A *heap H* is a partial map from $\mathbb{N}$ to objects, with $0 \notin \mathrm{dom}(H)$. Formally, $H : \mathbb{N} \hookrightarrow (\bigcup_{i=1}^{\infty} \mathbb{N}^i) \times (\bigcup_{i=0}^{\infty} \mathbb{N}^i)$. We assume that the set A_H of regions of H is available. For simplicity of notation the source language names of the region constants are reused as names for the corresponding runtime regions, so $C_R \subseteq A_H$.

Our semantics will use two environments during evaluation:

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

The natural operational semantics (Figure 7) has rules of the form $< H, E, R, s > \rightsquigarrow < H', E', R' >$ meaning that evaluation of s with heap H, environment E and abstract region map R produces a heap H', environment E' and abstract region map R'. The rules for assignment (s_assign), field read (s_read) and write (s_write) are straightforward: they apply sub-rules for assignment (Figure 8, detailed below) to update the abstract region map, then modify the environment or heap as necessary. Creation of an object (s_new) is similar, but must pick an unique value (v) for the pointer to the new object. Assignment of `null` (s_null) is a little strange as it does not update R for the abstract regions (used in μ_0) bound as a result of this assignment. These newly bound abstract regions have no meaningful value as x_0 does not point to an object after being assigned `null`.

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

An implementation of rlang does not need the abstract region map, and its heap need only contain the field values and the r_0 field of the region tuple (which is necessary for implementing the `regionof` function, reference counting and the `chk` operations).

Assignment (Figure 6) binds abstract regions, so can update the abstract region map R. Hence assignment reduction rules (Figure 8) take the form $< H, R, v, \tau_1, \tau_2 > \rightsquigarrow R'$ to represent assignment of a value v of type τ_2 to a location of type τ_1 with heap H and abstract region map R. These rules return an updated abstract region map R'.

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

The instantiation of existential types is more complex as the reduction rules must pick a region r for the instantiated abstract region ρ. This is straightforward if ρ' is used in the base type of τ' or if ρ' is the region of v: r can be found in the object stored at $H(v)$. If $v = 0$, ρ' is used only as a bound in subsequent existential quantifiers in τ', or is not used at all in τ' there is no value for r that can be read directly from the heap. Rather than enumerate all the cases that must be considered, we instead pick an arbitrary r that that matches the existential quantifier's bound and is consistent with τ' and $H(v)$, as specified by *partial consistency*. Partial consistency asserts that the regions stored in the heap object for v match those specified in v's type (we write $\mathrm{fr}(\tau)$ for the free abstract regions of a type τ):

Definition 4.1 *$v : \tau$ is partially consistent with H under R* (with $\mathrm{fr}(\tau) \subseteq \mathrm{dom}(R)$) if it is not partially inconsistent with H under R.[4] *$v : \tau$ is partially inconsistent with H under R*:

- if $v = 0$ and $\tau = \mu@\sigma$ then $R\sigma \neq 0$.

[4]We choose to make partial inconsistency the primary definition to match Definition 4.2.

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

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

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

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

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

$$\frac{H(E(x_0)) = ((r, \ldots), \_) \qquad H(E(x_1)) = ((r, \ldots), \_)}{< H, E, R, \texttt{chk}\ x_0 \leq x_1 > \rightsquigarrow < H, E, R >} \quad \text{(s\_chk)}$$

$$\frac{E(x_0) = 0}{< H, E, R, \texttt{chk}\ x_0 \leq x_1 > \rightsquigarrow < H, E, R >} \quad \text{(s\_chk\_null)}$$

$$\frac{H(E(x_0)) = ((R_0, \ldots), \ldots)}{< H, E, R, \texttt{chk}\ x_0 \leq R_0 > \rightsquigarrow < H, E, R >} \quad \text{(s\_chk\_const)}$$

$$\frac{E(x_0) = 0}{< H, E, R, \texttt{chk}\ x_0 \leq R_0 > \rightsquigarrow < H, E, R >} \quad \text{(s\_chk\_const\_null)}$$

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

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

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

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

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

Figure 7: Semantic reduction rules

$$\frac{< H, R, v, \tau[\sigma'/\rho], \tau' > \leadsto R' \qquad \sigma' \text{ from } \exists\text{gen.}}{< H, R, v, \exists\rho \leq \sigma.\tau, \tau' > \leadsto R'} \quad \text{(a\_gen)}$$

$$\frac{\begin{array}{c}\text{exists } r \in A_H \text{ such that } v : \tau' \text{ is partially consistent with } H \text{ under } R[\rho' = r] \\ r \preceq R[\rho' = r]\sigma' \qquad \rho \text{ from } \exists\text{inst.} \qquad < H, R[\rho = r], v, \tau, \tau'[\rho/\rho'] > \leadsto R'\end{array}}{< H, R, v, \tau, \exists\rho' \leq \sigma'.\tau' > \leadsto R'} \quad \text{(a\_inst)}$$

$$\frac{\begin{array}{c}\text{there does not exist } r \in A_H \text{ such that } r \preceq R[\rho' = r]\sigma' \\ \text{and } v : \tau' \text{ is partially consistent with } H \text{ under } R[\rho' = r]\end{array}}{< H, R, v, \tau, \exists\rho' \leq \sigma'.\tau' > \leadsto R} \quad \text{(a\_inst\_unsafe)}$$

$$\frac{}{< H, R, v, \texttt{region}@\sigma, \texttt{region}@\sigma' > \leadsto R[\sigma = R\sigma']} \quad \text{(a\_region)}$$

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

Figure 8: Semantic assignment rules

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

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

4.4 Soundness

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

Definition 4.2 *$v : \tau$ is consistent with H under R* (with $\text{fr}(\tau) \subseteq \text{dom}(R)$) if it is not inconsistent with H under R. *$v : \tau$ is inconsistent with H under R*:

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

Definition 4.3 A *set of values $v_1 : \tau_1, \ldots, v_n : \tau_n$ is consistent with H under R* if each $v_i : \tau_i$ is consistent with H under R.

Definition 4.4 An abstract region map R over X is *consistent with a constraint set C* (with $X \subseteq \text{dom}(C)$) if $\forall \sigma, \sigma_1, \sigma_2 \in X$: $C \vdash \sigma \neq \bot \Rightarrow R\sigma \neq 0$ and $C \vdash \sigma_1 \leq \sigma_2 \Rightarrow R\sigma_1 \preceq R\sigma_2$.

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

The main soundness theorem is as follows:

Theorem 4.5 *Soundness:* If

- $C, L \vdash s, C'$
- $< H, E, R, s > \leadsto < H', E', R' >$
- Variables $x_1 : \tau_1, \ldots, x_n : \tau_n$ are live before s
- Variables $x_1' : \tau_1', \ldots, x_m' : \tau_m'$ are live after s
- R is consistent with C
- $E(x_1) : \tau_1, \ldots, E(x_n) : \tau_n$ are consistent with H under R
- $w_1 : \alpha_1, \ldots, w_l : \alpha_l$ are consistent with H under Q

then

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

Proof: See Appendix A.

An important lemma used in the proof of soundness shows that the semantic rules for assignment preserve soundness, and that the (a_inst_unsafe) reduction rule is never used:

Lemma 4.6 *Assignability* If:

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

then $v : \tau_1$ is consistent with H under R', R' is consistent with C' and $R//L = R'//L$. Additionally, the (a_inst_unsafe) assignment rule is not used in the reduction.
Proof: See Appendix A

4.5 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 [TT97] 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` 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 ρ represents the region in which the structure is stored. So pointers to X in P' are always of the form $X[\sigma]@\sigma$.

  A field f in $X[\rho]$ of type T which is not `sameregion` or `traditional` in P can point to any region. So its type in P' is $\exists\rho'.T[\rho']@\rho'$. If f is `traditional` then it can be `null` or point to the traditional region so its type is $\exists\rho' \leq R_T.T[\rho']@\rho'$. If f is `sameregion` then it can be `null` or point to an object in ρ, so its type is $\exists\rho' \leq \rho.T[\rho']@\rho'$. For example,

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

  `struct` $L[\rho]\{v : \exists\rho'.\texttt{region}@\rho', n : \exists\rho' \leq \rho.L[\rho']@\rho'\}$

  Global variables are represented as fields of a `Global` structure, stored in the traditional region, which is passed to every function.
- Every field assignment $x_1.f = x_2$ is immediately preceded by an appropriate runtime check: `chk` $x_2 \leq x_1$ if f is `sameregion` in P; `chk` $x_2 \leq R_T$ 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` or `traditional` are not met.
- Every local variable and function argument x in P' is associated with a distinct abstract region ρ_x. If x is of type T in P, its type becomes $T[\rho_x]@\rho_x$ in P'. Function arguments are never assigned or used directly as the function result, and the destination of an assignment is not used elsewhere in the assignment statement. [5]
- The result type of a function f is parameterised by a distinct abstract region ρ_f. Combined with the previous rule, this implies that a function f with arguments T and result T' always has signature

  $$f[\rho_x, \rho_f][C](x : T[\rho_x]@\rho_x) : T'[\rho_f]@\rho_f, C'$$

  for some constraint sets C and C'. The use of a distinct abstract region for f's result relies on the special handling of abstract regions not used in the function arguments (Figure 6). This allows us to have the same type (ignoring the constraint sets) for a function returning the region of its argument (`myregionof`) and for a function returning a new region (`mynewregion`):

  $\texttt{myregionof}[\rho_x, \rho_f][\emptyset](x : T[\rho_x]@\rho_x) :$
  $\quad \texttt{region}@\rho_f, [\rho_f = \rho_x]$

  $\texttt{mynewregion}[\rho_x, \rho_f][\emptyset](x : T[\rho_x]@\rho_x) :$
  $\quad \texttt{region}@\rho_f, [\emptyset]$

Without the special rule for function result types, we would have to perform type inference to find function result types, or would have to assume that all result types were existential, which would reduce the precision of our analysis.

It is easy to verify that an rlang program with these properties can be type checked, under the assumption that all function input and output constraint sets are $\emptyset$.

However we can infer better constraint sets using the typechecking rules: we start by assuming all possible constraints hold (including contradictions such as $\perp \neq \perp$) everywhere, except for `main`'s input constraint set which is $\emptyset$. We then iteratively remove any constraints that violate the typechecking rules of Figure 6 until a valid typing is found. The operations transforming input to output constraint sets are monotonic, the constraint sets form a lattice and there is at least one solution, so this process will find the greatest valid constraint sets and hence the most precise typing. Once the best constraint sets have been found, any `chk` statement that asserts a relation that already holds in its input constraint set can be safely eliminated.

RC implements the transformation and analysis outlined above on a single source file. Calls to unknown functions (including calls via function pointers) are conservatively assumed to have the empty input and output constraint set, any function callable from other files (or used as a function pointer) is conservatively required to have the empty 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 [GA98]). Table 1 reports the benchmarks' sizes (in lines of code) and summarises their memory allocation behaviour.

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

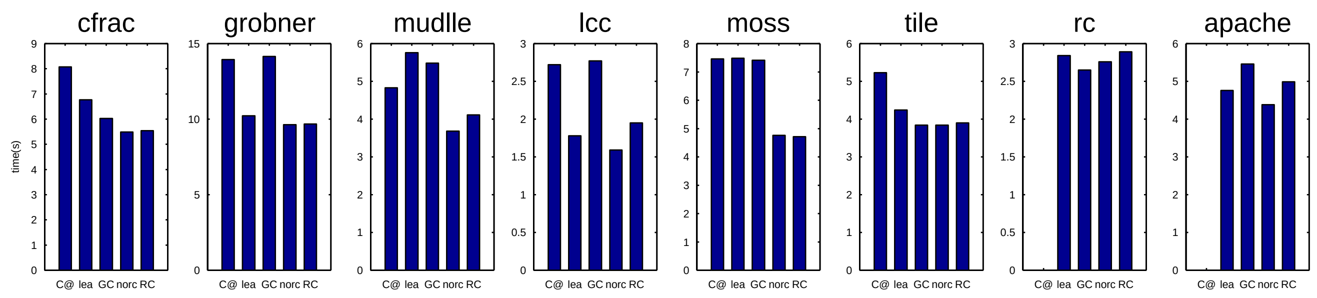


Figure 9: Execution time

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	380206	22236	2470
moss	2675	553986	6312	2185
tile	926	10459	309	153
rc	22823	80756	4714	4204
apache	62289	164296	30806	78

Table 1: Benchmark characteristics.

Name	C@ (s)	RC (s)	Region unscan (s)
cfrac	0.53	0.05	.01
gröbner	1.12	0.05	.02
mudlle	0.65	0.43	.09
lcc	0.44	0.36	.11
moss	0.30	-0.05	<.01
tile	-0.03	0.06	<.01
rc		0.13	.01
apache		0.62	.10

Table 2: Reference counting overhead in RC and C@

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 9 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[6] (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, reference counting enabled and the "Function" approach for reference counting local variables. 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 performs better than C@ and is faster than malloc/free or the Boehm-Weiser garbage collector on `cfrac`, `gröbner`, `mudlle` and `moss` (up to 59%). At worst, RC is 12% slower (on `lcc`).

Table 2 shows the reference counting cost for C@ and RC, and, for RC, the time spent removing references from deleted regions ("Region unscan"). The largest reference counting overhead is for `lcc` at 18% of execution time, it is below 12% on all other benchmarks. The region unscan accounts for 6% of execution time on `lcc`, and 2% or less on all other benchmarks. This table also shows that the better performance of RC over C@ is due to to both a better base compiler (gcc vs lcc) and 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.

[6]This library is available at ftp://g.oswego.edu/pub/misc/malloc.c

5.2 Region Type System results

We added `sameregion` 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` 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`. In `lcc` and `moss` 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 10 shows the effects on execution time of the `sameregion` and `traditional` annotations and of our constraint inference system. 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 (and is thus the maxi-

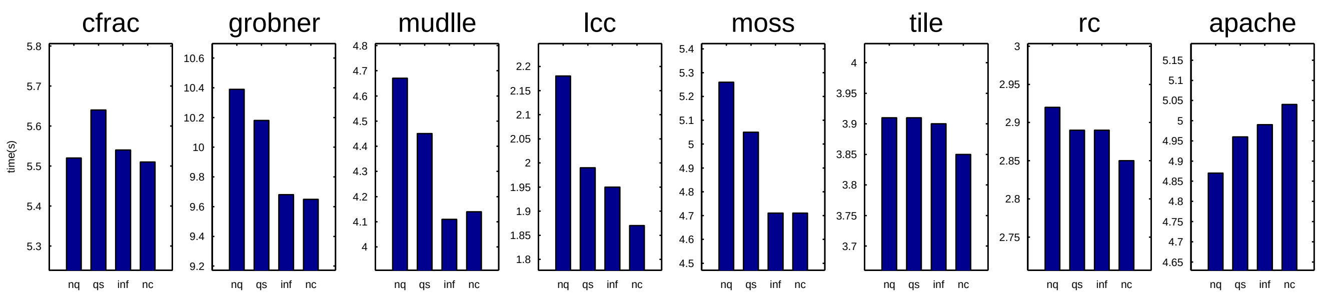


Figure 10: Execution time with `sameregion` and `traditional`

Name	Keywords added	Lines changed	% safe assigns
cfrac	8	0	50
gröbner	4	217	80
mudlle	59	0	90
lcc	57	3	18
moss	20	22	89
tile	21	0	84
rc	331	0	7
apache	47	0	37

Table 3: `sameregion` and `traditional`: static statistics

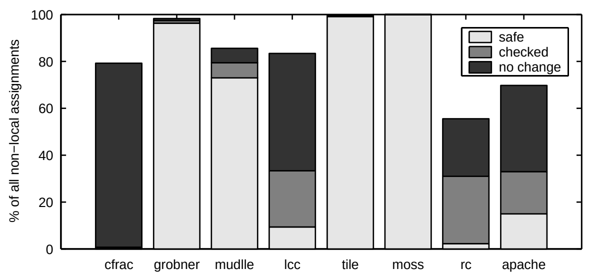


Figure 11: Details of reference count operations

mum improvement our inference system can provide). Some of these results are anomalous: the change from "nq" to "qs" in `cfrac` should produce no change in execution time as less than 1% of all pointer assignments are to annotated types and the total reference count overhead is 0.05s. The execution time for `apache` increases as less work is performed. 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 times above are other instances of this phenomenon.

Figure 11 presents the runtime frequencies of several categories of pointer assignments (excluding assignments to local variables) in our benchmarks. The first category is "safe", the percentage of pointer assignments to `sameregion` 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` or `traditional` pointers that required a runtime check. The remaining category, "no change", is the percentage of assignments to unannotated pointer types that did not lead to a change in any region's reference count. The goal of our annotations is to reduce the number of "no change" pointer assignments; the goal of our constraint inference system is to reduce the number of "checked" pointer assignments.

From figures 10 and 11 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 18%, and the overhead of `mudlle` would be 21% instead of 10%. The anomalous performance results for `apache` prevent any useful conclusion. In all these benchmarks at least 30% of pointer assignments are of annotated types. The programs (`gröbner`, `mudlle`, `tile`, `moss`) where the percentage of annotated assignments is high 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 `rc`, 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 *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->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.

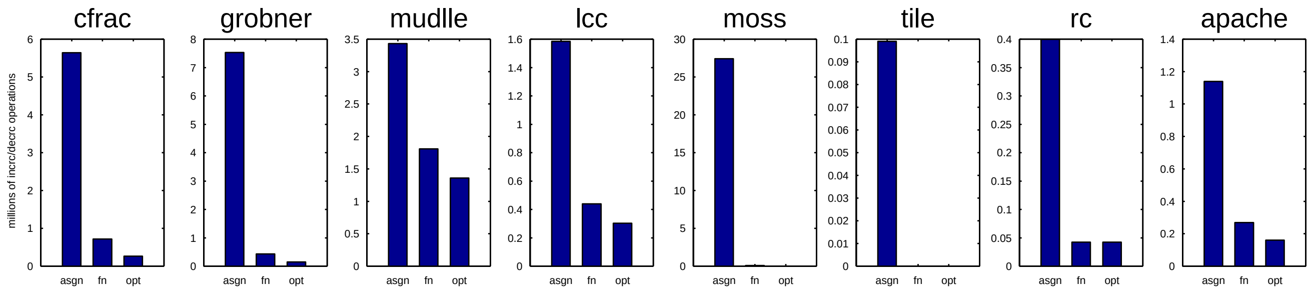


Figure 12: Number of `incrc` and `decrc` operations

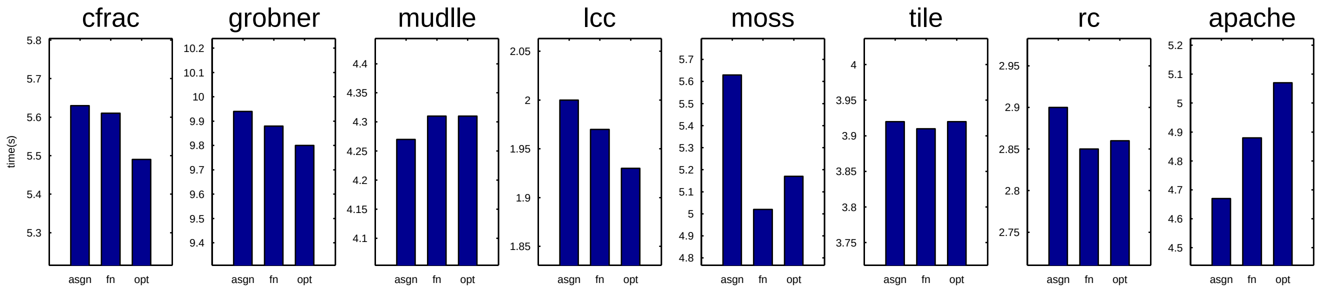


Figure 13: Performance of `incrc`/`decrc` placement algorithms

The `sameregion` 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 interprocedural 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), 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`.[7]

5.3 Local Variable Reference Counting

Figure 12 shows the number of `incrc` and `decrc` operations executed to maintain the reference counts from local variables using the three approaches presented in Section 3.4: "asgn" is Assignment, "fn" is Function and "opt" is Optimal. Figure 13 shows the corresponding variations in runtime, which are not always consistent with the reduction in work.

The Optimal algorithm always gives the lowest number of scan and unscan operations (from from 61% to 99.998% less than Assignment), but Function also keeps the overhead of reference counting local variables low and has the simplest implementation of all three approaches. In `moss` and `tile`, Optimal leaves less than a thousand scan or unscan operations.

[7]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.

6 Conclusion and Future Work

We have designed and implemented RC, a dialect of C extended with safe regions. The overhead of safety is low (less than 11%) on all but one benchmark (where it reaches 20%). Even with this overhead, RC programs perform competitively with malloc/free based programs (from 13% slower to 53% faster) on our benchmarks. Our main contributions are a type system for dynamically checked region systems that brings some structure to region-based programs and improves the performance of reference counting, and an algorithm for efficiently tracking reference counts for local variables when compiling to a high-level language.

There are still a number of issues open in RC. Our previous paper [GA98] did a detailed breakdown of reference count overheads by counting the cost of each reference count operation using the UltraSparc's cycle counters. We could not repeat this approach when compiling to C because the C compiler's instruction scheduling mixes the instructions for updating reference counts with the surrounding code. We plan to perform a detailed analysis of the costs of reference counting in RC using other methods.

Deleting a region is relatively expensive. We have implemented a version of RC where counts are kept of the number of references between every pair of regions. This approach makes deleting a region very efficient, however it does not scale to programs which use large numbers of regions simultaneously. We plan to investigate this and other approaches for reducing the cost of `deleteregion` further.

The ordering relation between regions in our region type system could be extended to a tree of regions. Reference counts would not be kept from region a to region b if $b \leq a$. This would allow the creation of subregions that could be deleted independently, or automatically when the parent region is deleted. It is not yet clear if this is a useful concept or if it can be implemented efficiently enough.

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.

References

[Bob80] Daniel G. Bobrow. Managing re-entrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980.

[BZ93] David A. Barrett and Benjamin 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.

[CLR90] Thomas H Cormen, Charles E Leiserson, and Ronald L Rivest. *Introduction to Algorithms*, chapter 27. MIT Press, Cambridge, Mass., 1990.

[CWM99] Karl Crary, David Walker, and Greg 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, San Antonio, Texas*, pages 262–275, New York, NY, January 1999. ACM.

[FH95] Chris W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1995.

[GA98] D. Gay and A. Aiken. Memory Management with Explicit Regions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 313–323, June 1998.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, October 1988. Preliminary version in Proc. 18th Annual ACM Symposium on the Theory of Computing, pages 136–146, 1986.

[Han90] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software Practice and Experience*, 20(1):5–12, January 1990.

[IY90] Yuuji Ichisugi and Akinori Yonezawa. Distributed garbage collection using group reference counting. In *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, October 1990.

[Joh75] S. C. Johnson. YACC: Yet another compiler compiler. *Computing Science TR*, 32, 1975.

[Ros67] D. T. Ross. The AED free storage package. *Communications of the ACM*, 10(8):481–492, August 1967.

[SO96] David Stoutamire and Stephen Omohundro. The Sather 1.1 Specification. Technical Report TR-96-012, International Computer Science Institute, Berkeley, CA, August 1996.

[Sto97] D. Stoutamire. *Portable, Modular Expression of Locality*. PhD thesis, University of California at Berkeley, 1997.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, February 1997.

[Vo96] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. *Software Practice and Experience*, 26(3):357–374, March 1996.

[Wil92] Paul 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, September 1992. Springer-Verlag.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David 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, September 1995. Springer-Verlag.

A Soundness Proof

Some preliminary lemmas:

Lemma A.1 If $R//\mathrm{fr}(\tau) = R'//\mathrm{fr}(\tau)$ then $v : \tau$ is consistent with H under R iff $v : \tau$ is consistent with H under R'.
Proof: obvious from the definition of consistency.

Lemma A.2 $v : \tau$ is consistent with H under $R[\rho = R\sigma]$ iff $v : \tau[\sigma/\rho]$ is consistent with H under R.
Proof: obvious from the definition of consistency.

Lemma A.3 Let τ be a type and $\rho_1, \dots, \rho_m$ be distinct abstract regions with $\mathrm{fr}(\tau) \subseteq \{\rho_1, \dots, \rho_m\}$. $v : \tau$ is consistent with H under $[\rho_1 = R\sigma_1, \dots, \rho_m = R\sigma_m]$ iff $v : \tau[\sigma_1/\rho_1, \dots, \sigma_m/\rho_m]$ is consistent with H under R.
Proof: follows from lemmas A.2 and A.1.

Lemma A.4 Let R be an abstract region map, $\textsc{c}$ a constraint set with domain $\{\rho_1, \dots, \rho_m\} \cup C_R$. Then R is consistent with $\textsc{c}[\sigma_1/\rho_1, \dots, \sigma_m/\rho_m]$ iff $[\rho_1 = R\sigma_1, \dots, \rho_m = R\sigma_m]$ is consistent with $\textsc{c}$.
Proof: obvious.

Lemma A.5 *Constraint set closure properties:* Let R be an abstract region map, $\textsc{c}$, $\textsc{d}$ be constraint sets, $\sigma, \sigma_1, \sigma_2$ be region expressions and $\{\sigma, \sigma_1, \sigma_2\} \subseteq \mathrm{dom}(R) = \mathrm{dom}(\textsc{c}) = \mathrm{dom}(\textsc{d})$. Then:

- If R is consistent with $\textsc{c}$ and $\textsc{d}$ then R is consistent with $\textsc{c} \sqcup \textsc{d}$.
- If R is consistent with $\textsc{c}$ and $R\sigma \neq 0$ then R is consistent with $\textsc{c}[\sigma \neq \bot]$.
- If R is consistent with $\textsc{c}$ and $R\sigma_1 \preceq R\sigma_2$ then R is consistent with $\textsc{c}[\sigma_1 \leq \sigma_2]$.
- If R is consistent with $\textsc{c}$ then $R[\rho = \_]$ is consistent with $\textsc{c}[\neg\rho]$.

Proof: constraint set properties match those of the $\preceq$ partial order.

Lemma A.6 If $v : \tau$ is consistent with H under R then $v : \tau$ is partially consistent with H under R.
Proof: obvious.

Lemma A.7 If $v : \exists\rho \leq \sigma.\tau$ is consistent with H under R, r is such that $r \preceq R[\rho = r]\sigma$ and $v : \tau$ is partially consistent with H under $R[\rho = r]$ then $v : \tau$ is consistent with H under $R[\rho = r]$.
Proof: obvious from definition of consistency and partial consistency (the abstract region map used for checking the consistency of fields of objects does not depend on the instantiation of quantified variables)

Lemma A.8 *Assignability* If:

- $\textsc{c}, L \vdash \tau_1 \leftarrow \tau_2, \textsc{c}', L'$
- $< H, R, v, \tau_1, \tau_2 > \rightsquigarrow R'$,
- $v : \tau_2$ is consistent with H under R
- R is consistent with $\textsc{c}$

then $v : \tau_1$ is consistent with H under R', R' is consistent with $\textsc{c}'$ and $R//L = R'//L$. Additionally, the (a_inst_unsafe) assignment rule is not used in the reduction.
Proof: By induction on the structure of the evaluation of the type assignment. The proof considers each reduction rule in turn (each case starts with the reduction and type checking rules).

- $$\frac{< H, R, v, \tau[\sigma'/\rho], \tau' > \rightsquigarrow R'}{< H, R, v, \exists\rho \leq \sigma.\tau, \tau' > \rightsquigarrow R'} \qquad \frac{\sigma' \in L \quad \sigma[\sigma'/\rho] \in L \quad \textsc{c} \vdash \sigma' \leq \sigma[\sigma'/\rho] \quad \textsc{c}, L \vdash \tau[\sigma'/\rho] \leftarrow \tau', \textsc{c}', L'}{\textsc{c}, L \vdash \exists\rho \leq \sigma.\tau \leftarrow \tau', \textsc{c}', L'}$$
  By induction $v : \tau[\sigma'/\rho]$ is consistent with H under R', R' is consistent with $\textsc{c}'$ and $R//L = R'//L$. By lemma A.2 $v : \tau$ is consistent with H under $R'[\rho = R'\sigma']$. Also $R'\sigma' \preceq R'[\rho = R'\sigma']\sigma$ (from $R//L = R'//L$, $\sigma' \in L$, $\sigma[\sigma'/\rho] \in L$ and R consistent with $\textsc{c}$). Therefore $v : \exists\rho \leq \sigma.\tau$ is consistent with H under R'.

- $$\frac{\begin{array}{c}\text{exists } r \in A_H \text{ such that } v : \tau' \text{ is partially consistent with } H \text{ under } R[\rho' = r] \\ r \preceq R[\rho' = r]\sigma' \qquad < H, R[\rho = r], v, \tau, \tau'[\rho/\rho'] > \rightsquigarrow R'\end{array}}{< H, R, v, \tau, \exists\rho' \leq \sigma'.\tau' > \rightsquigarrow R'}$$

  $$\frac{\rho \notin L \quad \rho \in \mathrm{dom}(\textsc{c}) \quad \textsc{c}[\neg\rho][\rho \leq \sigma'[\rho/\rho']], L \cup \{\rho\} \vdash \tau \leftarrow \tau'[\rho/\rho'], \textsc{c}', L'}{\textsc{c}, L \vdash \tau \leftarrow \exists\rho' \leq \sigma'.\tau', \textsc{c}', L'}$$
  By hypothesis, $v : \exists\rho' \leq \sigma'.\tau'$ is consistent with H under R so (lemma A.7) $v : \tau'$ is consistent with H under $R[\rho' = r]$. By lemmas A.2 and A.1 $v : \tau'[\rho/\rho']$ is consistent with H under $R[\rho = r]$. By lemma A.5 $R[\rho = r]$ is consistent with $\textsc{c}[\neg\rho][\rho \leq \sigma'[\rho/\rho']]$. By induction, $v : \tau$ is consistent with H under R', R' is consistent with $\textsc{c}'$ and $R//L = R'//L$.

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

  By hypothesis, $v : \exists\rho' \leq \sigma'.\tau'$ is consistent with H under R so (lemma A.6) $v : \exists\rho' \leq \sigma'.\tau'$ is partially consistent with H under R. Therefore there does exist $r \in A_H$ such that $r \preceq R[\rho' = r]\sigma'$
  and $v : \tau'$ is partially consistent with H under $R[\rho' = r]$, a contradiction. Therefore this (a_inst_unsafe) rule can never be applied.

- $$\frac{}{< H, R, v, \texttt{region}@\sigma, \texttt{region}@\sigma' > \rightsquigarrow R[\sigma = R\sigma']} \qquad \frac{C, L \vdash \sigma \leftarrow \sigma', C', L'}{C, L \vdash \texttt{region}@\sigma \leftarrow \texttt{region}@\sigma', C', L'}$$

  $$\frac{\sigma \in L \quad C \vdash \sigma = \sigma'}{C, L \vdash \sigma \leftarrow \sigma', C, L} \qquad \frac{\sigma \notin L}{C, L \vdash \sigma \leftarrow \sigma', C[\neg\sigma][\sigma = \sigma'], L \cup \{\sigma\}}$$

  If $\sigma \in L$, then by consistency of R with C, $R\sigma = R\sigma'$ so $R[\sigma = R\sigma'] = R$. Therefore $v : \texttt{region}@\sigma'$ is consistent with H under $R[\sigma = R\sigma']$.

  If $\sigma \notin L$, then $R[\sigma = R\sigma']//L = R//L$. By lemma A.5 $R[\sigma = R\sigma']$ is consistent with $C[\neg\sigma][\sigma = \sigma']$ and $v : \texttt{region}@\sigma'$ is consistent with H under $R[\sigma = R\sigma']$.

- $$\frac{R_1 = R[\sigma = R\sigma'] \qquad R_{i+1} = R_i[\sigma_i = R_i\sigma'_i]}{< H, R, v, T[\sigma_1, \ldots, \sigma_m]@\sigma, T[\sigma'_1, \ldots, \sigma'_m]@\sigma' > \rightsquigarrow R_{m+1}}$$

  $$\frac{C, L \vdash \sigma \leftarrow \sigma', C_1, L_1 \qquad C_i, L_i \vdash \sigma_i \leftarrow \sigma'_i, C_{i+1}, L_{i+1}}{C, L \vdash T[\sigma_1, \ldots, \sigma_m]@\sigma \leftarrow T[\sigma'_1, \ldots, \sigma'_m]@\sigma', C_{m+1}, L_{m+1}}$$

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

Theorem A.9 *Soundness:* If

- $C, L \vdash s, C'$
- $< H, E, R, s > \rightsquigarrow < H', E', R' >$
- Variables $x_1 : \tau_1, \ldots, x_n : \tau_n$ are live before s
- Variables $x'_1 : \tau'_1, \ldots, x'_m : \tau'_m$ are live after s
- R is consistent with C
- $E(x_1) : \tau_1, \ldots, E(x_n) : \tau_n$ are consistent with H under R
- $w_1 : \alpha_1, \ldots, w_l : \alpha_l$ are consistent with H under Q

then

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

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

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

In these rules we will thus only show that R' is consistent with C', $R//L = R'//L$, and assigned variables are consistent with H under R'.
The fact that (a_inst_unsafe) is not used in the reduction follows from lemma A.8, used in all cases where semantic reduction rules invoke the semantic assignment rules. This fact will not be repeated in the cases below.

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

$$\frac{c, L \vdash s_1, c' \quad c', L_{s_2} \vdash s_2, c''}{c, L \vdash s_1; s_2, c''}$$

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

- $$\frac{E(x) = 0 \quad <H,E,R,s_2> \rightsquigarrow <H',E',R'>}{<H,E,R,\texttt{if}\ x\ s_1\ s_2> \rightsquigarrow <H',E',R'>} \qquad \frac{c, L_{s_1} \vdash s_1, c' \quad c, L_{s_2} \vdash s_2, c''}{c, L \vdash \texttt{if}\ x\ s_1\ s_2, c' \sqcap c''}$$

The live variables before s_1 are a subset of those before the `if`, so by induction, we conclude that R' is consistent with c', that variables live after s_1 (which are the same as those after the `if`) are consistent with H' under R', and that $w_1 : \alpha_1, \ldots, w_l : \alpha_l$ are consistent with H' under Q. As $c' \sqcap c'' \leq c''$ this case concludes. The $E(x) \neq 0$ case is essentially identical.

- $$\frac{E(x) \neq 0 \quad <H,E,R,s> \rightsquigarrow <H',E',R'> \quad <H',E',R',\texttt{while}\ x\ s> \rightsquigarrow <H'',E'',R''>}{<H,E,R,\texttt{while}\ x\ s> \rightsquigarrow <H'',E'',R''>}$$

$$\frac{c', L_s \vdash s, c'' \quad c' = c \sqcap c''}{c, L \vdash \texttt{while}\ x\ s, c'}$$

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

- $$\frac{E(x) = 0}{<H,E,R,\texttt{while}\ x\ s> \rightsquigarrow <H,E,R>} \qquad \frac{c', L_s \vdash s, c'' \quad c' = c \sqcap c''}{c, L \vdash \texttt{while}\ x\ s, c'}$$

The live variables after the `while` are a subset of those before it, and $c' \leq c$ so this case concludes.

- $$\frac{v_0 : \tau_0 \quad v_1 : \tau_1 \quad <H,R,E(v_1),\tau_0,\tau_1> \rightsquigarrow R'}{<H,E,R,v_0 = v_1> \rightsquigarrow <H,E[v_0 = E(v_1)],R'>} \qquad \frac{c, L \vdash \tau_0 \leftarrow \tau_1, c', L'}{c, L \vdash v_0 = v_1, c'}$$

By assumption, $E(v_1) : \tau_1$ is consistent with H under R and R is consistent with c, so by lemma A.8, $E(v_1) : \tau_0$ is consistent with H under R', R' is consistent with c' and $R'//L = R//L$.

- $$\frac{\begin{array}{c} x_0 : \tau_0 \quad x_1 : T[\sigma_1, \ldots, \sigma_m]@\sigma \quad \texttt{struct}\ T[\rho_1, \ldots, \rho_m]\{\ldots, f_i : \tau_i, \ldots\} \\ H(E(x_1)) = ((r_0, \ldots), (v_1, \ldots, v_n)) \quad <H,R,v_i,\tau_0,\tau_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m]> \rightsquigarrow R' \end{array}}{<H,E,R,x_0 = x_1.f_i> \rightsquigarrow <H,E[x_0 = v_i],R'>}$$

$$\frac{c[\sigma \neq \bot], L \vdash \tau_0 \leftarrow \tau_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m], c', L'}{c, L \vdash x_0 = x_1.f_i, c'}$$

From the definition of a heap, $r_0 \neq 0$ so by consistency of $E(x_1)$ with H under R, $R\sigma = r_0 \neq 0$. Therefore by lemma A.5 R is consistent with $c[\sigma \neq \bot]$. Also $v_i : \tau_i$ is consistent with H under $[\rho_1 = R\sigma_1, \ldots, \rho_m = R\sigma_m]$, so (lemma A.3) $v_i : \tau_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m]$ is consistent with H under R. By lemma A.8 $v_i : \tau_0$ is consistent with H under R', R' is consistent with c' and $R'//L = R//L$.

- $$\frac{\begin{array}{c} x_1 : T[\sigma_1, \ldots, \sigma_m]@\sigma \quad \texttt{struct}\ T[\rho_1, \ldots, \rho_m]\{\ldots, f_i : \tau_i, \ldots\} \quad x_2 : \tau_2 \\ H(E(x_1)) = ((r_0, \ldots, r_m), (v_1, \ldots, v_n)) \quad <H,R,E(x_2),\tau_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m],\tau_2> \rightsquigarrow R' \\ o = ((r_0, \ldots, r_m), (v_1, \ldots, v_{i-1}, E(x_2), v_{i+1}, \ldots, v_n)) \quad H' = H[E(x_1) = o] \end{array}}{<H,E,R,x_1.f_i = x_2> \rightsquigarrow <H',E,R'>}$$

$$\frac{c[\sigma \neq \bot], L \vdash \tau_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m] \leftarrow \tau_2, c', L'}{c, L \vdash x_1.f_i = x_2, c'}$$

From the definition of a heap, $r_0 \neq 0$ so by consistency of $E(x_1)$ with H under R, $R\sigma = r_0 \neq 0$. Therefore by lemma A.5 R is consistent with $c[\sigma \neq \bot]$. Also $E(x_2) : \tau_2$ is consistent with H under R, so by lemma A.8 $E(x_2) : \tau_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m]$ is consistent with H under R', R' is consistent with c' and $R//L = R'//L$. By lemma A.3 $E(x_2) : \tau_i$ is consistent with H under $[\rho_1 = R'\sigma_1, \ldots, \rho_m = R'\sigma_m]$. Also $L = L'$ as $\{\sigma_1, \ldots, \sigma_m\} \subseteq L$, so $R = R'$.

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

We consider these variables and the w_i's together by showing that there is no value $v : \tau$ consistent with H under some abstract region map P and inconsistent with H' under P.

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

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

- $v : \tau$ is partially inconsistent with H' (so also with H) under P. But $v : \tau$ is consistent with H under P, a contradiction.
- There exists $w : U[\ldots]@\_$ reachable in H' from v such that $H'(w) = ((s_0, \ldots, s_p), (w'_1, \ldots, w'_q))$, $\texttt{struct}\ U[\rho'_1, \ldots, \rho'_p]\{f_1 : \tau'_1, \ldots, f_q : \tau'_q\}$ and $w'_k : \tau'_k$ is partially inconsistent with H' under $P' = [\rho'_1 = s_1, \ldots, \rho'_q = s_q]$. So $w'_k : \tau'_k$ is partially inconsistent with H under P'. Note also that $H(w) = ((s_0, \ldots, s_p), \ldots)$ and w must be reachable in H from some some $v' : \tau'$ (either the value of a live variable or one of $w_1, \ldots, w_l$), with $v' : \tau'$ consistent with H under P. There are again two cases:
  * If $w \neq E(x_1)$ or $k \neq i$ (i.e., y_k is not the assigned field) then $w'_k : \tau'_k$ is not partially inconsistent with H under P', a contradiction.
  * If $w = E(x_1)$ and $k = i$ (i.e., we are considering the assigned field) we saw above that $E(x_2) = w'_k : \tau_i = \tau'_k$ is consistent with H under $[\rho_1 = R\sigma_1, \ldots, \rho_m = R\sigma_m]$. By the consistency of $E(x_1) : T[\sigma_1, \ldots, \sigma_m]@\sigma$ with R we conclude $P' = [\rho_1 = R\sigma_1, \ldots, \rho_m = R\sigma_m]$ so $w'_k : \tau'_k$ is not partially inconsistent with H under P', a contradiction.

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

We show that $R//L = R[\sigma_0 = 0]//L$:

- $\sigma_0 \notin L$: obvious.
- $\sigma_0 \in L$: from the assignment rules we get $\textsc{c} \vdash \sigma_0 = \bot$ and from the consistency of R with $\textsc{c}$ we get $R\sigma_0 = 0$, so $R//L = R[\sigma_0 = 0]//L$.

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

Consistency of $R[\sigma_0 = 0]$ with $\textsc{c}'$:

- $\sigma_0 \in L$: from the assignment rules it is easy to see that $\textsc{c}' \leq \textsc{c}$ so $R[\sigma_0 = 0] = R$ is consistent with $\textsc{c}'$.
- $\sigma_0 \notin L$: from lemma A.5 we conclude that $R[\sigma_0 = 0]$ is consistent with $\textsc{c}[\neg\sigma_0][\sigma_0 = \bot]$. As in the first case, $\textsc{c}' \leq \textsc{c}[\neg\sigma_0][\sigma_0 = \bot]$, so $R[\sigma_0 = 0] = R$ is consistent with $\textsc{c}'$.

- $$\frac{\begin{array}{cccc} x_0 : \tau_0 & x_i : \tau_i & x' : \texttt{region}@\sigma' & \texttt{struct}\ T[\rho_1, \ldots, \rho_m]\{f_1 : \tau'_1, \ldots, f_n : \tau'_n\} \\ \multicolumn{3}{c}{< H, R_i, E(x_i), \tau'_i[\sigma_1/\rho_1, \ldots, \sigma_m, \rho_m], \tau_i > \rightsquigarrow R_{i+1}} & < H[v = o], R_{n+1}, x, \tau_0, T[\sigma_1, \ldots, \sigma_m]@\sigma' > \rightsquigarrow R' \\ v \notin \mathrm{dom}(H) \wedge v \neq 0 & \multicolumn{2}{c}{H(E(x')) = (\texttt{region}, r, \_)} & o = ((r, R_{n+1}\sigma_1, \ldots, R_{n+1}\sigma_m), (E(x_1), \ldots, E(x_n))) \end{array}}{< H, E, R_1, x_0 = \texttt{new}\ T[\sigma_1, \ldots, \sigma_m](x_1, \ldots, x_n)@x' > \rightsquigarrow < H[v = o], E[x_0 = v], R' >}$$

$$\frac{\textsc{c}_i, L_i \vdash \tau'_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m] \leftarrow \tau_i, \textsc{c}_{i+1}, L_{i+1} \qquad \textsc{c}_{n+1}, L_{n+1} \vdash \tau_0 \leftarrow T[\sigma_1, \ldots, \sigma_m]@\sigma', \textsc{c}', L'}{\textsc{c}_1, L_1 \vdash x_0 = \texttt{new}\ T[\sigma_1, \ldots, \sigma_m](x_1, \ldots, x_n)@x', \textsc{c}'}$$

By induction on $1 \ldots n$ and lemmas A.8, A.3 and A.1 we conclude that $E(x_i) : \tau'_i$ are consistent with $[\rho_1 = R_{n+1}\sigma_1, \ldots, \rho_m = R_{n+1}\sigma_m]$, R_{n+1} is consistent with C_{n+1} and $R_{n+1}//L = R$. As $v \notin \mathrm{dom}(H)$ (so v is not a value in the environment or in any object in H), $v : T[\sigma_1, \ldots, \sigma_m]@\sigma$ is consistent with $H[v = o]$ under R_{n+1}, and all live variables after this statement are consistent with $H[v = o]$ under R_{n+1}. Similarly $w_1 : \alpha_1, \ldots, w_l : \alpha_l$ are consistent with $H[v = 0]$ under Q.

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

- $$\frac{H(E(x_0)) = ((r, \ldots), \_) \qquad H(E(x_1)) = ((r, \ldots), \_)}{< H, E, R, \texttt{chk}\ x_0 \leq x_1 > \rightsquigarrow < H, E, R >} \qquad \frac{x_0 : \mu_0@\sigma_0 \qquad x_1 : \mu_1@\sigma_1}{\textsc{c}, L \vdash \texttt{chk}\ x_0 \leq x_1, \textsc{c}[\sigma_0 \leq \sigma_1]}$$

By consistency of R with $\textsc{c}$ and of $E(x_0) : \mu_0@\sigma_0$, $E(x_1) : \mu_1@\sigma_1$ with H under R we conclude that $R\sigma_0 = r = R\sigma_1$. By lemma A.5 R is consistent with $C[\sigma_0 \leq \sigma_1]$.

- $$\frac{E(x_0) = 0}{< H, E, R, \texttt{chk}\ x_0 \leq x_1 > \rightsquigarrow < H, E, R >} \qquad \frac{x_0 : \mu_0@\sigma_0 \qquad x_1 : \mu_1@\sigma_1}{\textsc{c}, L \vdash \texttt{chk}\ x_0 \leq x_1, \textsc{c}[\sigma_0 \leq \sigma_1]}$$

By consistency of R with $\textsc{c}$ and of $0 = E(x_0) : \mu_0@\sigma_0$ with H under R we conclude that $R\sigma_0 = 0$. Therefore $R\sigma_0 \preceq R\sigma_1$ and by lemma A.5 R is consistent with $C[\sigma_0 \leq \sigma_1]$.

- $$\frac{H(E(x_0)) = ((R_0, \ldots), \ldots)}{< H, E, R, \texttt{chk}\ x_0 \leq R_0 > \leadsto < H, E, R >} \qquad \frac{x_0 : \mu_0 @ \sigma_0}{\textsc{c}, L \vdash \texttt{chk}\ x_0 \leq R_0, \textsc{c}[\sigma_0 \leq R_0]}$$

  By consistency of R with $\textsc{c}$ and of $E(x_0) : \mu_0 @ \sigma_0$ with H under R we conclude that $R\sigma_0 = R_0$. By lemma A.5 R is consistent with $C[\sigma_0 \leq R_0]$.

- $$\frac{E(x_0) = 0}{< H, E, R, \texttt{chk}\ x_0 \leq R_0 > \leadsto < H, E, R >} \qquad \frac{x_0 : \mu_0 @ \sigma_0}{\textsc{c}, L \vdash \texttt{chk}\ x_0 \leq R_0, \textsc{c}[\sigma_0 \leq R_0]}$$

  By consistency of R with $\textsc{c}$ and of $0 = E(x_0) : \mu_0 @ \sigma_0$ with H under R we conclude that $R\sigma_0 = 0$. Therefore $R\sigma_0 \preceq RR_0$ so by lemma A.5 R is consistent with $C[\sigma_0 \leq R_0]$.

- $$\frac{\begin{array}{c} f[\rho_1, \ldots, \rho_m][\textsc{d}](y_1 : \tau'_1, \ldots y_n : \tau'_n) : \tau', \textsc{d}' \ \texttt{is}\ [\rho'_1, \ldots, \rho'_p] y'_1 : \tau''_1, \ldots, y'_q : \tau''_q, s, y \\ E_f = [y_1 = E(x_1), \ldots, y_n = E(x_n), y'_1 = 0, \ldots, y'_q = 0] \qquad < H, E_f, R_f, s > \leadsto < H', E'_f, R'_f > \\ R_f = [\rho_1 = R_{n+1}\sigma_1, \ldots, \rho_m = R_{n+1}\sigma_m, \rho'_1 = 0, \ldots, \rho'_p = 0] \\ x_i : \tau_i \qquad < H, R_i, E(x_i), \tau'_i[\sigma_1/\rho_1, \ldots, \sigma_m, \rho_m], \tau_i > \leadsto R_{i+1} \\ \sigma \in F_{\tau'} \Rightarrow R_{\tau'}\sigma = R'_f\sigma \quad \sigma \notin F_{\tau'} \Rightarrow R_{\tau'}\sigma = R_{n+1}\sigma \quad < H', R_{\tau'}, E'_f(y), \tau_0, \tau'[\sigma_1/\rho_1, \ldots, \sigma_m, \rho_m] > \leadsto R' \end{array}}{< H, E, R_1, x_0 = f[\sigma_1, \ldots, \sigma_m](x_1, \ldots, x_n) > \leadsto < H', E[x_0 = E'_f(y)], R' >}$$

  $$\frac{\begin{array}{c} \textsc{c}_i, L_i \vdash \tau'_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m] \leftarrow \tau_i, \textsc{c}_{i+1}, L_{i+1} \qquad \textsc{d}[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m] \leq \textsc{c}_{n+1} \\ \textsc{c}_{n+1}//L_{n+1} \sqcup \textsc{d}'[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m], L_{n+1} \cup F_{\tau'} \vdash \tau_0 \leftarrow \tau'[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m], \textsc{c}', L' \\ \sigma_i \in F_{\tau'} \iff \rho_i \text{ free in } \tau' \text{ and for all } k, \rho_i \text{ not free in } \tau'_k \qquad F_{\tau'} \cap L_{n+1} = \emptyset \\ \textsc{d}, L_s \vdash s, \textsc{d}'' \qquad \textsc{d}' \leq \textsc{d}''//(\{\rho_1, \ldots, \rho_m\} \cup C_R) \end{array}}{\textsc{c}_1, L_1 \vdash x_0 = f[\sigma_1, \ldots, \sigma_m](x_1, \ldots, x_n), \textsc{c}'}$$

  By induction on $1 \ldots n$ we conclude that $\forall i. E(x_i) : \tau'_i[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m]$ is consistent with H under R_{n+1}, and R_{n+1} is consistent with $\textsc{c}_{n+1}$ and $\textsc{d}[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m]$. Therefore (lemmas A.1 and A.3) $\forall i. E(x_i) = E_f(y_i) : \tau'_i$ is consistent with H under R_f and R_f is consistent with $\textsc{d}$ (lemma A.4). By assumption, $y'_1, \ldots, y'_q$ are dead before s, so all live variables before s are consistent with H under R_f.

  Let N be the abstract regions in L_{n+1} ($N = L_{n+1} - C_R$). For simplicity of exposition, we assume $N \cap \mathrm{dom}(Q) = \emptyset$ (this is easily achieved by suitable renaming). We define Q' as $Q'\sigma = Q\sigma$ if $\sigma \in \mathrm{dom}(Q)$ and $Q'\sigma = R_{n+1}\sigma$ if $\sigma \in N$. Let the live variables before the function call be $x''_1 : \tau''_1, \ldots, x''_p : \tau''_p$.

  By induction, with extra values $w_1 : \alpha_1, \ldots, w_l : \alpha_l, E(x''_1) : \tau''_1, \ldots, E(x''_p) : \tau''_p$ consistent with H under Q', we conclude that R'_f is consistent with $\textsc{d}'$, $E'_f(y) : \tau'$ is consistent with H' under R'_f and $w_1 : \alpha_1, \ldots, w_l : \alpha_l, E(x''_1) : \tau''_1, \ldots, E(x''_p) : \tau''_p$ consistent with H' under Q'. From this we conclude that $w_1 : \alpha_1, \ldots, w_l : \alpha_l$ are consistent with H' under Q.

  Let F be the abstract regions free in $\tau'_1, \ldots, \tau'_n$. F is in every live abstract region set of f, so it is easy to check that $\forall \rho \in F. R'_f\rho = R_f\rho$. Therefore, $R_{\tau'}\sigma_i = R'_f\rho_i$, so (lemma A.4) $R_{\tau'}$ is consistent with $\textsc{d}'[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m]$. By lemmas A.1 and A.3 $E'_f(y) : \tau'[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m]$ is consistent with H' under $R_{\tau'}$. As $\forall \sigma \in L_{n+1}. R_{\tau'}\sigma = R_{n+1}\sigma$, $R_{\tau'}$ is consistent with $\textsc{c}_{n+1}//L_{n+1}$, therefore (lemma A.5) $R_{\tau'}$ is consistent with $C_{n+1}//L_{n+1} \sqcup \textsc{d}'[\sigma_1/\rho_1, \ldots, \sigma_m/\rho_m]$. By lemma A.8, $E[x_0 = E'_f(y)](x_0) = E'_f(y) : \tau_0$ is consistent with H' under R', R' is consistent with $\textsc{c}'$ and $R'//(L_{n+1} \cup F_{\tau'}) = R_{tau'}//(L_{n+1} \cup F_{\tau'})$.

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