Language Support for Regions

David Gay
Alex Aiken
University of California, Berkeley

- Traditional memory management:

	malloc	GC
Safety	-	+
Control	+	-
Ease of use	-	+
Space usage	+	-

- A different approach: regions

  safety and efficiency, expressiveness

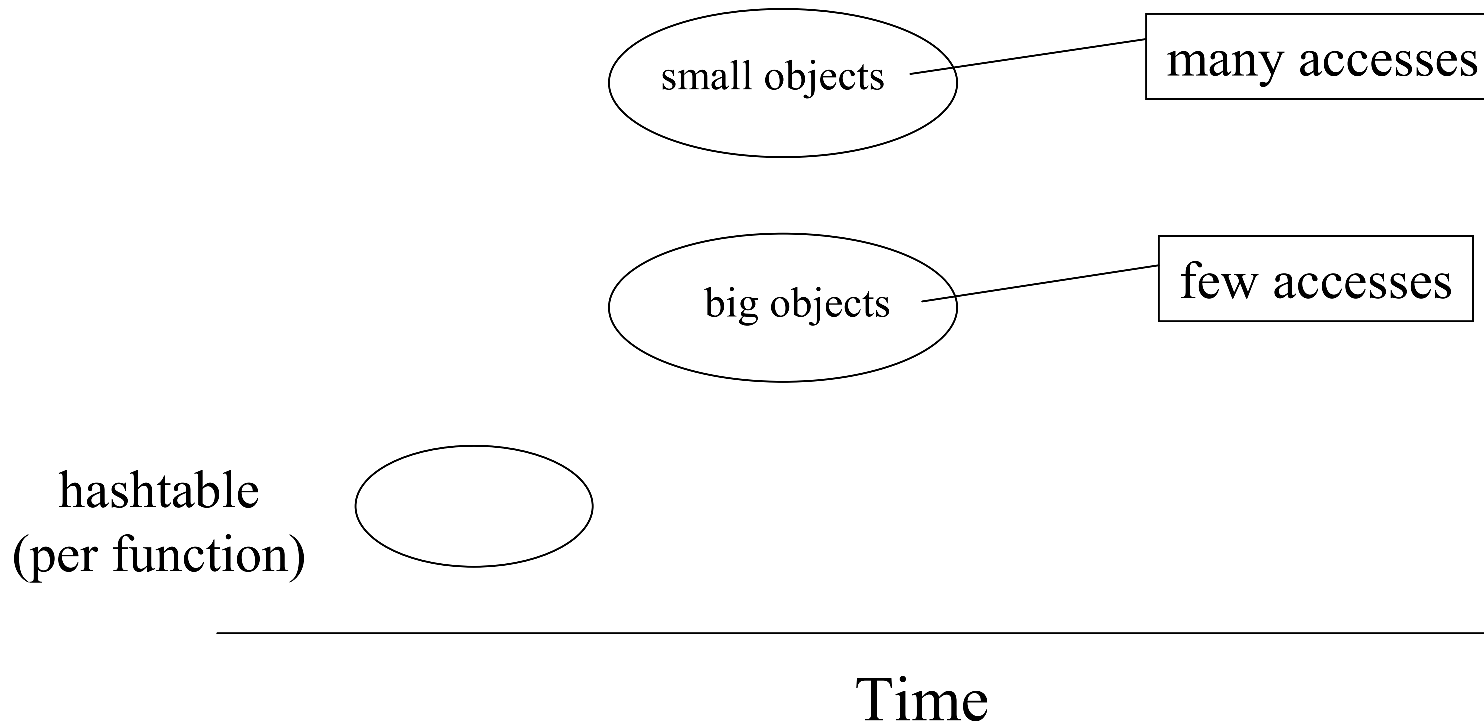
RC: C with Regions

- *Regions* represent unbounded areas of memory
- Objects are allocated in a given region
- Explicit deallocation of whole region

```
struct decls {
  char *name;
  struct decls *next;
} *dl = NULL;
region r = newregion();
while (...)
  dl = rcons(r, somename, dl);
work(dl);
deleteregion(r);
```

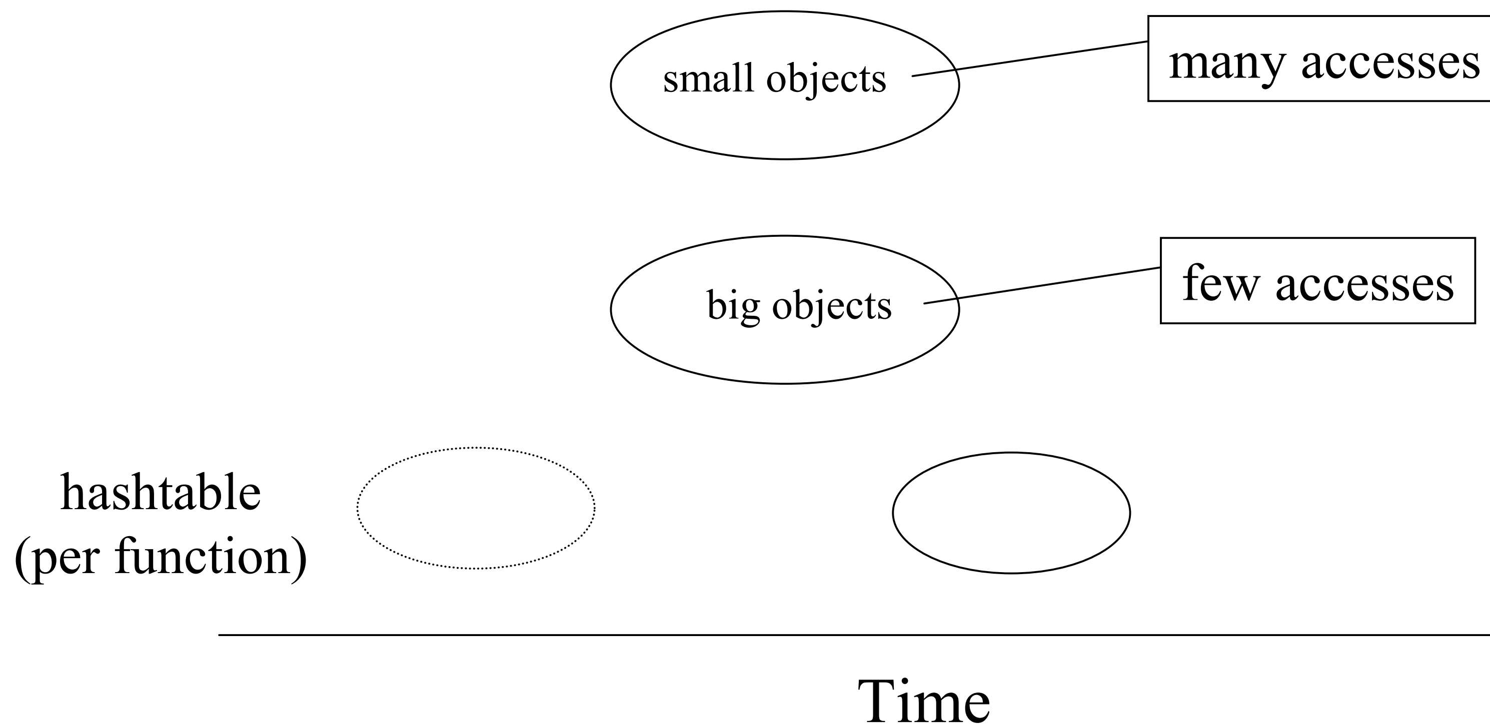
Region Example: moss

- plagiarism detection program
  - regions group related objects
  - region structure expresses spatial locality



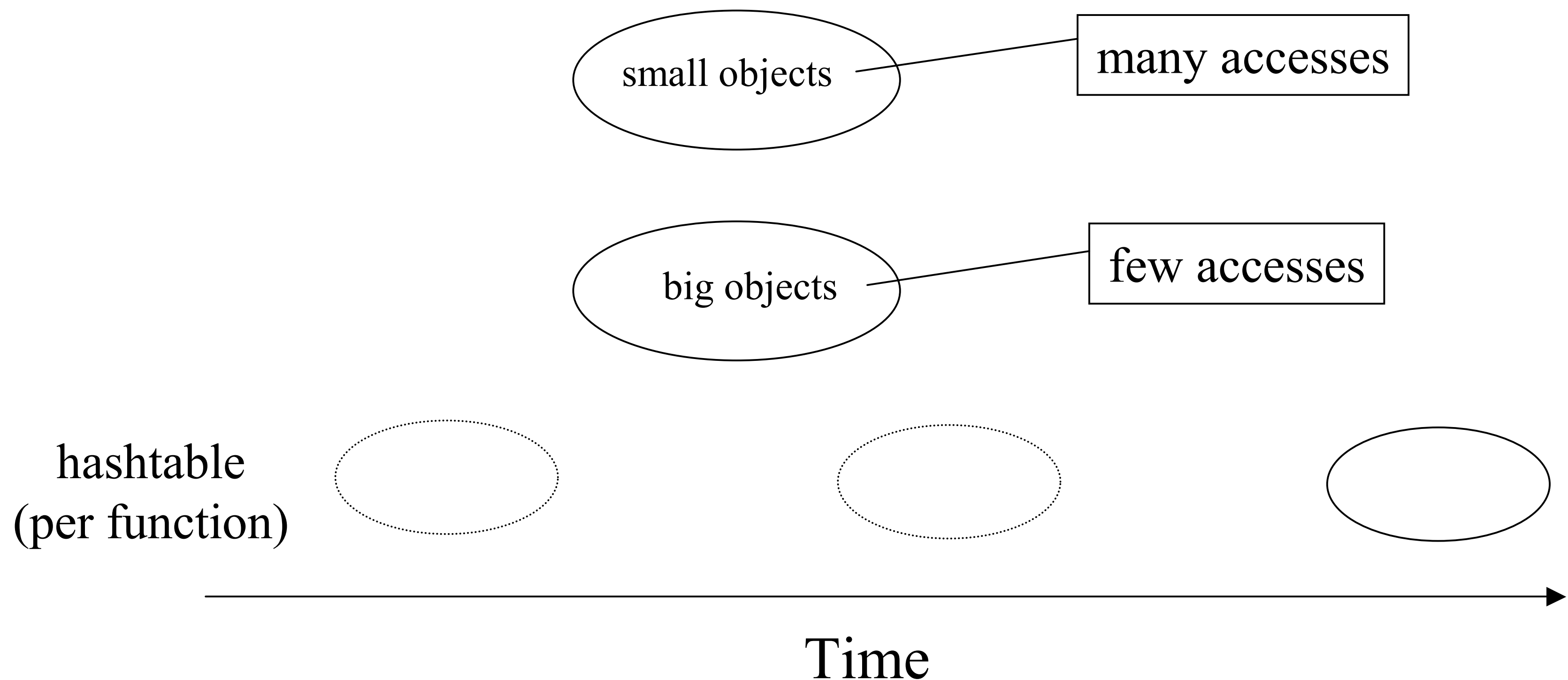
Region Example: moss

- plagiarism detection program
  - regions group related objects
  - region structure expresses spatial locality



Region Example: moss

- plagiarism detection program
  - regions group related objects
  - region structure expresses spatial locality

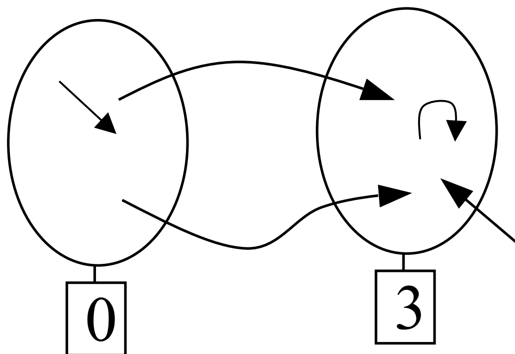


Contributions

- language design for region-based programming
  - memory safety
  - captures natural memory patterns found in applications
    - same region pointers
    - other qualifiers, sub regions: see paper
  - formalisation of these patterns
- efficient implementation
  - on our benchmarks: safety overhead $\leq$ 11% of runtime

RC: Reference Counted Regions

- Unrestricted use of regions
- Safety via reference-counting:
  - refcount(region r) = number of references to objects in r from outside r
  - deleteregion(r) fails if refcount(r) > 0

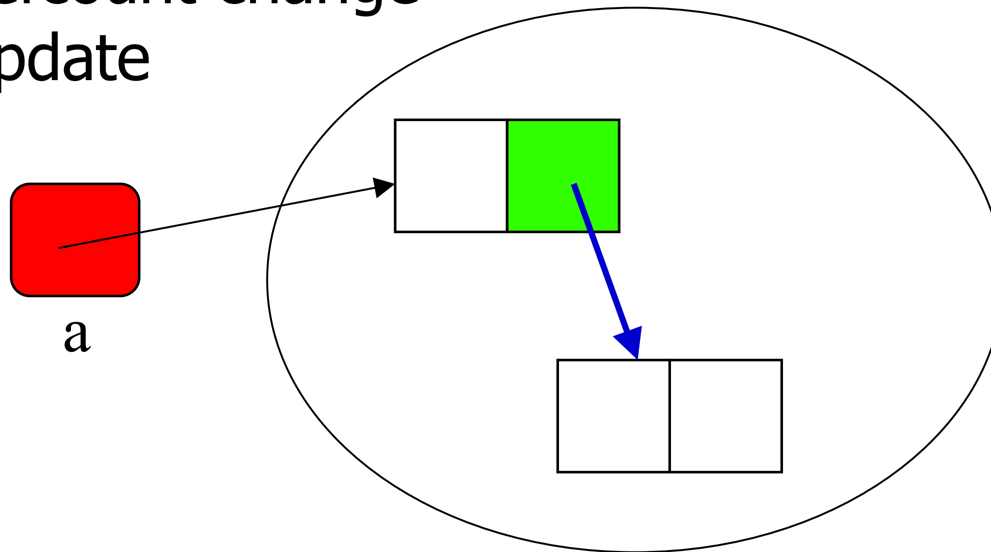


- Advantages:
  - low space cost
  - cycles allowed within single region

RC: sameregion

- A *sameregion* pointer is null or points to the same region as its container
  No refcount change on update

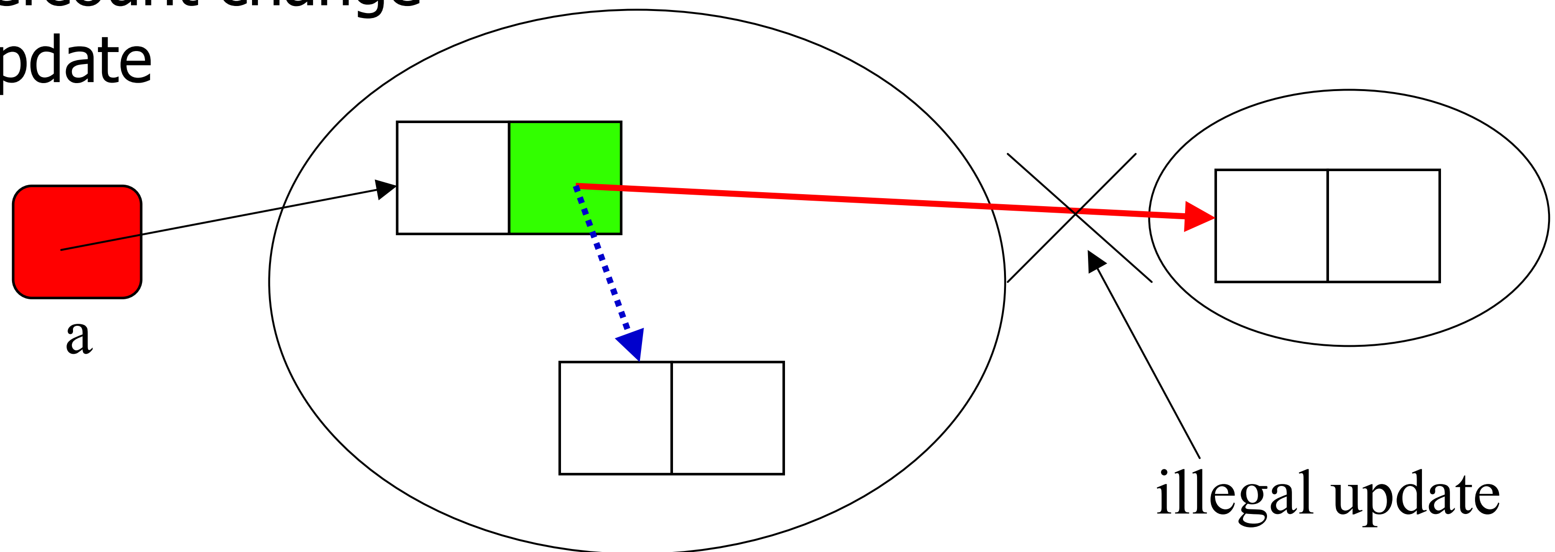
```
struct decls {
  char *name;
  struct decls *sameregion next;
} *a;
```



RC: sameregion

- A *sameregion* pointer is null or points to the same region as its container
  No refcount change on update

```
struct decls {
  char *name;
  struct decls *sameregion next;
} *a;
```

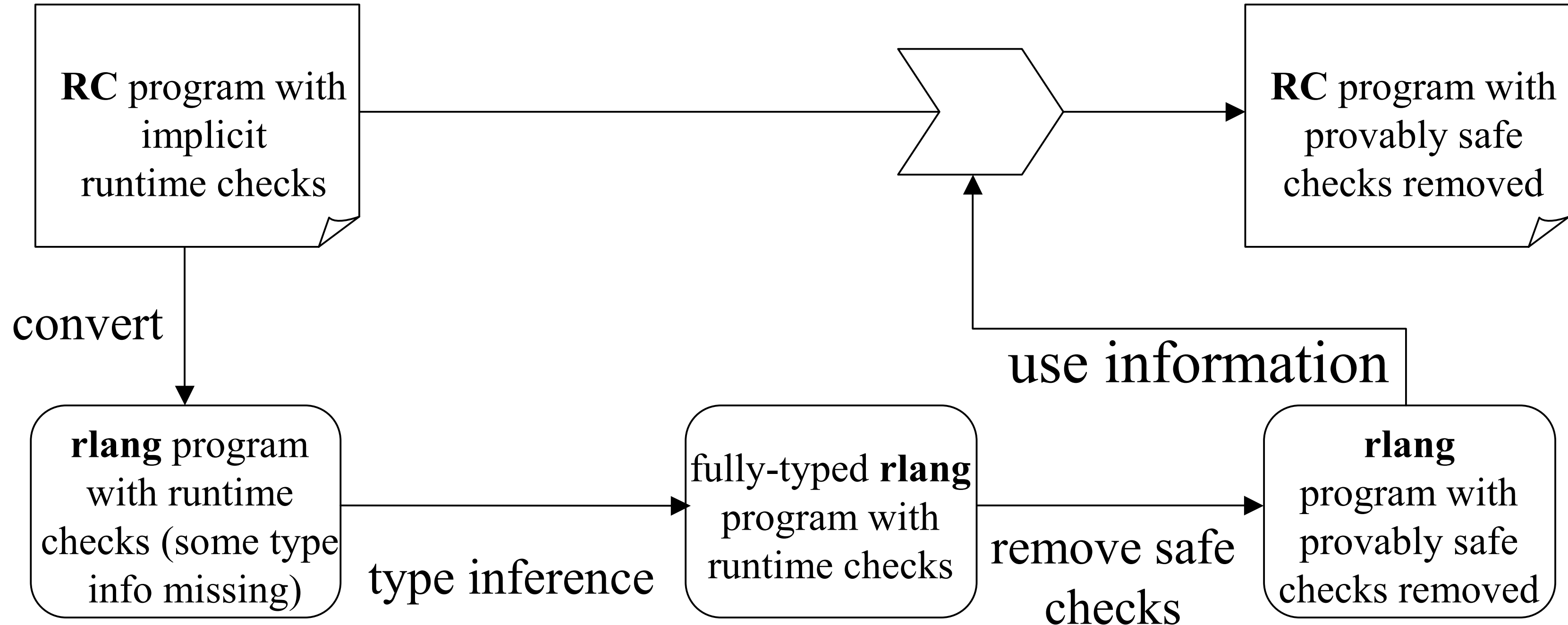


RC: Other Language Features

- See paper for details:
  - sub regions
  - traditional region
  - related type qualifiers: traditional, parentptr

RC: Compiling with Qualifiers

- rlang: simple C-like language with a region type system
  - formalises the meaning of type qualifiers
  - type-checking rlang allows removal of safe runtime checks for these qualifiers



Region Expressions

- Every pointer points to some region denoted by a *region expression* σ:
  - : the region of null pointers
  - R_T:the traditional region
  - ρ: a region variable

- Translation of sameregion:
  - ρ_1 is sameregion as ρ_2: $\rho_1 = \quad \vee \rho_1 = \rho_2$

rlang Types

$\sigma ::= \quad | R_T | \rho$ (region expressions)

$\forall \rho_1, \ldots, \rho_m$ struct T $\{ f_1 : \tau_1, \ldots, f_n : \tau_n \}$ (structures)

$\tau ::=$ region@σ | $T[\sigma_1, \ldots, \sigma_m]@\sigma$ | $\exists \rho / \delta . \tau$ (types)

$\delta ::= \sigma = \sigma \mid \delta \wedge \delta \mid \delta \vee \delta \mid \neg \delta \mid (\delta)$ (region properties)

```
struct D *list
```

→ list : $D[\rho_1]@\rho_1$

```
struct D {
  struct D *sameregion next;
  char *name;
}
```

→

$\forall \rho$ struct D {
    next : $\exists \rho' / \rho' = \quad \vee \rho' = \rho . D[\rho']@\rho'$,
    name : $\exists \rho' .$char*$[\rho']@\rho'$
}

rlang Type System Overview

- explicit runtime checks (chk δ)
  - allows checking every field assignment
- type rules are flow sensitive
  - summarises relations between region expressions as an expression δ at each program point
  - chk δ' redundant if $\delta \Rightarrow \delta'$
- Thm: type system is sound

RC: Implementation

- Based on a C-to-C compiler
- Efficient runtime:
  - page-based allocation in regions
  - low-cost reference counting
  - optimisation of local-variable reference counting
- Qualifiers:
  - runtime check elimination / efficient runtime checks

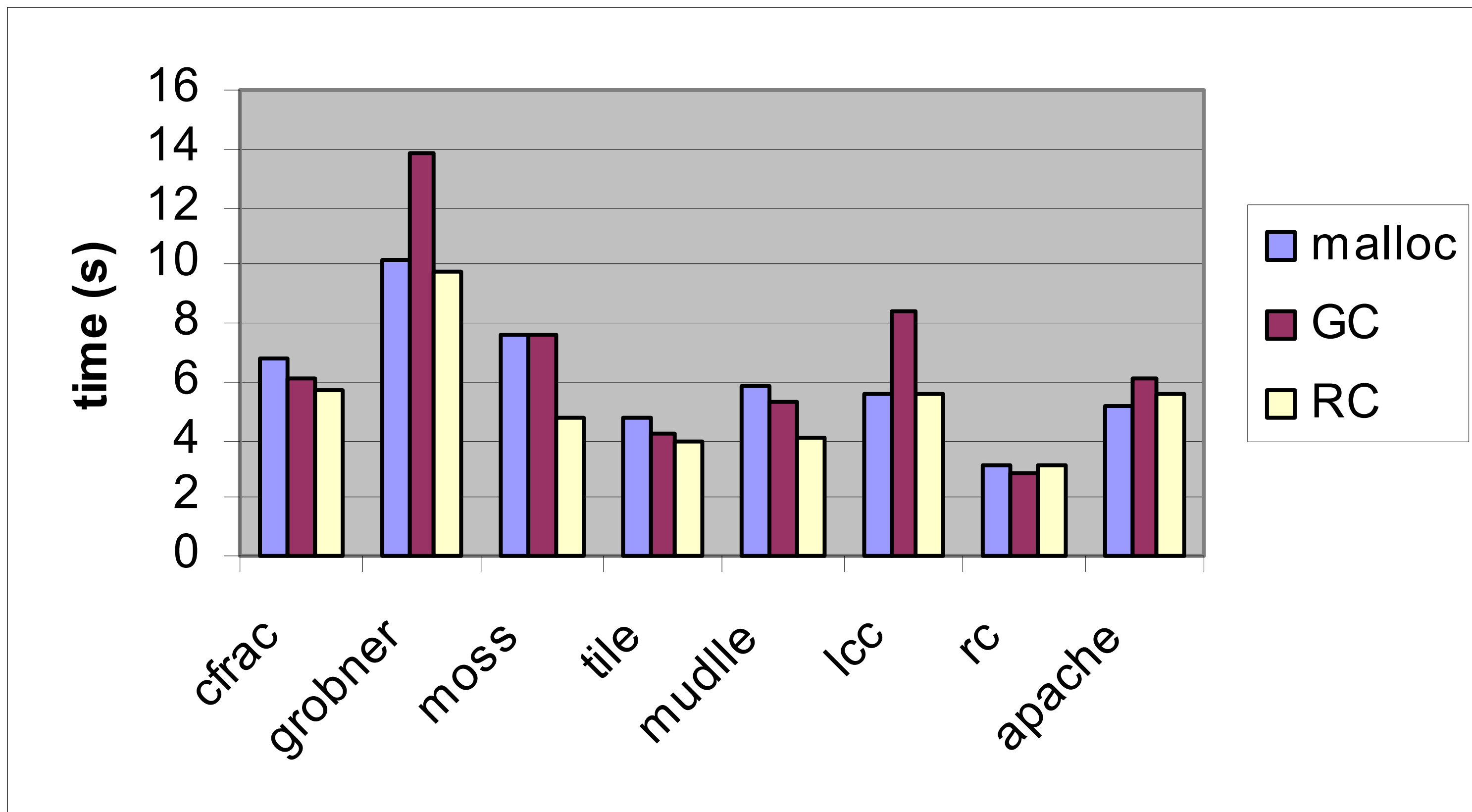
Results: Experimental Framework

- Machine: 333 MHz UltraSparc II, Solaris 2.7
- Benchmarks: 8 medium to large C programs
- Regions vs
  - malloc/free (Doug Lea v2.6.6)
  - conservative GC (Boehm-Weiser v5.3)
- C compiler: gcc 2.95.2
- Wall-clock execution time

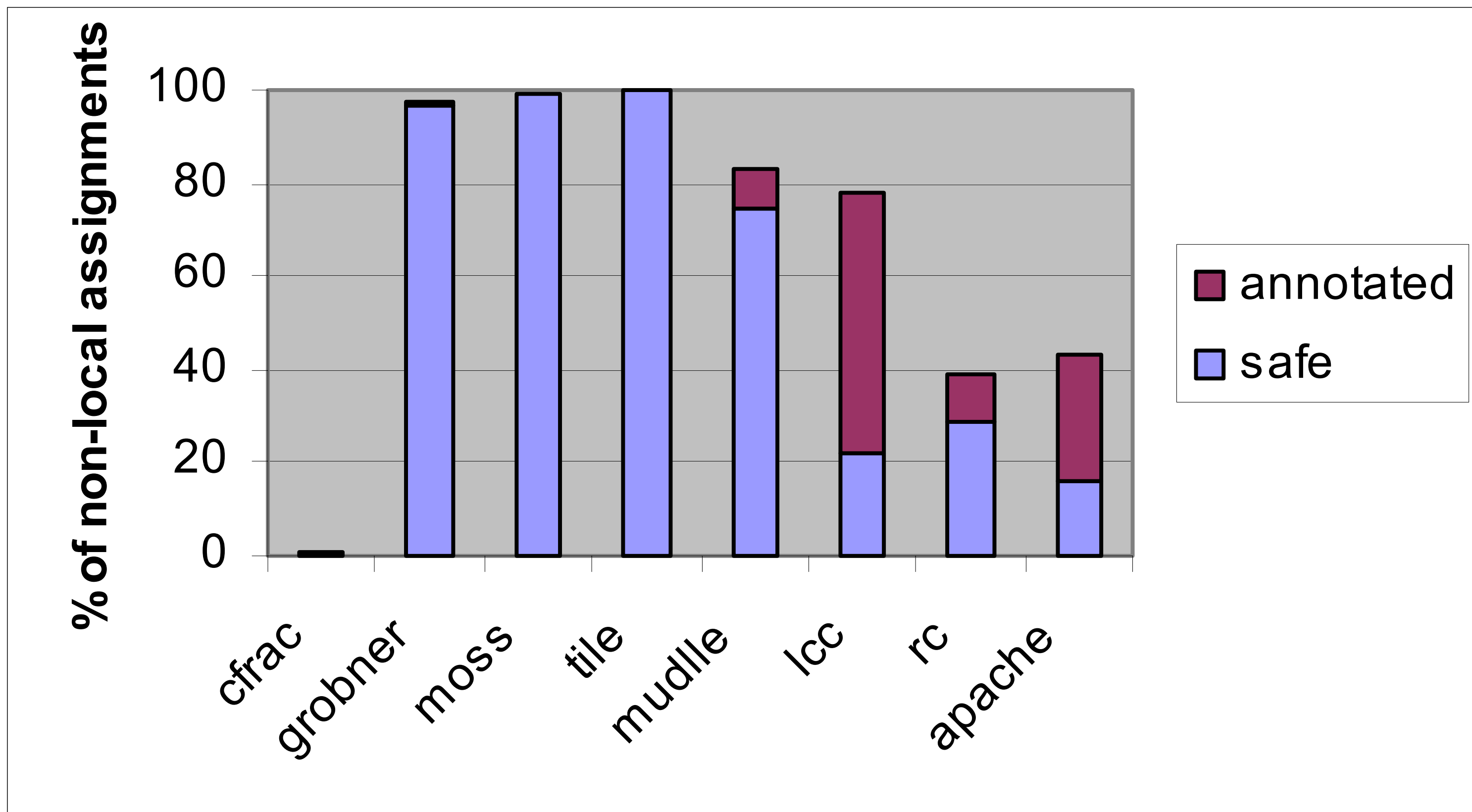
Results: Benchmark Summary

Name	*Lines*	*Allocations*	*Bytes*	*Live bytes*
cfrac	4203	3.8M	55MB	102KB
grobner	3219	6.0M	306MB	474KB
tile	926	0.01M	0.3MB	153KB
moss	2675	0.6M	6.2MB	2185KB
mudlle	5078	1.6M	22MB	210KB
lcc	12430	1.0M	54MB	4121KB
rc	22823	0.1M	4.6MB	4214KB
apache	62289	0.2M	30MB	78KB

Results: Execution Time



Results: Annotation Checking



Related Work: Regions

- Runtime libraries, e.g., for C:
  - +: no type restrictions
  - - : no safety
  - zones (D.T. Ross, 1967), vmalloc (K. Vo, 1996), arenas (D. Hanson, 1990), apache, gcc
- Region type systems:
  - +: compile-time safety
  - - : type and deleteregion placement restrictions
  - Region Inference: M. Tofte, J.-P. Talpin (1994), Calculus of Capabilities: Crary, Walker, Morrisett (1999), Alias Types: Walker, Morrisett (2000), Vault: Deline, Fähndrich (2001)

Related Work: Other Memory Management

	regions	malloc	GC
Safety	+	-	+
Control	++	+	-
Ease of use	=	-	+
Space usage	+	+	-
Time	+	+	+

Conclusion

- Regions are a nice programming model
- RC provides safety at low cost
- RC annotations are effective and often statically checkable
  - future: infer annotations ?
- Distribution of RC will be available July 1st at
  - http://www.cs.berkeley.edu/~dgay/rc.tar.gz
  - Tested on Solaris, Linux