Region-Based Memory Management in Cyclone

Dan Grossman

Cornell University

June 2002

Joint work with: Greg Morrisett, Trevor Jim (AT&T), Michael Hicks, James Cheney, Yanling Wang

Cyclone

- A safe C-level language
- Safe: Memory safety, abstract data types

  must forbid dereferencing dangling pointers
- C-Level: User controlled data representation and resource management

  cannot always resort to extra tags and checks

for legacy and low-level systems

Dangling pointers unsafe

```
void bad() {
 int* x;
 if(1){
   int  y;
   int* z = &y;
   x = z;
 }
 *x = 123;
}
```

- Access after lifetime "undefined"
- Notorious problem
- Re-user of memory cannot maintain invariants

High-level language solution:

- Language definition: infinite lifetimes
- Implementation: sound garbage collection (GC)

Cyclone memory management

- Flexible: GC, stack allocation, region allocation
- Uniform: Same library code regardless of strategy
- Static: no “has it been deallocated” run-time checks
- Convenient: few explicit annotations
- Exposed: users control lifetime of objects
- Scalable: all analysis intraprocedural
- Sound: programs never follow dangling pointers

The plan from here

- Cyclone regions
- Basic type system
  - Restricting pointers
  - Increasing expressiveness
  - Avoiding annotations
- Interaction with abstract types
- Experience
- Related and future work

Regions

- a.k.a. zones, arenas, …
- Every object is in exactly one region
- Allocation via a region *handle*
- All objects in a region are deallocated simultaneously (no **`free`** on an object)

An old idea with recent support in languages and implementations

Cyclone regions

- heap region: one, lives forever, conservatively GC'd
- stack regions: correspond to local-declaration blocks

```
{int x; int y; s}
```

- dynamic regions: scoped lifetime, but growable

```
region r {s}
```

- allocation: **`rnew(r,3)`**, where **`r`** is a *handle*
- handles are first-class
  - caller decides where, callee decides how much
  - no handles for stack regions

The big restriction

- Annotate all pointer types with a region name

    a (compile-time) type variable of region kind

- **`int*`r`** means "pointer into the region created by the construct that introduced **`` `r ``**"
  - heap introduces **`` `H ``**
  - **`L:`**… introduces **`` `L ``**
  - **`region r {`*`s`*`}`** introduces **`` `r ``**

    `r` has type **`` region_t<`r> ``**

So what?

Perhaps the scope of type variables suffices

```
void bad() {
 int*`?? x;
 if(1){
 L:{int     y;
    int*`L z = &y;
    x = z;
   }
 }
 *x = 123;
}
```

- What region name for type of x?
- `` `L `` is not in scope at allocation point
- good intuition for now
- but simple scoping does *not* suffice in general

The plan from here

- Cyclone regions
- Basic type system
  - Restricting pointers
  - Increasing expressiveness
  - Avoiding annotations
- Interaction with abstract types
- Experience
- Related and future work

Region polymorphism

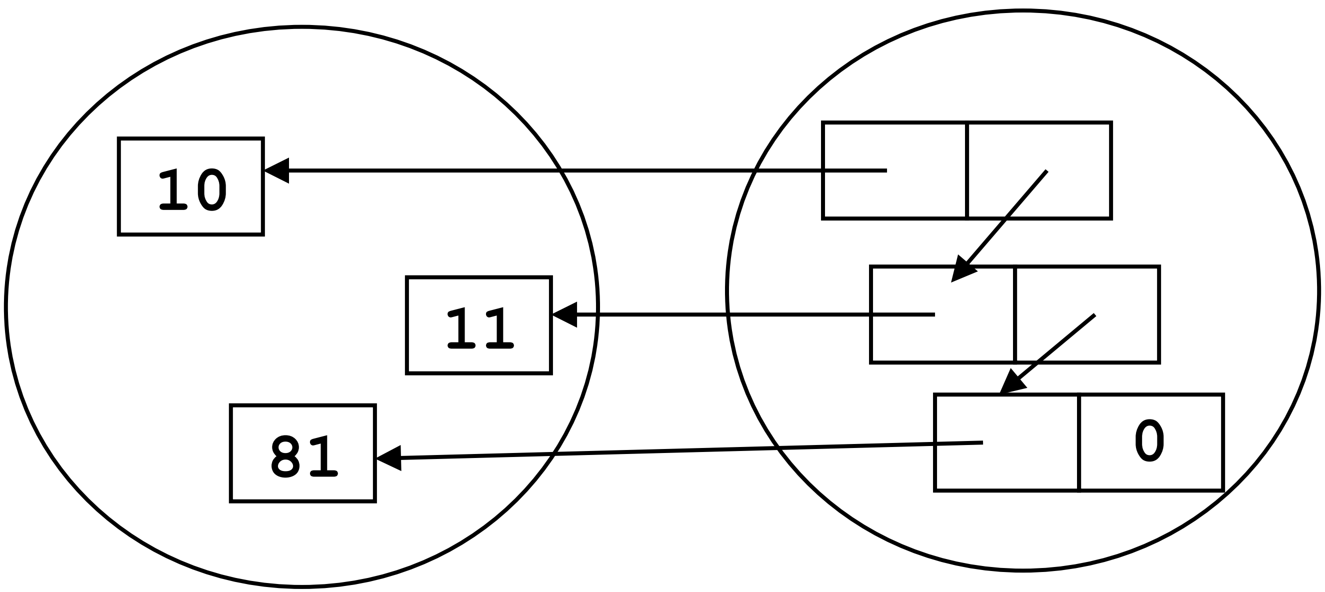
Use parametric polymorphism just like you would for other type variables

```
void swap<`r1,`r2>(int*`r1 x, int*`r2 y){
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

int*`r newsum<`r>(region_t<`r> r,
                  int x, int y){
  return rnew(r) (x+y);
}
```

Type definitions

```
struct ILst<`r1,`r2> {
  int*`r1 hd;
  struct ILst<`r1,`r2> *`r2 tl;
};
```



Region subtyping

If `p` points to an `int` in a region with name `` `r1 ``, is it ever sound to give `p` type `` int*`r2 ``?

- If so, let `` int*`r1 < int*`r2 ``
- Region subtyping is the outlives relationship

```
region r1 {… region r2 {…}…}
```

- LIFO makes subtyping common
- Function preconditions can include outlives constraints:

```
void f(int*`r1, int*`r2  :`r1 > `r2);
```

The plan from here

- Cyclone regions
- Basic type system
  - Restricting pointers
  - Increasing expressiveness
  - Avoiding annotations
- Interaction with abstract types
- Experience
- Related and future work

Who wants to write all that?

- Intraprocedural inference
  - Determine region annotation based on uses
  - Same for polymorphic instantiation
  - Based on unification (as usual)
  - So we don't need `L:`

- Rest is by defaults
  - Parameter types get fresh region names
    (default is region-polymorphic with no equalities)
  - Everything else gets `` `H ``
    (return types, globals, struct fields)

Example

You write:

```
void fact(int* result, int n) {
    int x = 1;
    if(n > 1) fact(&x,n-1);
    *result = x*n;
}
```

Which means:

```
void fact<`r>(int*`r result, int n) {
L:  int x = 1;
    if(n > 1) fact<`L>(&x,n-1);
    *result = x*n;
}
```

Annotations for equalities

```
void g(int*`r* pp, int*`r p) {
  *pp = p;
}
```

- Callee writes the equalities the caller must know
- Caller writes nothing

The plan from here

- Cyclone regions
- Basic type system
  - Restricting pointers
  - Increasing expressiveness
  - Avoiding annotations
- Interaction with abstract types
- Experience
- Related and future work

Existential types

- Programs need first-class abstract types

```
struct T {
   void (*f)(void*, int);
   void* env;
};
```

- We use an existential type:

```
struct T { <`a>  // ∃α…
  void (*f)(`a, int);
  `a env;
};
```

- `struct T mkT();` could make a dangling pointer!

Same problem occurs with closures or objects

Our solution

- "leak a region bound"

```
struct T<`r> { <`a> :regions(`a) > `r
   void (*f)(`a, int);
   `a env;
};
```

- Dangling pointers never dereferenced
- Really we have a powerful effect system, but
  - Without using ∃, no effect errors
  - With ∃, use region bounds to avoid effect errors
- See the paper

Region-system summary

- Restrict pointer types via region names
- Add polymorphism, constructors, and subtyping for expressiveness
- Well-chosen defaults to make it palatable
- A bit more work for safe first-class abstract types

- Validation:
  – Rigorous proof of type safety
  – 100KLOC of experience…

Writing libraries

- Client chooses GC, region, or stack
- Adapted OCaml libraries (List, Set, Hashtable, …)

```
struct L<`a,`r> {`a hd; struct L<`a,`r>*`r tl;};

typedef struct L<`a,`r>*`r l_t<`a,`r>;

l_t<`b,`r> rmap(region_t<`r>,`b f(`a),l_t<`a>);

l_t<`a,`r> imp_append(l_t<`a,`r>, l_t<`a,`r>);

void app(`b f(`a), l_t<`a>);

bool cmp(bool f(`a,`b), l_t<`a>, l_t<`b>);
```

Porting code

- about 1 region annotation per 200 lines
- regions can work well (mini web server without GC)
- other times LIFO is a bad match
- other limitations (e.g., stack pointers in globals)

Running code

- No slowdown for networking applications
- 1x to 3x slowdown for numeric applications
  - Not our target domain
  - Largely due to array-bounds checking (and we found bugs)
- We use the bootstrapped compiler every day
  - GC for abstract syntax
  - Regions where natural
  - Address-of-locals where convenient
  - Extensive library use

The plan from here

- Cyclone regions
- Basic type system
  - Restricting pointers
  - Increasing expressiveness
  - Avoiding annotations
- Interaction with abstract types
- Experience
- Related and future work

Related: regions

- ML Kit [Tofte, Talpin, et al], GC integration [Hallenberg et al]
  - full inference (no programmer control)
  - effect variables for ∃ (not at source level)
- Capability Calculus [Walker et al]
  - for low-level machine-generated code
- Vault [DeLine, Fähndrich]
  - restricted region aliasing allows “must deallocate”
- Direct control-flow sensitivity [Henglein et al.]
  - first-order types only
- RC [Gay, Aiken]
  - run-time reference counts for inter-region pointers
  - still have dangling stack, heap pointers

Related: safer C

- LCLint [Evans], metal [Engler et al]
  - sacrifice soundness for fewer false-positives
- SLAM [Ball et al], ESP [Das et al], Cqual [Foster]
  - verify user-specified safety policy with little/no annotation
  - assumes data objects are infinitely far apart
- CCured [Necula et al]
  - essentially GC (limited support for stack pointers)
  - better array-bounds elimination, less support for polymorphism, changes data representation
- Safe-C, Purify, Stackguard, …

Future work

- Beyond LIFO ordering
- Integrate more dynamic checking ("is this a handle for a deallocated region")
- Integrate threads
- More experience where GC is frowned upon

Conclusion

- Sound, static region-based memory management
- Contributions:
  - Convenient enough for humans
  - Integration with GC and stack
  - Code reuse (write libraries once)
  - Subtyping via outlives
  - Novel treatment of abstract types

http://www.cs.cornell.edu/projects/cyclone

http://www.research.att.com/projects/cyclone