Summer School on
Language-Based Techniques for Integrating with the External World

Types for Safe C-Level Programming Part 3: Basic Cyclone-Style Region-Based Memory Management

Dan Grossman
University of Washington
26 July 2007

C-level Quantified Types

- As usual, a type variable hides a type's identity
  - Still usable because multiple in same scope hide the same type
- For code reuse and abstraction
- But so far, if you have a τ***** (and τ has known size), then you can dereference it
  - If the pointed-to location has been *deallocated*, this is broken ("should get stuck")
  - Cannot happen in a garbage-collected language
- All this type-variable stuff will help us!

Safe Memory Management

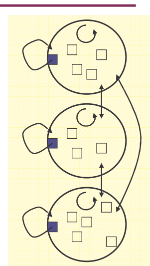
- Accessing recycled memory violates safety (*dangling pointers*)
- *Memory leaks* crash programs
- In most safe languages, objects conceptually *live forever*
- Implementations use *garbage collection*
- Cyclone needs *more options,* without sacrificing safety/performance

The Selling Points

- Sound: programs never follow dangling pointers
- Static: no "has it been deallocated" run-time checks
- Convenient: few explicit annotations, often allow address-of-locals
- Exposed: users control lifetime/placement of objects
- Comprehensive: uniform treatment of stack and heap
- Scalable: all analysis intraprocedural

Regions

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



An old idea with some support in languages (e.g., RC) and implementations (e.g., ML Kit)

Cyclone Regions

- heap region: one, lives forever, conservatively GC'd
- stack regions: correspond to local-declaration blocks:
  `{int x; int y; s}`
- dynamic regions: lexically 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*
  - heap's handle: **`heap_region`**
  - stack region's handle: none

That's the Easy Part

The implementation is *dirt simple* because the type system statically prevents dangling pointers

```
void f() {
 int* x;
 if(1) {
  int y=0;
  x=&y;
 }
 *x;
}
```

```
int* g(region_t r) {
  return rnew(r,3);
}
void f() {
 int* x;
 { region r;
   x=g(r);
 }
 *x;
}
```


The Big Restriction

- Annotate all pointer types with a *region name* (a type variable of region kind)
- `int*`ρ can point only into the region created by the construct that introduces ρ
  - heap introduces ρ_H
  - `L:`… introduces ρ_L
  - `{region r; s}` introduces ρ_r
    `r` has type `region_t<`ρ_r`>`

So What?

Perhaps the scope of type variables suffices

```
void f() {
 int*ρL x;
 if(1) {
  L: int y=0;
     x=&y;
 }
 *x;
}
```

- type of x makes no sense
- good intuition for now
- but simple scoping will *not* suffice in general

Where We Are

- Basic region constructs
- Type system annotates pointers with type variables of region kind
- More expressive: region *polymorphism*
- More expressive: region *subtyping*
- More convenient: avoid explicit annotations
- Revenge of existential types

Region Polymorphism

Apply everything we did for type variables to region names (only it's more important!)

```
void swap(int *ρ1 x, int *ρ2 y){
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

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

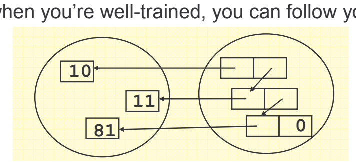

Polymorphic Recursion

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

int g = 0;

int main() {
  fact<ρH>(&g,6);
  return g;
}
```

Type Definitions

```
struct ILst<ρ1,ρ2> {
  int*ρ1 hd;
  struct ILst<ρ1,ρ2> *ρ2 tl;
};
```

- What if we said `ILst <ρ2,ρ1>` instead?
- Moral: when you're well-trained, you can follow your nose



Region Subtyping

If `p` points to an `int` in a region with name ρ_1, is it ever sound to give `p` type `int*` ρ_2?

- If so, let `int*`ρ_1 `<` `int*`ρ_2
- Region subtyping is the outlives relationship

```
void f() { region r1; … { region r2; … }}
```

- But pointers are still invariant:

  `int*`ρ_1`*`ρ `<` `int*`ρ_2`*`ρ only if $\rho_1 = \rho_2$

- Still following our nose

Subtyping cont'd

- Thanks to LIFO, a new region is outlived by all others
- The heap outlives everything

```
void f (int b, int*ρ1 p1, int*ρ2 p2) {
 L:  int*ρL p;
     if(b) p=p1; else p=p2;
     /* ...do something with p... */
}
```

- Moving beyond LIFO restricts subtyping, but the user has more options

Where We Are

- Basic region region constructs
- Type system annotates pointers with type variables of region kind
- More expressive: region *polymorphism*
- More expressive: region *subtyping*
- More convenient: avoid explicit annotations
- Revenge of existential types

Who Wants to Write All That?

- Intraprocedural inference
  - determine region annotation based on uses
  - same for polymorphic instantiation
  - based on unification (as usual)
  - so forget all those `L:` things
- Rest is by defaults
  - Parameter types get fresh region names (so default is region-polymorphic with no equalities)
  - Everything else (return values, globals, struct fields) gets ρ_H

Examples

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

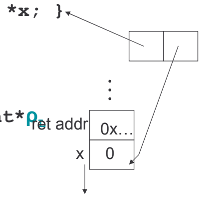
- The callee ends up writing just the equalities the caller needs to know; caller writes nothing
- Same rules for parameters to structs and typedefs
- In porting, "one region annotation per 200 lines"

But Are We Sound?

- Because types can mention only in-scope type variables, it is hard to create a dangling pointer
- But not impossible: an existential can hide type variables
- Without built-in closures/objects, eliminating existential types is a real loss
- With built-in closures/objects, you have the same problem: `(fn x -> (*y) + x) : int->int`

The Problem

```
struct T { <α>
  int (*f)(α);
  α env;
};
int read(int*ρ x) { return *x; }

struct T dangle() {
  L: int x = 0;
     struct T ans =
       T(read<ρL>,&x); //int*ρ
     return ans;
}
```



And The Dereference

```
void bad() {
  let T{<β> .f=fp, .env=ev} = dangle();
  fp(ev);
}
```

Strategy:
- Make the system "feel like" the scope-rule except when using existentials
- Make existentials usable (strengthen `struct T`)
- Allow dangling pointers, prohibit dereferencing them

Capabilities and Effects

- Attach a compile-time *capability* (a set of region names) to each program point
- Dereference requires region name in capability
- Region-creation constructs add to the capability, *existential unpacks do not*
- Each function has an effect (a set of region names)
  - body checked with effect as capability
  - call-site checks effect (after type instantiation) is a subset of capability

Not Much Has Changed Yet…

If we let the default effect be the region names in the prototype (and ρ_H), everything seems fine

```
void fact(int*ρ result, int n ;{ρ}) {
 L:  int x = 1;
     if(n > 1) fact<ρL>(&x,n-1);
     *result = x*n;
}
int g = 0;
int main(;{}) {
  fact<ρH>(&g,6);
  return g;
}
```


But What About Polymorphism?

```
struct Lst<α> {
   α hd;
   struct Lst<α>* tl;
};
struct Lst<β>* map(β f(α ;??),
                   struct Lst<α> *ρ l
                   ;??);
```

- There's no good answer
- Choosing {} prevents using `map` for lists of non-heap pointers (unless `f` doesn't dereference them)
- The Tofte/Talpin solution: effect variables

  a type variable of kind "set of region names"

Effect-Variable Approach

- Let the default effect be:
  - the region names in the prototype (and ρ_H)
  - the effect variables in the prototype
  - a fresh effect variable

```
struct Lst<β>* map(
      β f(α ; ε1),
      struct Lst<α> *ρ l
      ; ε1 + ε2 + {ρ});
```


It Works

```
struct Lst<β>* map(
      β f(α ; ε1),
      struct Lst<α> *ρ l
      ; ε1 + ε2 + {ρ});

int read(int*ρ x ;{ρ}+ε1) { return *x; }

void g(;{}) {
 L: int x=0;
     struct Lst<int*ρL>*ρH l =
               new Lst(&x,NULL);
     map< α=int*ρL β=int ρ=ρH ε1=ρL ε2={} >
        (read<ε1={} ρ=ρL>, l);
}
```


Not Always Convenient

- With *all default effects*, type-checking will never fail because of effects (!)
- Transparent until there's a function pointer in a struct:

```
struct Set<α,ε> {
  struct Lst<α> elts;
  int (*cmp)(α,α; ε)
};
```

Clients must know why ε is there

- And then there's the compiler-writer

It was time to do something new

Look Ma, No Effect Variables

- Introduce a type-level operator regions(τ)
- regions(τ) means the set of regions mentioned in t, so it's an effect
- regions(τ) reduces to a normal form:
  - regions(int) = {}
  - regions(τ*ρ) = regions(τ) + {ρ}
  - regions((τ_1,…, τ_n) $\rightarrow$ τ = regions(τ_1) + … + regions(τ_n) + regions(τ)
  - regions(α) = regions(α)

Simpler Defaults and Type-Checking

- Let the default effect be:
  - the region names in the prototype (and ρ_H)
  - regions(α) for all α in the prototype

```
struct Lst<β>* map(
      β f(α ; regions(α) + regions(β)),
      struct Lst<α> *ρ l
      ; regions(α)+ regions(β) + {ρ});
```


`map` Works

```
struct Lst<β>* map(
      β f(α ; regions(α) + regions(β)),
      struct Lst<α> *ρ l
      ; regions(α) + regions(β) + {ρ});

int read(int *ρ x ;{ρ}) { return *x; }

void g(;{}) {
 L: int x=0;
     struct Lst<int*ρL>*ρH l =
               new Lst(&x,NULL);
     map<α=int*ρL β=int ρ=ρH>
        (read<ρ=ρL>, l);
}
```

Function-Pointers Work

- With all default effects *and no existentials*, type-checking still won't fail due to effects
- And we fixed the struct problem:

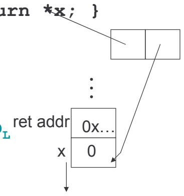
```
struct Set<α> {
  struct Lst<α> elts;
  int (*cmp)(α,α ; regions(α))
};
```


Now Where Were We?

- Existential types allowed dangling pointers, so we added effects
- The effect of polymorphic functions wasn't clear; we explored two solutions
  - effect variables (previous work)
  - regions(τ)
    - simpler
    - better interaction with structs
- Now back to existential types
  - effect variables (already enough)
  - regions(τ) (need one more addition)

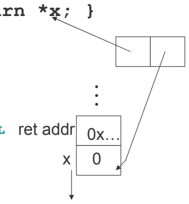
Effect-Variable Solution

```
struct T<ε>{ <α>
  int (*f)(α ;ε);
  α env;
};

int read(int*ρ x; {ρ}) { return *x; }

struct T<{ρ_L}> dangle() {
 L: int x = 0;
     struct T ans =
      T(read<ρ_L>,&x);//int*ρ_L
     return ans;
}
```



Cyclone Solution, Take 1

```
struct T { <α>
  int (*f)(α ; regions(α));
  α env;
};

int read(int*ρ x; {ρ}) { return *x; }

struct T dangle() {
L: int x = 0;
     struct T ans =
      T(read<ρ_L>,&x);//int*ρ_L
     return ans;
}
```



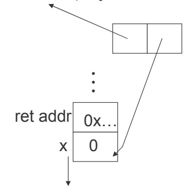
Allowed, But Useless!

```
void bad() {
  let T{<β> .f=fp, .env=ev} = dangle();
  fp(ev); // need regions(β)
}
```

- We need some way to "leak" the capability needed to call the function, preferably without an effect variable
- The addition: a *region bound*

Cyclone Solution, Take 2

```
struct T<ρ_B> { <α> α > ρ_B
  int (*f)(α ; regions(α));
  α env;
};
int read(int*ρ x; {ρ}) { return *x; }

struct T<ρ_L> dangle() {
  L: int x = 0;
     struct T<ρ_L> ans =
  T(read<ρ_L>,&x);//int*ρ_L
   return ans;
}
```



Not Always Useless

```
struct T<ρB> { <α> α > ρB
  int (*f)(α ; regions(α));
  α env;
};
```

```
struct T<ρ> no_dangle(region_t<ρ> ;{ρ});

void no_bad(region_t<ρ> r ;{ρ}) {
  let T{<β> .f=fp, .env=ev} = no_dangle(r);
  fp(ev); // have ρ and ρ ⇒ regions(β)
}
```

"Reduces effect to a single region"

Effects Summary

- Without existentials (closures,objects), simple region annotations sufficed
- With hidden types, we need effects
- With effects and polymorphism, we need abstract sets of region names
  - effect variables worked but were complicated and made function pointers in structs clumsy
  - regions(α) and region bounds were our technical contributions

We Proved It

- 40 pages of formalization and proof
- Heap organized into a stack of regions at run-time
- Quantified types can introduce region bounds of the form $\varepsilon > \rho$
- "Outlives" subtyping with subsumption rule
- Type Safety proof shows
  - no dangling-pointer dereference
  - all regions are deallocated ("no leaks")
- Difficulties
  - type substitution and regions(α)
  - proving LIFO preserved

Scaling it up (another 3 years)

Region types and effects form the core of Cyclone's type system for memory management

- Defaults are *crucial* for hiding most of it most of the time!
- But LIFO is too restrictive; need more options
  - "Dynamic regions" can be deallocated whenever
  - Statically prevent deallocation while "using"
  - Check for deallocation before "using"
  - Combine with *unique pointers* to avoid leaking the space needed to do the check
  - See SCP05/ISMM04 papers (after PLDI02 paper)

Conclusion

- Making an efficient, safe, convenient C is a lot of work
- Combine cutting-edge language theory with careful engineering and user-interaction
- Must get the common case right
- Formal models take a lot of taste to make as simple as possible and no simpler
  - They don't all have to look like ML or TAL