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 \( \tau \) (and \( \tau \) 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 intra-procedural

Regions

- a.k.a. zones, arenas, ...
- Every object is in exactly one region
- All objects in a region are deallocated simultaneously (no \texttt{free} on an object)
- Allocation via a region \texttt{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:
  \[
  \{ \texttt{int } \texttt{x}; \texttt{int } \texttt{y}; \texttt{s} \}
  \]
- dynamic regions: lexically scoped lifetime, but
  growable:
  \[
  \{ \texttt{region } \texttt{r}; \texttt{s} \}
  \]
- allocation: \texttt{rnew}(\texttt{r}, 3), where \texttt{r} is a handle
- handles are first-class
  - caller decides where, callee decides how much
  - heap's handle: \texttt{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

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

```c
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*p can point only into the region created by the construct that introduces p
  - heap introduces p_h
  - L... introduces p_L
  - {region r; s} introduces p_r
    r has type region_t<p_r>

So What?

Perhaps the scope of type variables suffices

```c
void f() {
    int p_h 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!)

```c
void swap(int *p1 x, int *p2 y){
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
int*p sumptr(region_t<p> r, int x, int y){
    return rnew(r) (x+y);
}
```

Polymorphic Recursion

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

```c
struct ILst<\rho_1, \rho_2> {
    int* p1, hd;
    struct ILst<\rho_1, \rho_2> *p2, tl;
};
```

- What if we said `ILst<\rho_2, \rho_1>` instead?
- Moral: when you're well-trained, you can follow your nose

Region Subtyping

If \( \rho \) points to an int in a region with name \( \rho_a \), is it ever sound to give \( \rho \) type `int* \rho_2`?

- If so, let `int* \rho_1 < int* \rho_2`
- Region subtyping is the outlives relationship
  ```c
  void f() { region r1; ... { region r2; ... }}
  ```
  But pointers are still invariant:
  ```c
  int* \rho_1*p < int* \rho_2*p 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
  ```c
  void f (int b, int* p1, int* p2, p2) {
    L: int* p, 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 \( \rho_a \)

Examples

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

- 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: \( \text{fn x -> (*y) + x} : \text{int->int} \)

And The Dereference

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

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

Not Much Has Changed Yet...

If we let the default effect be the region names in the prototype (and \( p \)), \text{everything seems fine}

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

The Problem

```c
struct T { <\alpha> 
    int (*f)(\alpha); 
    a env; 
};
```

```c
int read(int\*p x) { return *x; }
```

```c
struct T dangle() { 
    L: int x = 0; 
    struct T ans = 
        T(read\(<\beta>\&(x); 
    return ans; 
}
```

Capabilities and Effects

- Attach a compile-time \textit{capability} (a set of region names) to each program point
- Dereference requires region name in capability
- Region-creation constructs add to the capability, \textit{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

But What About Polymorphism?

```c
struct Lst<\alpha> { 
    a hd; 
    struct Lst<\alpha>* tl; 
};
```

```c
struct Lst<\beta>* map(\beta f(\alpha ;??), 
    struct Lst<\alpha> *p l 
    ;??); 
```

- There’s no good answer
- Choosing \( \} \) prevents using \text{map} for lists of non-heap pointers (unless \( \ell \) doesn’t dereference them)
- The Toft/Talpin solution: \textit{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 $\rho_a$)
  - the effect variables in the prototype
  - a fresh effect variable

```c
struct List<beta>* map(
    beta f(a : e),
    struct List<alpha>* p l
    ; e_1 + e_2 + {p});
```

It Works

```c
struct List<beta>* map(
    beta f(a : e),
    struct List<alpha>* p l
    ; e_1 + e_2 + {p});
int read(int*p*x :{p}+e_1) { return *x; }
void g();
L: int x=0;
    struct List<int*p>**p_1 = new List<int*, NULL>;
    map<alpha=1, beta=int p=p_1 e_1=p_1 e_2=0> (read<e_1>(); p=p_1, 1);
```

Not Always Convenient

- With all default effects, type-checking will never fail
- Transparent until there’s a function pointer in a struct:

```c
struct Set<alpha,e> { 
    struct List<alpha> elts;
    int (*cmp)(alpha, alpha; e)
}

Clients must know why e 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(t)
- regions(t) means the set of regions mentioned in t,
  so it’s an effect
- regions(t) reduces to a normal form:
  - regions(int) = {} 
  - regions(t*p) = regions(t) + {p} 
  - regions(t_1, ..., t_n) = regions(t_1) + ... + regions(t_n)
  - regions(a) = regions(a)

Simpler Defaults and Type-Checking

- Let the default effect be:
  - the region names in the prototype (and $\rho_a$)
  - regions(a) for all $\alpha$ in the prototype

```c
struct List<beta>* map(
    beta f(a : regions(a) + regions(beta)),
    struct List<alpha>* p l
    ; regions(a) + regions(beta) + {p});
```

map Works

```c
struct List<beta>* map(
    beta f(a : regions(a) + regions(beta)),
    struct List<alpha>* p l
    ; regions(a) + regions(beta) + {p});
int read(int*p*x :{p}) { return *x; }
void g();
L: int x=0;
    struct List<int*p>**p_1 = new List<int*, NULL>;
    map<alpha=1, beta=int p=p_1 e_1=p_1 e_2=0> (read<p=p_1>(); p=p_1, 1);
```
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:
  
  ```c
  struct Set<α> { 
    struct List<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(())
    • simpler
    • better interaction with structs

• Now back to existential types
  — effect variables (already enough)
  — regions(()) (need one more addition)

Effect-Variable Solution

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

```c
int read(int*p x; {p}) { return *x; }

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

Cyclone Solution, Take 1

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

```c
int read(int*p x; {p}) { return *x; }

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

Allowed, But Useless!

```c
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

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

```c
int read(int*p x; {p}) { return *x; }

struct T<ρ_p> dangle() {
  L; int x = 0;
  struct T<ρ_p> ans =
    T(read<ρ_p>, &x); /*/ int*ρ_p = T.read<ρ_p>, &x; /*/ int*ρ_p = T.read<ρ_p>, &x; /*/ int*ρ_p = T.read<ρ_p>, &x; /*/ int*ρ_p = T.read<ρ_p>, &x; /*/ int*ρ_p = T.read<ρ_p>, &x; */
  return ans;
}
```
Not Always Useless

```c
struct T<\rho> { <\alpha> \rho > p \\
  int (*\ell)(\alpha ; regions(\alpha)); \\
  \alpha env;
};

struct T<\rho> no_dangle(region_t<\rho> ;(\rho));
void no_bad(region_t<\rho> r ;(\rho)) { \\
  let T<\beta> .f=fp, .env=ev = no_dangle(r); \\
  fp(ev); // have p and p \Rightarrow regions(\beta)
}
```

"Reduces effect to a single region"

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 \( e>\rho \)
- "Outlives" subtyping with subsumption rule
- Type Safety proof shows
  - no dangling-pointer dereference
  - all regions are deallocated ("no leaks")
- Difficulties
  - type substitution and regions(\alpha)
  - proving LIFO preserved

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(\alpha) and region bounds were our technical contributions

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/ISM04 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