Cyclone, Regions, and Language-Based Safety

CS598e, Princeton University

27 February 2002

Dan Grossman

Cornell University

Some Meta-Comments

- This is a class lecture
  (*not* a conference talk or colloquium)
- Ask questions, especially when I assume you have K&R memorized
- Cyclone is really used, but this is a chance to:
  - focus on some of the advanced features
  - take advantage of a friendly audience

Where to Get Information

- www.cs.cornell.edu/projects/cyclone (with user's guide)
- www.cs.cornell.edu/home/danieljg

- *Cyclone: A Safe Dialect of C* [USENIX 02]
- *Region-Based Memory Management in Cyclone* [PLDI 02], proof in TR
- *Existential Types for Imperative Languages* [ESOP 02]

- The group: Trevor Jim (AT&T), Greg Morrisett, Mike Hicks, James Cheney, Yanling Wang
- Related work: bibliographies and rest of your course (so pardon omissions)

Cyclone in One Slide

A safe, convenient, and modern language/compiler

at the C level of abstraction

- Safe: Memory safety, abstract types, no core dumps
- C-level: User-controlled data representation, easy interoperability, resource-management control
- Convenient: “looks like C, acts like C”, but may need more type annotations
- Modern: discriminated unions, pattern-matching, exceptions, polymorphism, existential types, regions, …

“New code for legacy or inherently low-level systems”

I Can’t Show You Everything…

- Basic example and design principles
- Some pretty-easy improvements
  - Pointer types
  - Type variables
- Region-based memory management
  - A programmer’s view
  - Interaction with existentials

A Complete Program

```
#include <stdio.h>
int main(int argc, char?? argv){
  char s[] = "%s ";
  while(--argc)
    printf(s, *++argv);
  printf("\n");
  return 0;
}
```

More Than Curly Braces

```
#include <stdio.h>
int main(int argc,char??argv){
  char s[] = "%s ";
  while(--argc)
    printf(s, *++argv);
  printf("\n");
  return 0;
}
```

- diff to C: 2 characters
- pointer arithmetic
- s stack-allocated
- “\n” allocated as in C
- mandatory return

Bad news: Data representation for **`argv`** and arguments to **`printf`** is not like in C

Good news: Everything exposed to the programmer, future versions will be even more C-like

Basic Design Principles

- Type Safety (!)
- “If it looks like C, it acts like C”
  - no hidden state, easier interoperability
- Support as much C as possible
  - can’t “reject all programs”
- Add easy-to-use features to capture common idioms
  - parametric polymorphism, regions
- No interprocedural analysis
- Well-defined language at the source level
  - no automagical compiler that might fail

I Can't Show You Everything…

- Basic example and design principles
- Some pretty-easy improvements
  - Pointer types
  - Type variables
- Region-based memory management
  - A programmer's view
  - Interaction with existentials

Cyclone Pointers

- C pointers serve a few common purposes, so we distinguish them
- Basics:

`t*`	pointer to one **`t`** value or **`NULL`**
`t@`	pointer to one **`t`** value
`t?`	pointer to array of **`t`** values, plus bounds information; or **`NULL`**

Basic Pointers cont'd

Already interesting:

- Subtyping: `t@ < t* < t?`
  - one has a run-time effect, one doesn't
  - downcasting via run-time checks
- Checked pointer arithmetic on `t?`
  - don't check until subscript despite ANSI C
- `t?` are "fat", hurting C interoperability
- `t*` and `t?` may have inserted `NULL` checks
  - why not just use the hardware trap?

Example

```
FILE* fopen(const char?, const char?);
int fgetc(FILE @);
int fclose(FILE @);
void g() {
  FILE* f = fopen("foo");
  while(fgetc(f) != EOF) {…}
  fclose(f);
}
```

- Gives warnings and inserts a **`NULL`** check
- Encourages a hoisted check

The Same Old Moral

```
FILE* fopen(const char?, const char?);
int fgetc(FILE @);
int fclose(FILE @);
```

- Richer types make interface stricter
- Stricter interface make implementation easier/faster
- Exposing checks to user lets them optimize
- Can’t check everything statically (e.g., close-once)
- “never **`NULL`**” is an *invariant* an analysis may not find
- Memory safety is indispensable

More Pointer Types

- Constant-size arrays: **`t*{18}, t@{42}, t x[100]`**
- Width subtyping: **`t*{42} < t*{37}`**
- Brand new: Zero-terminators
- Coming soon: “abstract constants” (i.e. singleton ints)
- What about lifetime of the object pointed to?

I Can’t Show You Everything…

- Basic example and design principles
- Some pretty-easy improvements
  - Pointer types
  - Type variables
- Region-based memory management
  - A programmer’s view
  - Interaction with existentials

“Change `void*` to Alpha”

```
struct Lst {
  void* hd;
  struct Lst* tl;
};

struct Lst* map(
  void* f(void*);
  struct Lst*);

struct Lst* append(
  struct Lst*,
  struct Lst*);
```

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

struct Lst<`b>* map(
  `b f(`a),
  struct Lst<`a> *);

struct Lst<`a>* append(
  struct Lst<`a>*,
  struct Lst<`a>*);
```

Not Much New Here

- `struct Lst` is a type constructor:
  $Lst = \lambda\alpha.\ \{\ \alpha\ \ hd;\quad (Lst\ \alpha)\ *\ \ tl;\ \}$
- The functions are polymorphic:
  $map : \forall\alpha, \beta.\ (\alpha\rightarrow\beta,\ Lst\ \alpha) \rightarrow (Lst\ \beta)$
- Closer to C than ML
  - less type inference allows first-class polymorphism
  - data representation restricts `` `a `` to thin pointers, `int` (why not structs? why not `float`? why `int`?)
- Not C++ templates

Existential Types

- C doesn't have closures or objects, so users create their own "callback" types:

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

- We need an α (not quite the syntax):

```
struct T { ∃ α
   int (@f)(α, int);
   α env;
};
```

Existential Types cont'd

```
struct T { ∃ α
  int (@f)(α,int);
  α env;
};
```

- α is the *witness type*
- creation requires a "consistent witness"
- type is just **`struct T`**

- use requires an explicit "unpack" or "open":

```
int applyT(struct T pkg, int arg) {
  let T{<β> .f=fp, .env=ev} = pkg;
  return fp(ev,arg);
}
```

Closures and Existential Types

- Consider compiling higher-order functions:

  λ**x.e : $\alpha \to \beta$ $\Rightarrow$**

  $\exists\gamma$\{ λx.e′:(α′* γ)$\to\beta$′, env:γ \}

- That's why explicit existentials are rare in high-level languages

- In Cyclone we can write:

```
struct Fn<`a,`b> { ∃ `c
   `b (@f)(`a,`c); `c env;
};
```

  But this is not a function pointer

I Can’t Show You Everything…

- Basic example and design principles
- Some pretty-easy improvements
  - Pointer types
  - Type variables
- Region-based memory management
  - A programmer’s view
  - Interaction with existentials

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 recent 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 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 will restrict subtyping, but the user will have 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 $\rho_{\mathtt{H}}$

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”

I Can’t Show You Everything…

- Basic example and design principles
- Some pretty-easy improvements
  - Pointer types
  - Type variables
- Region-based memory management
  - A programmer’s view
  - Interaction with existentials

But Are We Sound?

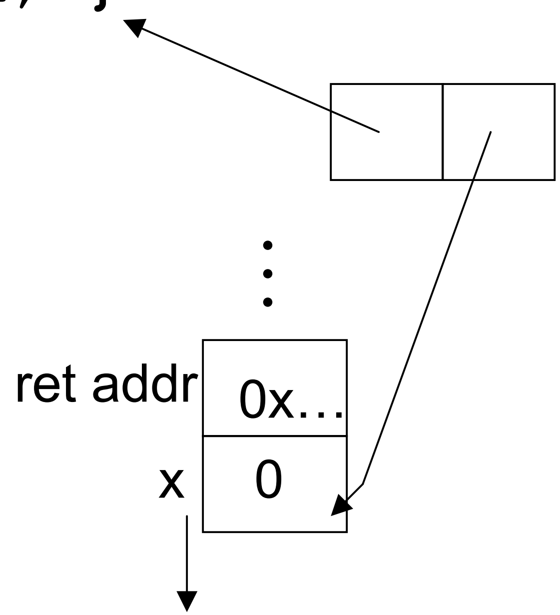
- Because types can mention only in-scope type variables, it is hard to create a dangling pointer
- But not impossible: an existential can hide type variables
- Without built-in closures/objects, eliminating existential types is a real loss
- With built-in closures/objects, you have the same problem

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 = {<int@ρ_L>
       .f = read<ρ_L>,
       .env = &x};
     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* $\boldsymbol{\rho}_H$*), 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 $\boldsymbol{\rho}_{\mathtt{H}}$)
  - 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

- Conjecture: With all default effects and no existentials, type-checking 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?

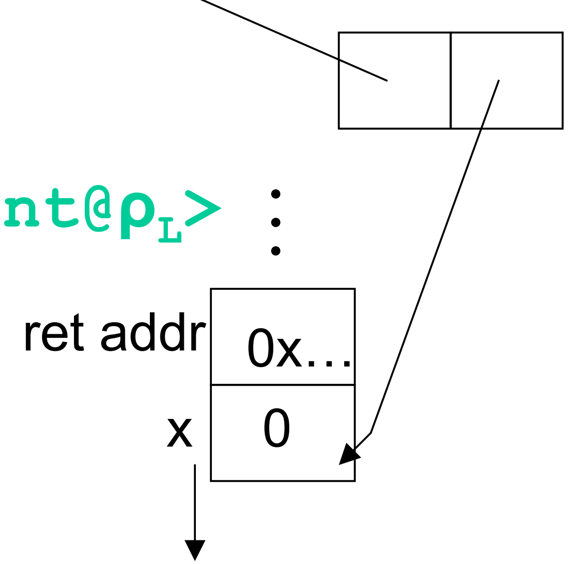
- Existential types allowed dangling pointers, so we added effects
- The effect of polymorphic functions wasn't clear; we explored two solutions
  - effect variables (previous work)
  - regions(τ)
    - simpler
    - better interaction with structs
- Now back to existential types
  - effect variables (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<{ρ_L}> ans = {<int@ρ_L> 
       .func = read<ρ_L>,
       .env = &x};
     return ans;
}
```

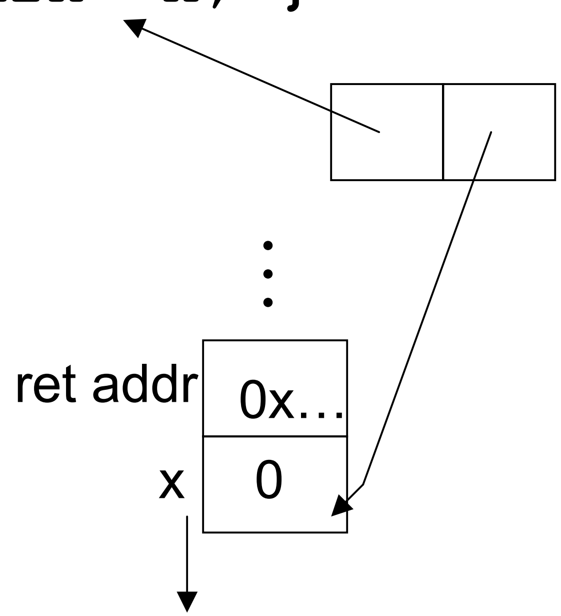


Cyclone Solution, Take 1

```
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 = {<int@ρ_L>
       .func = read<ρ_L>,
       .env = &x};
     return ans;
}
```



Allowed, But Useless!

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

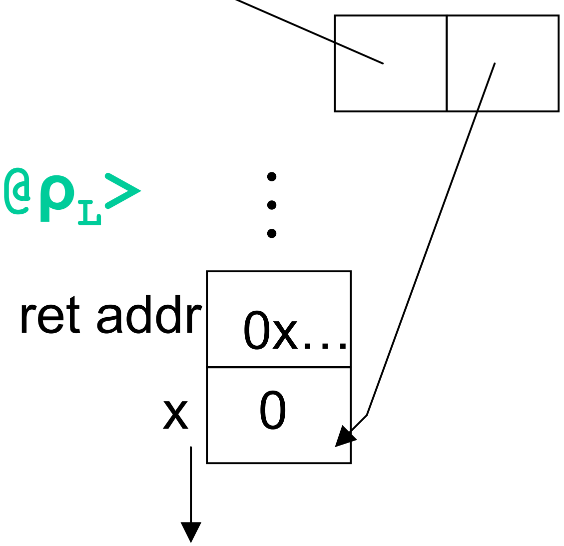
- We need some way to "leak" the capability needed to call the function, preferably without an effect variable
- The addition: a *region bound*

Cyclone Solution, Take 2

```
struct T<ρ_B> { ∃ α > 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 = {<int@ρ_L>
       .func = read<ρ_L>,
       .env = &x};
     return ans;
}
```



Not Always Useless

```
struct T<ρB> { ∃ 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

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
- Plenty of work left (e.g., error messages)

We Proved It

- 40 pages of formalization and proof
- 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

Important work, but "write only"?

Project Ideas

- Write something interesting in Cyclone
  - some secure interface
  - objects via existential types
- Change implementation to restrict memory usage
  - prevent stack overflow
  - limit heap size
- Extend formalization
  - exceptions
  - garbage collection

For implementation, get the current version!