

---

# Cyclone: Safe Programming at the C Level of Abstraction

Dan Grossman  
Cornell University  
May 2002

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

---

# A disadvantage of C

---

- Lack of *memory safety* means code cannot enforce modularity/abstractions:

```
void f() { *((int*)0xBAD) = 123; }
```

- What might address `0xBAD` hold?
- Memory safety is crucial for your favorite policy

*No desire to compile programs like this*

# Safety violations rarely local

---

```
void g(void**x, void*y) ;  
  
int  y = 0;  
int  *z = &y;  
g(&z, 0xBAD) ;  
*z = 123;
```

- Might be safe, but not if `g` does `*x=y`
- Type of `g` enough for separate code generation
- Type of `g` not enough for separate safety checking

# Some other problems

---

- One safety violation can make your favorite policy extremely difficult to enforce

- So prohibit:

incorrect casts, array-bounds violations, misused unions, uninitialized pointers, dangling pointers, null-pointer dereferences, dangling longjmp, vararg mismatch, not returning pointers, data races, ...

# What to do?

---

- Stop using C
  - YFHLL is *usually* a better choice
- Compile C more like Scheme
  - type fields, size fields, live-pointer table, ...
  - fail-safe for legacy whole programs
- Static analysis
  - very hard, less modular
- Restrict C
  - not much left

# Cyclone in brief

---

***A safe, convenient, and modern language  
at the C level of abstraction***

- **Safe:** memory safety, abstract types, no core dumps
- **C-level:** user-controlled data representation and resource management, easy interoperability, “manifest cost”
- **Convenient:** may need more type annotations, but work hard to avoid it
- **Modern:** add features to capture common idioms

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

# The plan from here

---

- Not-null pointers
- Type-variable examples
  - parametric polymorphism
  - region-based memory management
- Dataflow analysis
- Status
- Related work

*I will skip many very important features*

# Not-null pointers

---

$t^*$	pointer to a $t$ value or <b>NULL</b>
$t@$	pointer to a $t$ value

- Subtyping:  $t@ < t^*$  but  $t@@ \not< t^*@$
- Downcast via run-time check, often avoided via flow analysis

# Example

---

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

- Gives warning and inserts one 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)

# “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

---

Closer to C than ML:

- less type inference allows first-class polymorphism and polymorphic recursion
- data representation may restrict  $\alpha$  to pointers, `int` (why not structs? why not `float`? why `int`?)
- Not C++ templates

# Existential types

---

- Programs need a way for “call-back” types:

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

- We use an existential type (simplified for now):

```
struct T { <`a>  
    void (@f) (`a, int);  
    `a env;  
};
```

***more C-level than baked-in closures/objects***

# The plan from here

---

- Not-null pointers
- Type-variable examples ( $\alpha$ ,  $\forall$ ,  $\exists$ ,  $\lambda$ )
  - parametric polymorphism
  - region-based memory management
- Dataflow analysis
- Status
- Related work

*I will skip many very important features*

# 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 (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**: 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

# That's the easy part

---

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

```
void f() {
    int* x;
    if(1) {
        int y = 0;
        x = &y; // x not dangling
    }
    *x; // x dangling
}
```

# The big restriction

---

- Annotate all pointer types with a *region name* (a type variable of region kind)
- `int@`r` means “pointer into the region created by the construct that introduces ``r`”
  - heap introduces ``H`
  - `L:...` introduces ``L`
  - `region r {s}` introduces ``r`  
`r` has type `region_t<`r>`

# Region polymorphism

---

Apply what we did for type variables to region names  
(only it's more important and could be more onerous)

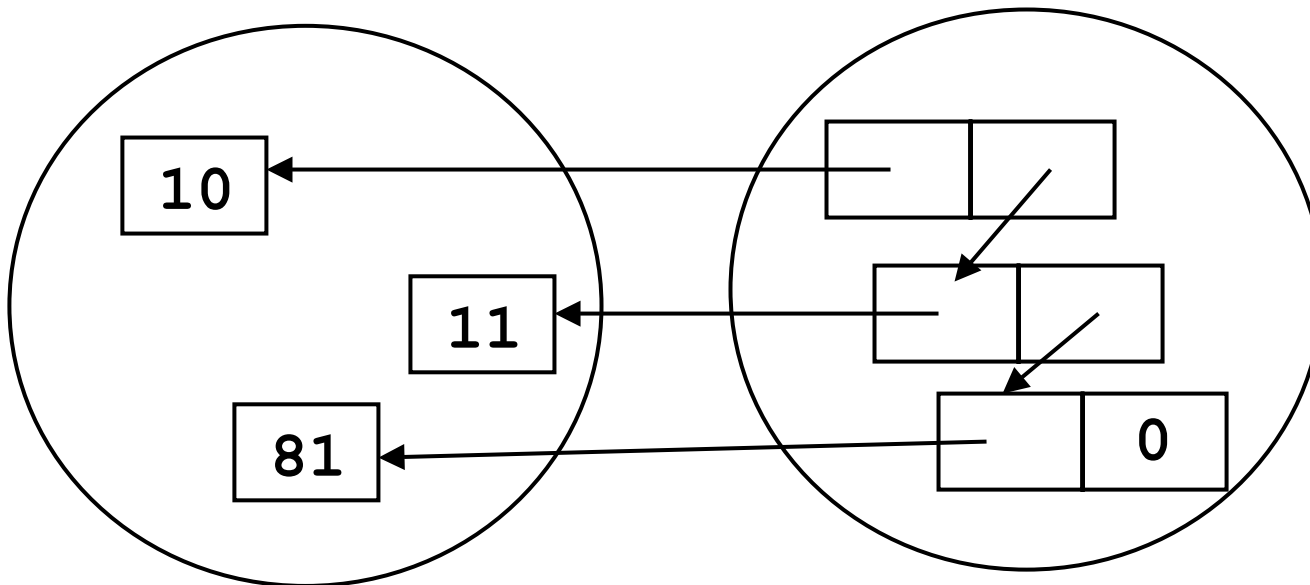
```
void swap(int @`r1 x, int @`r2 y) {  
    int tmp = *x;  
    *x = *y;  
    *y = tmp;  
}
```

```
int@`r sumptr(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 t1;  
};
```



# 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

# Soundness

---

- Ignoring  $\exists$ , scoping prevents dangling pointers

```
int* `L f() { L: int x; return &x; }
```

- End of story if you don't use  $\exists$
- For  $\exists$ , we leak a *region bound*:

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

- A powerful effect system is there in case you want it

# Regions summary

---

- Annotating pointers with region names (type variables) makes a sound, simple, static system
- Polymorphism, type constructors, and subtyping recover much expressiveness
- Inference and defaults reduce burden
- Other ~~chapters~~ future features:
  - array bounds: `void f(tag_t<`i>,int*`i);`  
default 1
  - mutual exclusion: `void f(lock_t<`L>,int*`L);`  
default thread-local

# The plan from here

---

- Not-null pointers
- Type-variable examples
  - parametric polymorphism
  - region-based memory management
- **Dataflow analysis**
- Status
- Related work

*I will skip many very important features*

# Example

---

```
int*`r* f(int*`r q) {  
    int **p = malloc(sizeof(int*));  
    // p not NULL, points to malloc site  
    *p = q;  
    // malloc site now initialized  
    return p;  
}
```

- Harder than in Java because of pointers
- Analysis includes must-points-to information
- Interprocedural annotation: “initializes” a parameter

# Flow-analysis strategy

---

- Current uses: definite-assignment, null-checks, array-bounds checks, must return
- When invariants are too strong, program-point information is more useful
- Sound with respect to aliases (programmers can make copies)
- Checked interprocedural annotations keep analysis local

# Status

---

- Cyclone really exists
  - 110KLOC, including bootstrapped compiler, web server, multimedia overlay network, ...
  - gcc back-end (Linux, Cygwin, OSX, ...)
  - user's manual, mailing lists, ...
  - still a research vehicle
  - more features: exceptions, tagged unions, varargs, ...
- Publications
  - overview: USENIX 2002
  - regions: PLDI 2002
  - existentials: ESOP 2002

# Related work: higher and lower

---

- Adapted/extended ideas:
  - polymorphism [ML, Haskell, ...]
  - regions [Tofte/Talpin, Walker et al., ...]
  - safety via dataflow [Java, ...]
  - existential types [Mitchell/Plotkin, ...]
  - controlling data representation [Ada, Modula-3, ...]
- Safe lower-level languages [TAL, PCC, ...]
  - engineered for machine-generated code
- Vault: stronger properties via restricted aliasing

# Related work: making C safer

---

- Compile to make dynamic checks possible
  - Safe-C [Austin et al., ...]
  - Purify, Stackguard, Electric Fence, ...
  - CCured [Necula et al.]
    - performance via whole-program analysis
    - more on array-bounds, less on memory management and polymorphism
- RC [Gay/Aiken]: reference-counted regions separate from stack and heap
- LCLint [Evans]: unsound-by-design, but very useful
- SLAM: checks user-defined property w/o annotations; assumes no bounds errors

# Plenty left to do

---

- Beyond LIFO memory management
- Resource exhaustion (e.g., stack overflow)
- More annotations for aliasing properties
- More “compile-time arithmetic” (e.g., array initialization)
- Better error messages (not a beginner’s language)

# Summary

---

- Memory safety is essential for your favorite policy
- C isn't safe, but the world's software-systems infrastructure relies on it
- Cyclone combines advanced types, flow analysis, and run-time checks, to create a safe, usable language with C-like data, resource management, and control

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

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

*best to write some code*