Cyclone: Safe C-Level Programming
(With Multithreading Extensions)

Dan Grossman
Cornell University
October 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

```c
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
  - multithreading
- Dataflow analysis
- Status
- Related work

*I will skip many very important features*
Not-null pointers

| \( \ast \) | pointer to a \( t \) value or \texttt{NULL} |
| \( @ \)     | pointer to a \( t \) value |

- Subtyping: \( @ < \ast \) but \( @@ \not< \ast@ \)
- Downcast via run-time check, often avoided via flow analysis
Example

```c
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

```c
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”

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

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

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

```c
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, \texttt{int} (why not structs? why not \texttt{float}? why \texttt{int}?)

• Not C++ templates
Existential types

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

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

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

```c
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
  – parametric polymorphism (α, ∀, ∃, λ)
  – region-based memory management
  – multithreading
• 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

```c
void f() {
    int* x;
    if(1) {
        int y = 0;
        x = &y; // x not dangling
    }
    *x = 123; // 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)

```c
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 tl;
};
```
Region subtyping

If \( p \) points to an \texttt{int} in a region with name \texttt{`r1}, is it ever sound to give \( p \) type \texttt{int*`r2}?

- If so, let \texttt{int*`r1 < int*`r2}

- Region subtyping is the \texttt{outlives} relationship

  \[
  \text{region r1 \{... region r2 \{...\}...}
  \]

- LIFO makes subtyping common
Soundness

• Ignoring $\exists$, scoping prevents dangling pointers
  \[ \text{int*`L f() { L: int x; return &x; }} \]

• End of story if you don’t use $\exists$

• For $\exists$, we leak a region bound:
  \[ \text{struct T<`r> { `<a> :regions(`a) > `r void (@f)(`a, int); `a env; }} \]

• 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

• With additional run-time checks, can move beyond LIFO, but checks can fail
• Key point: do not check on every access
The plan from here

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

*I will skip many very important features*
Data races break safety

- Data race: One thread accessing memory while another thread writes it
- On shared-memory MPs, a data race can corrupt a pointer
- Atomic word writes insufficient
  - struct with array bound and pointer to array
  - more generally, existential types

Cyclone must prevent data races
Preventing data races

• Static
  – Don’t have threads
  – Don’t have thread-shared memory
  – Require mutexes for all memory
  – Require mutexes for shared memory
  – Require sound synchronization for shared memory
  – ...

• Dynamic
  – Detect races as they occur
  – Control scheduling and preemption
  – ...

9 Oct 2002 Dan Grossman, Cyclone
Mutual exclusion support

Require mutual exclusion for shared memory:
- For each shared object, there exists a lock that must be acquired before access
- Thread-local data must not escape its thread

New terms:
- \texttt{spawn(f,p,sz)} run \( f(p2) \) in a thread where \( *p2 \) is a shallow copy of \( *p1 \) and \( sz \) is \( \text{sizeof}(*p1) \)
- \texttt{newlock()} create a new lock
- \texttt{nonlock} a pseudolock for thread-local data
- \texttt{sync e s} acquire lock \( e \), run \( s \), release lock

\textit{Only sync requires language support}
Example (w/o types)

```c
void inc(int@ p){*p = *p + 1;}
void inc2(lock_t m, int@ p){sync m inc(p);}
struct LkInt {lock_t m; int@ p;};
void g(struct LkInt@ s){inc2(s->m, s->p);}

void f(){
    lock_t lk = newlock();
    int@ p1 = new 0;
    int@ p2 = new 0;
    struct LkInt@ s = new LkInt{.m=lk, .p=p1};
    spawn(g, s, sizeof(*s));
    inc2(lk, p1);
    inc2(nonlock, p2);
}

Once again, this is the easy part
```
Haven’t we been here before

- Annotate all pointers and locks with a lock name (e.g., `lock_t<\text{`L}>`, `int@`L`)

- Special lock name `loc` for thread-local (`nonlock` has type `lock_t<loc>`)  

- `newlock` has type `\exists \`L. lock_t<\`L>`

- `sync e s` where `e` has type `lock_t<\`L>` allows `*p` in `s` where `p` has type `int@`L`

- default is caller locks (perfect for thread-local):

  ```c
  void inc(int@`L p;{`L}) {*p=*p+1;}
  ```
More about access rights

- For each program point, there is a set of lock names describing “held locks”
  - loc is always in the set
  - functions have set annotations, but default is caller-locks
  - sync adds appropriate name to the set
- Lexical scope for sync keeps rules simple, but is not essential
Analogy with regions

- region_t<`r>
- int*`r
- `H
- region r s
- lock_t<`L>
- int*`L
- loc
- {let m<`L>=newlock();
  sync m s}

- Access rights: region live or lock held
- Static rights amplified in lexical scope: region, sync
- Can ignore for prototyping or common case: `H, loc
Differences as well

• ... 
• region r s  
• {let m<`L>=newlock(); 
  sync m s}

• A region’s objects are accessible from region creation to region deletion (which happens once)
• A lock’s objects are accessible within a sync (which happens many times)
• So region combines newlock and sync
• So locks don’t induce subtyping

Language/type-system design reflects reality
Safe multithreading, so far

• Terms `newlock`, `nonlock`, `sync`, `spawn`
• Types `lock_t<`L>, t*`L, lock_t<loc>, t*loc`
• Type system assigns access rights to each program point
• Strikingly similar to memory management
• But have we prevented data races?

*If we never pass thread-local data to spawn!*
Enforcing loc

• A possible type for spawn:
  ```c
  void spawn(void f(`a@loc ;{}), `a@`L,
              sizeof_t<`a> ;{`L});
  ```

• But not any `a will do

• We already have different kinds of type variables: R for regions, L for locks, B for pointer types, A for all types

• Examples: loc::L, `H::R, int*`H::B,
  ```c
  struct T :: :: A
  ```
Enforcing loc cont’d

- Enrich kinds with *sharabilities*, $S$ or $U$
- $loc::LU$
- `newlock()` has type $\exists`L::LS. \text{lock}_t<`L>$
- A type is sharable only if every part is sharable
- Every type is unsharable
- Unsharable is the default

```c
void spawn<`a::AS>(void(@f)(`a@;{}),
    `a@`L,
    sizeof_t<`a>
    ;{`L});
```
Threads summary

• A type system where:
  – thread-shared data must have locks
  – thread-local data must not escape
  – locks are first-class and code is reusable

• Like regions except locks are reacquirable and thread-local is harder than lives-forever

• Did not discuss: thread-shared regions (must not deallocate until all threads are done with it)
Threads shortcomings

- Global variables need top-level locks
  - otherwise, single-threaded code works unchanged
- Shared data enjoys an initialization phase
- Object migration
- Read-only data and reader/writer locks
- Semaphores, signals, ...
- Deadlock (not a safety problem)
The plan from here

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

*I will skip many very important features*
Example

```c
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

- Checked interprocedural annotations keep analysis local

- Two hard technical issues:
  - sound and explainable with respect to aliases
  - under-specified evaluation order
Status

• Cyclone really exists (except for threads)
  – 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 (threads work submitted)
  – 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., …]
  - lock types [Flanagan et al., Boyapati 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 array-bounds, less memory management
    - inherently single-threaded
- RC [Gay/Aiken]: reference-counted regions, unsafe 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.cs.cornell.edu/projects/cyclone

*best to write some code*