Cyclone: A Memory-Safe C-Level Programming Language

Dan Grossman

University of Washington

Joint work with:

Trevor Jim	AT&T Research
Greg Morrisett	Harvard University
Michael Hicks	University of Maryland

A safe C-level language

Cyclone is a programming language and compiler aimed at safe systems programming

- C is not *memory safe*:

```
void f(int* p, int i, int v) {
    p[i] = v;
}
```

- Address **`p+i`** might hold important data or code
- Memory safety is crucial for reasoning about programs

Caller's problem?

```
void g(void**, void*);

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 code generation
- Type of **g** *not* enough for safety checking

Safe low-level systems

- For a safety guarantee today, use YFHLL

  *Y*our *F*avorite *H*igh *L*evel *L*anguage

- YFHLL provides safety in part via:
  - hidden data fields and run-time checks
  - automatic memory management
- Data representation and resource management are essential aspects of low-level systems

There are strong reasons for C-like languages

Some insufficient approaches

- Compile C with extra information
  - type fields, size fields, live-pointer table, …
  - treats C as a higher-level language
- Use static analysis
  - very difficult
  - less modular
- Ban unsafe features
  - there are many
  - you need them

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

- Experience with Cyclone
  - Benchmarks, ports, systems, compiler, …
  - All on Earth so far ☺
- Not-NULL pointers
- Type-variable examples
  - generics
  - region-based memory management
- Brief view of “everything else”
- Related work

Really “just a taste” of Cyclone

Status

Cyclone really exists (except memory-safe threads)

- >150K lines of Cyclone code, including the compiler
- gcc back-end (Linux, Cygwin, OSX, Mindstorm, …)
- User's manual, mailing lists, …
- Still a research vehicle

Evaluation

1. Is Cyclone like C?
   – port code, measure source differences
   – interface with C code (extend systems)

2. What is the performance cost?
   – port code, measure slowdown

3. Is Cyclone good for low-level systems?
   – write systems, ensure scalability

Code differences

Example	Lines of C	diff total	incidental	bugs found
grobner (1 of 4)	3260	+ 257 (7.9%) – 190	41 (216=6.6%)	1 (half of examples)
mini-httpd (1 of 6)	3005	+ 273 (9.1%) – 245	12 (261=8.7%)	1
ccured-olden-mst (1 of 4)	584	+ 34 (5.8%) – 29	2 (32=5.5%)	0

- Porting not automatic, but quite similar
- Many changes identify arrays and lengths
- Some changes incidental (absent prototypes, new keywords)

Run-time performance

Example	Lines of C	diff total	execution time	faster	execution time
grobner (1 of 4)	3260	+ 257 – 190	1.94x	+ 336 – 196	1.51x
mini-httpd (1 of 6)	3005	+ 273 – 245	1.02x		
ccured-olden-mst (1 of 4)	584	+ 34 – 29	1.93x	+ 35 – 30 nogc	1.39x

RHLinux 7.1 (2.4.9), 1.0GHz PIII, 512MRAM, gcc2.96 -O3, glibc 2.2.4

- Comparable to other safe languages to start
- C level provides important optimization opportunities
- Understanding the applications could help

Larger program: the compiler

- Scalable
  - compiler + libraries (80K lines) build in < 30secs
- Generic libraries (e.g., lists, hashtables)
  - clients have no syntactic/performance cost
- Static safety helps exploit the C-level
  - I use `&x` more than in C

Other projects

- Open Kernel Environment [Bos/Samwel, OPENARCH 02]
- MediaNet [Hicks et al, OPENARCH 03]:
- RBClick [Patel/Lepreau, OPENARCH 03]
- STP [Patel et al., SOSP 03]
- FPGA synthesis [Teifel/Manohar, ISACS 04]
- Maryland undergrad O/S course (geekOS) [2004]
- Windows device driver (6K lines)
  - Only 100 lines left in C
  - But unrecoverable failures & other kernel corruptions remain

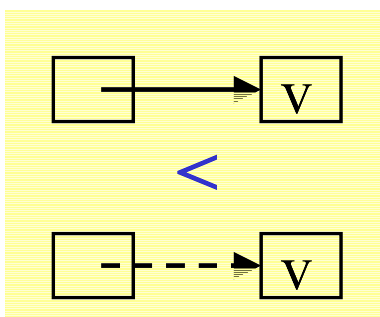
The plan from here

- Experience with Cyclone
- Not-NULL pointers
- Type-variable examples
  - generics
  - region-based memory management
- Brief view of “everything else”
- Related work

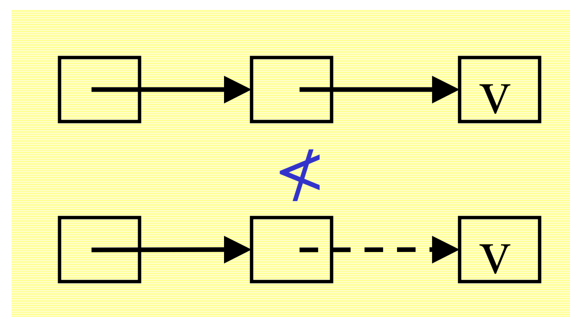
Not-null pointers

t*	pointer to a **t** value or **NULL**
t@	pointer to a **t** value

- Subtyping: t@ < t* but t@@ ≮ t*@



but



- 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");
  int c;
  while((c = fgetc(f)) != EOF) {…}
  fclose(f);
}
```

- Gives warning and inserts one null-check
- Encourages a hoisted check

A classic moral

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

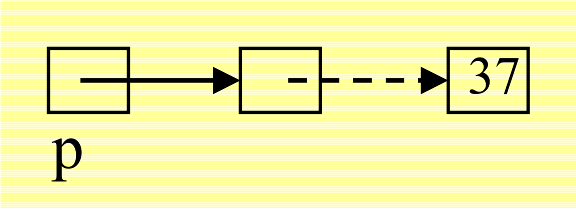
- Richer types make interface stricter
- Stricter interface make implementation easier/faster
- Exposing checks to user lets them optimize
- Can’t check everything statically (e.g., close-once)

Key Design Principles in Action

- Types to express invariants
  - Preconditions for arguments
  - Properties of values in memory
- Flow analysis where helpful
  - Lets users control explicit checks
  - *Soundness* + *aliasing limits usefulness*
- Users control data representation
  - Pointers are addresses unless user allows otherwise
- Often can interoperate with C more safely just via types

It’s always aliasing

```
void f(int*@p) {
 if(*p != NULL) {
  g();
  **p = 42;//inserted check even w/o g()
 }
}
```



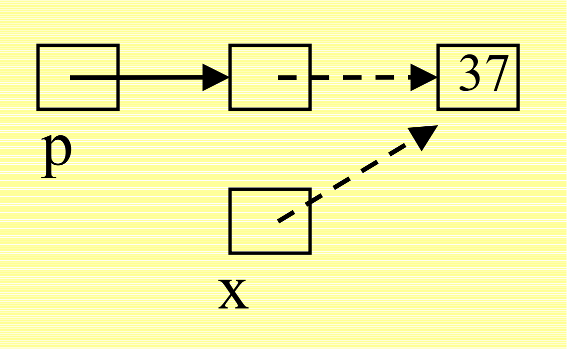
But can avoid checks when compiler knows all aliases.

Can know by:

- Types: precondition checked at call site
- Flow: new objects start unaliased
- Else user should use a temporary (the safe thing)

It’s always aliasing

```
void f(int**p) {
  int* x = *p;
  if(x != NULL) {
    g();
    *x = 42;//no check
  }
}
```



But can avoid checks when compiler knows all aliases.

Can know by:

- Types: precondition checked at call site
- Flow: new objects start unaliased
- Else user should use a temporary (the safe thing)

The plan from here

- Experience with Cyclone
- Not-NULL pointers
- Type-variable examples
  - generics
  - region-based memory management
- Brief view of "everything else"
- Related work

“Change `void*` to `` `a ``”

```
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 C++, Java generics, ML, etc.

- Unlike functional languages, data representation may restrict `a to pointers, **`int`**
  - why not structs? why not **`float`**? why **`int`**?
- Unlike templates, no code duplication or leaking implementations
- Unlike objects, no need to tag data

Existential types

- Programs need a way for "call-back" types:

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

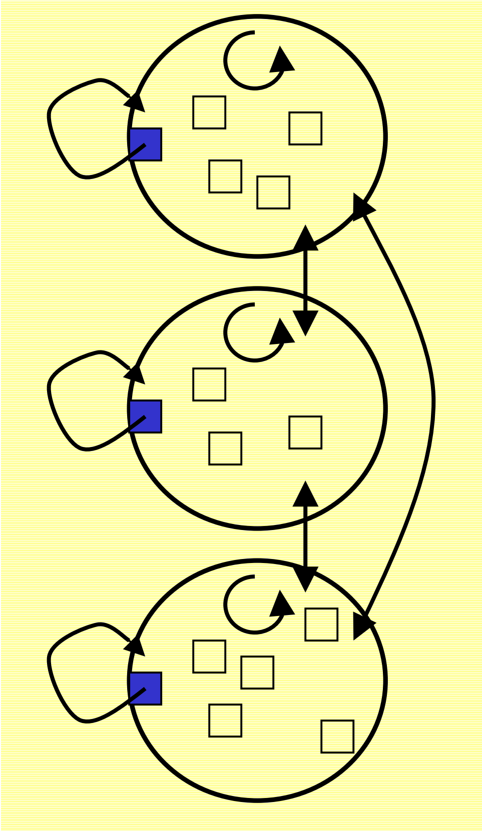
- We use an existential type (simplified):

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

more C-level than baked-in closures/objects

Regions

- a.k.a. zones, arenas, …
- Every object is in exactly one region
- Allocation via a region *handle*
- Deallocate an entire region simultaneously
  (cannot **`free`** an object)



Old idea with recent support in languages (e.g., RC, RTSJ) and implementations (e.g., ML Kit)

Cyclone regions [PLDI 02]

- heap region: one, lives forever, conservatively GC'd
- stack regions: correspond to local-declaration blocks:

```
{int x; int y; s}
```

- growable regions: scoped lifetime, but growable:

```
{region r; s}
```

- allocation routines take a region *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;
  {
    int  y = 0;
    x = &y; // x not dangling
  } // x dangling
  {
    int* z = NULL;
    *x = 123;
    ...
  }
}
```

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>``**
- compile-time check: only live regions are accessed
  - by default, function arguments point to live regions

Region polymorphism

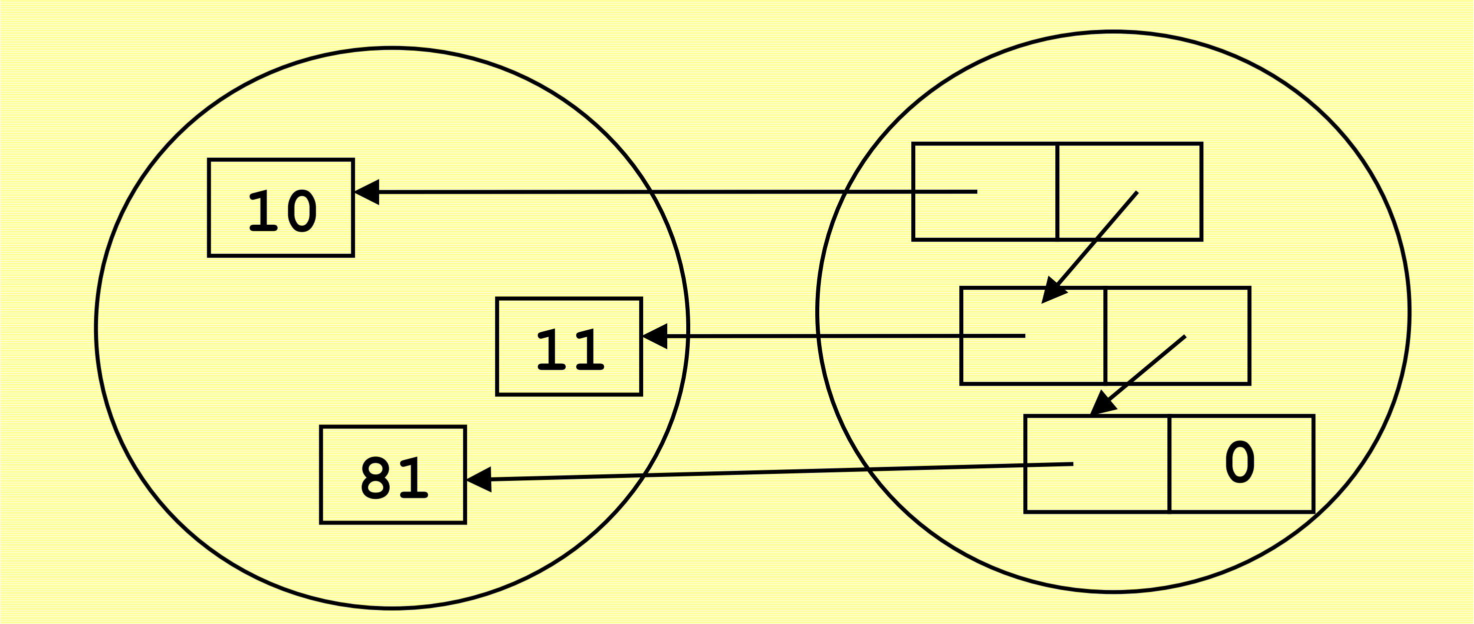
Apply what we did for type variables to region names (only it's more important and could be more onerous)

```
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 tl;
};
```



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

Regions evaluation

- LIFO regions good for some idioms awkward in C
- Regions generalize stack variables and the heap
- Defaults and inference make it surprisingly palatable
  - Worst part: defining region-allocated data structures
- Cyclone actually has much more [ISMM 04]
  - Non-LIFO regions
  - “Unique pointers”
  - Explicitly reference-counted pointers
  - A “unified system”, not *n* sublangages

The plan from here

- Experience with Cyclone
- Not-NULL pointers
- Type-variable examples
  - generics
  - region-based memory management
- Brief view of "everything else"
- Related work

Other safety holes

- Arrays (what or where is the size)
  - Options: dynamic bound, in a field/variable, compile-time bound, special string support
- Threads (avoiding races)
  - vaporware type system to enforce lock-based mutual exclusion
- Casts
  - Allow only “up casts” and casts to numbers
- Unions
  - Checked tags or bits-only fields
- Uninitialized data
  - Flow analysis (safer and easier than default initializers)
- Varargs (safe via changed calling convention)

And modern conveniences

30 years after C, some things are worth adding…

- Tagged unions and pattern matching on them
- Intraprocedural type inference
- Tuples (like anonymous structs)
- Exceptions
- Struct and array initializers
- Namespaces
- **`new`** for allocation + initialization

Plenty of work remains

Common limitations:

- Aliasing
- Arithmetic
- Unportable assumptions

(But interoperating with C is *much* simpler than in a HLL)

Big challenge for next generation:

guarantees beyond fail-safe (i.e., graceful abort)

Related work: making C safer

- Compile to make dynamic checks possible
  - Safe-C [Austin et al.], RTC [Yong/Horwitz], ...
  - Purify, Stackguard, Electric Fence, …
  - CCured [Necula et al.]
    - performance via whole-program analysis
    - less user burden
    - less memory management, single-threaded
- Control-C [Adve et al.] weaker guaranty, less burden
- SFI [Wahbe, Small, ...]: sandboxing via binary rewriting

Related Work: Checking C code

- Model-checking C code (SLAM, BLAST, …)
  - Leverages scalability of MC
  - Key is automatic building and refining of model
  - *Assumes* (weak) memory safety
- Lint-like tools (Splint, Metal, PreFIX, …)
  - Good at reducing false positives
  - *Cannot* ensure absence of bugs
  - Metal particularly good for user-defined checks
- Cqual (user-defined qualifiers, lots of inference)

Better for unchangeable code or user-defined checks (i.e., they're complementary)

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

Summary

- Cyclone: a safe language at the C-level of abstraction
- Synergistic combination of types, flow analysis, and run-time checks
- A real compiler and prototype applications
- Properties like "not NULL", "has longer lifetime", "has array length"… now in the language and checked
- Easy interoperability with C allow smooth and incremental move toward memory safety
  - in theory at least

Availability

Like any language, you have to “kick the tires”:

www.research.att.com/projects/cyclone

Also see:

- Jan. 2005 C/C++ User’s Journal
- USENIX 2002

Conversely, I want to know NASA’s C-level code needs

- Maybe ideas from Cyclone will help
- Maybe not

Either way would be fascinating