Cyclone: Safe Programming at the C Level of Abstraction

Dan Grossman

Cornell University

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

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

A question of trust

- We rely on our C-level software infrastructure to
  - not crash (or crash gracefully)
  - preserve data, restrict access, ...
  - serve customers, protect valuables, ...
- Infrastructure is enormous
  - careful humans not enough
- One safety violation breaks all isolation

Memory safety is necessary for trustworthy systems

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
- Write or extend your O/S with YFHLL?

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: a combined approach

Designed and implemented Cyclone, a safe C-level language

- Advanced type system for safety-critical invariants
- Flow analysis for tracking state changes
- Exposed run-time checks where appropriate
- Modern language features for common idioms

Today: focus on type system

Cyclone reality

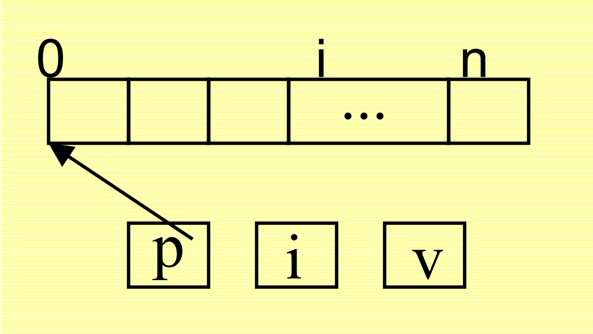
- 130K lines of code, bootstrapped compiler, Linux / Cygwin / OS X, ...
- All programs are safe (modulo interfacing to C)
- Users control if/where extra fields and checks occur
  - checks can be needed (e.g., pointer arithmetic)
- More annotations than C, but work hard to avoid it
- Sometimes slower than C
  - 1x to 2x slowdown
  - can performance-tune more than in HLLs

The plan from here

- Goals for the type system
- Safe multithreading
- Region-based memory management
- Evaluation (single-threaded)
- Related work
- Future directions

Must be safe

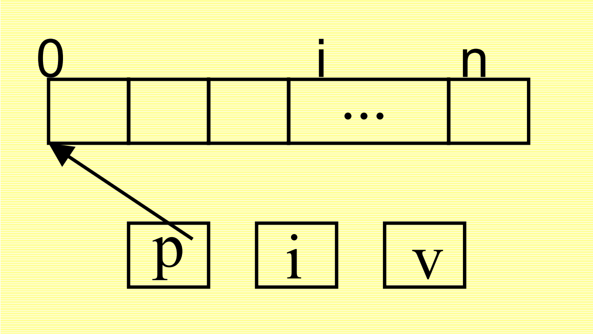
```
void f(int* p, int i, int v) {
   p[i] = v;
}
```



- All callers must ensure:
  - **p** is not NULL
  - **p** refers to an array of at least n ints
  - **0 <= i** < n
  - **p** does not refer to deallocated storage
  - no other thread corrupts **p** or **i**

But not too restrictive

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



- Different callers can have:
  - `p` refer to arrays of different lengths n
  - `i` be different integers such that `0 <= i` < n
  - `p` refer to memory with different lifetimes
  - `p` refer to thread-local or thread-shared data

Design goals

1. Safe
   – can express necessary preconditions
2. Powerful
   – parameterized preconditions allow code reuse
3. Scalable
   – explicit types allow separate compilation
4. Usable
   – simplicity vs. expressiveness
   – most convenient for common cases
   – common framework for locks, lifetimes, array bounds, and abstract types

The plan from here

- Goals for the type system
- Safe multithreading
- Region-based memory management
- Evaluation (single-threaded)
- Related work
- Future directions

Safe multithreading: the problem

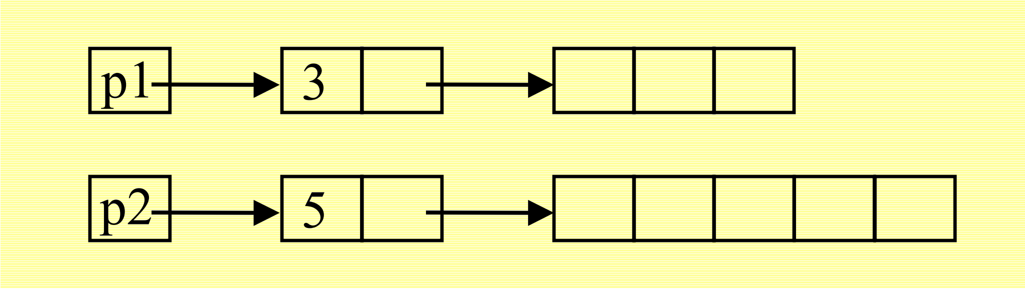
Data race: one thread mutating some memory while another thread accesses it (w/o synchronization)

1. Pointer update must be atomic
   – possible on many multiprocessors if you're careful

2. But writing addresses atomically is insufficient...

Data-race example

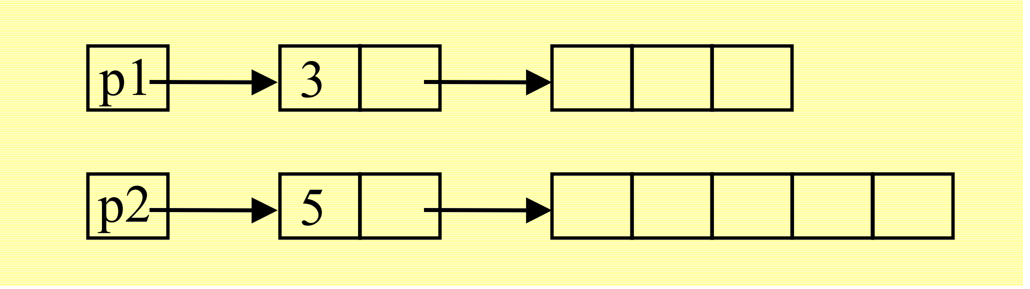
```
struct SafeArr {
  int  len;
  int* arr;
};
```




```
if(p1->len > 4)          ||  *p1 = *p2;
  (p1->arr)[4] = 42;
```

Data-race example

```
struct SafeArr {
  int  len;
  int* arr;
};
```




```
if(p1->len > 4)
  (p1->arr)[4] = 42;
```

||

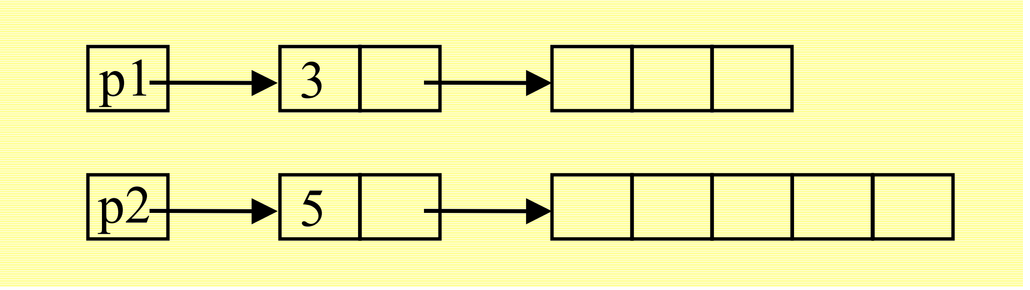
```
*p1 = *p2;
```

change ***p1->len*** *to 5*

change ***p1->arr***

Data-race example

```
struct SafeArr {
  int  len;
  int* arr;
};
```




```
if(p1->len > 4)
  (p1->arr)[4] = 42;
```

||

```
*p1 = *p2;
```

change ***p1->len*** *to 5*

check ***p1->len > 4***

write ***p1->arr[4]*** *XXX*

change ***p1->arr***

Preventing data races

Reject at compile-time code that may have data races?

- Limited power: problem is undecidable
- Trivial if too limited: e.g., don't allow threads
- A structured solution:
  Require mutual exclusion on all thread-shared data

Lock types

Type system ensures:

For each shared data object, there exists a lock that a thread must hold to access the object

- Basic approach for Java found many bugs
  [Flanagan et al]
- Extensions allow other locking idioms and code reuse for shared/local data
  [Boyapati et al]

Lock-type contributions [TLDI 03]

1. Adapt the approach to a C-level language
2. Integrate parametric polymorphism
3. Integrate region-based memory management
4. Code reuse for thread-local and thread-shared data
   – simple rule to "keep local data local"
5. Proof for an abstract machine where data races violate safety

Cyclone multithreading

- Multithreading language
  - terms
  - types
- Limitations
- Insight into why it’s safe

Multithreading terms

- **`spawn(`*«f»*`,`*«p»*`,`*«sz»*`)`**

  run **`f(p2)`** in a new thread (where **`*p2`** is a shallow copy of **`*p`** and **`sz`** is the size of **`*p`**)
  - thread initially holds no locks
  - thread terminates when **`f`** returns
  - creates shared data, but **`*p2`** is thread-local

- **`sync(`*«lk»*`){`*«s»*`}`** acquire **`lk`**, run **`s`**, release **`lk`**

- **`newlock()`** create a new lock

- **`nonlock`** a pseudo-lock for thread-local data

Examples, without types

Suppose **`*p1`** is shared (lock **`lk`**) and **`*p2`** is local

Caller-locks

```
void f(int* p) {
  « use *p »
}

void caller() {
  «...»
  sync(lk){f(p1);}
  f(p2);
}
```

Callee-locks

```
void g(int* p,
       lock_t l) {
  sync(l){« use *p »}
}
void caller() {
  «...»
  g(p1,lk);
  g(p2,nonlock);
}
```

Types

- *Lock names* in pointer types and lock types
- **``int*`L``** is a type for pointers to locations guarded by a lock with type **``lock_t<`L>``**
- Different locks cannot have the same name
  - **``lock_t<`L1>``** vs. **``lock_t<`L2>``**
  - this invariant will ensure mutual exclusion
- Thread-local locations use lock name **`` `loc ``**

lock names describe "what locks what"

Types for locks

- `nonlock` has type `` lock_t<`loc> ``
- `newlock()` has type `` ∃`L. lock_t<`L> ``
- Removing ∃ requires a fresh lock name
  - so different locks have different types
  - using ∃ is an established PL technique [ESOP 02]

Access rights

Assign each program point a set of lock names:

- if ***lk*** has type **`lock_t<`L>`**,
  `sync``(`***«lk»***`){`***«s»***`}` adds **`` `L ``**
- using location guarded by **`` `L ``** requires **`` `L ``** in set
- functions have explicit preconditions
  - default: caller locks

lock-name sets ensure code acquires the right locks

(Lock names and lock-name sets do not exist at run-time)

Examples, with types

Suppose **`*p1`** is shared (lock **`lk`**) and **`*p2`** is local

Caller-locks

```
void f(int*`L p
          ;{`L}) {
   « use *p »
}

void caller() {
  «...»
  sync(lk){f(p1);}
  f(p2);
}
```

Callee-locks

```
void g(int*`L p,
          lock_t<`L> l
          ;{}) {
   sync(l){« use *p »}
}

void caller() {
  «...»
  g(p1,lk);
  g(p2,nonlock);
}
```

Quantified lock types

- Functions universally quantify over lock names
- Existential types for data structures

```
struct LkInt {<`L> //there exists a lock-name
    int*`L       p;
    lock_t<`L>   lk;
};
```

- Type constructors for coarser locking

```
struct List<`L> { //lock-name parameter
   int*`L                head;
   struct List<`L>*`L tail;
};
```

Lock types so far

1. Safe
   – lock names describe what locks what
   – lock-name sets prevent unsynchronized access
2. Powerful
   – universal quantification for code reuse
   – existential quantification and type constructors for data with different locking granularities
3. Scalable
   – type-checking intraprocedural
4. Usable
   – default caller-locks idiom
   – bias toward thread-local data

But...

- What about spawn?

  `spawn(`«*f*»**`,`**«*p*»**`,`**«*sz*»**`)`**

  run **`f(p2)`** *in a new thread (***`*p2`*** a shallow copy of* **`*p`***)*

- Everything reachable from **`*p`** is shared

- Safe:
  - **`f`**'s argument type and **`p`**'s type should be the same
  - Type of **`*p`** must forbid (supposedly) local data

- Powerful: No other limits on the type of **`*p`**

Shareability

`spawn(«f»,«p»,«sz»)`

- Assign every type and lock name a shareability
  - `` `loc `` is unshareable
  - locks from newlock() have shareable names
  - type is shareable only if all its lock names are shareable
  - default: unshareable

    (necessary for local/shared code reuse)
- Type of `*p` must be shareable
- Result: thread-local data is really local

Cyclone multithreading

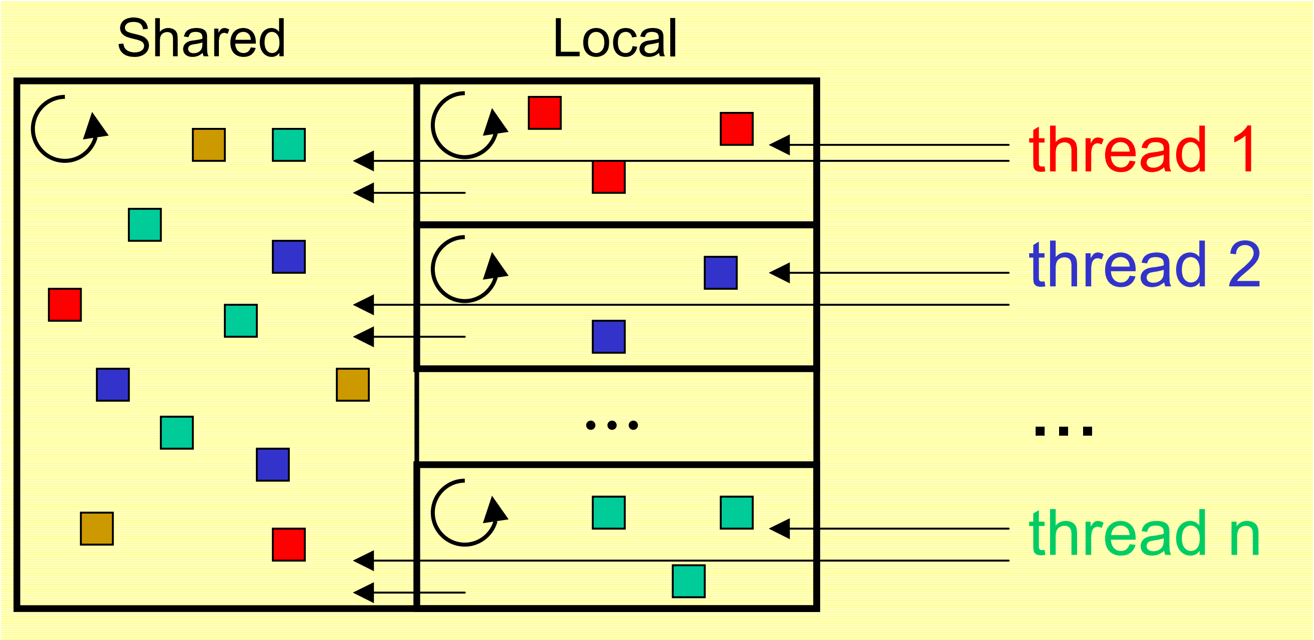
- Multithreading language
  - terms
  - types
- Limitations
- Insight into why it’s safe

Threads limitations

- Shared data enjoys an initialization phase
- Read-only data and reader/writer locks
- Object migration
- Global variables need top-level locks
- Semaphores, signals, ...
- Deadlock (not a safety problem)
- ...

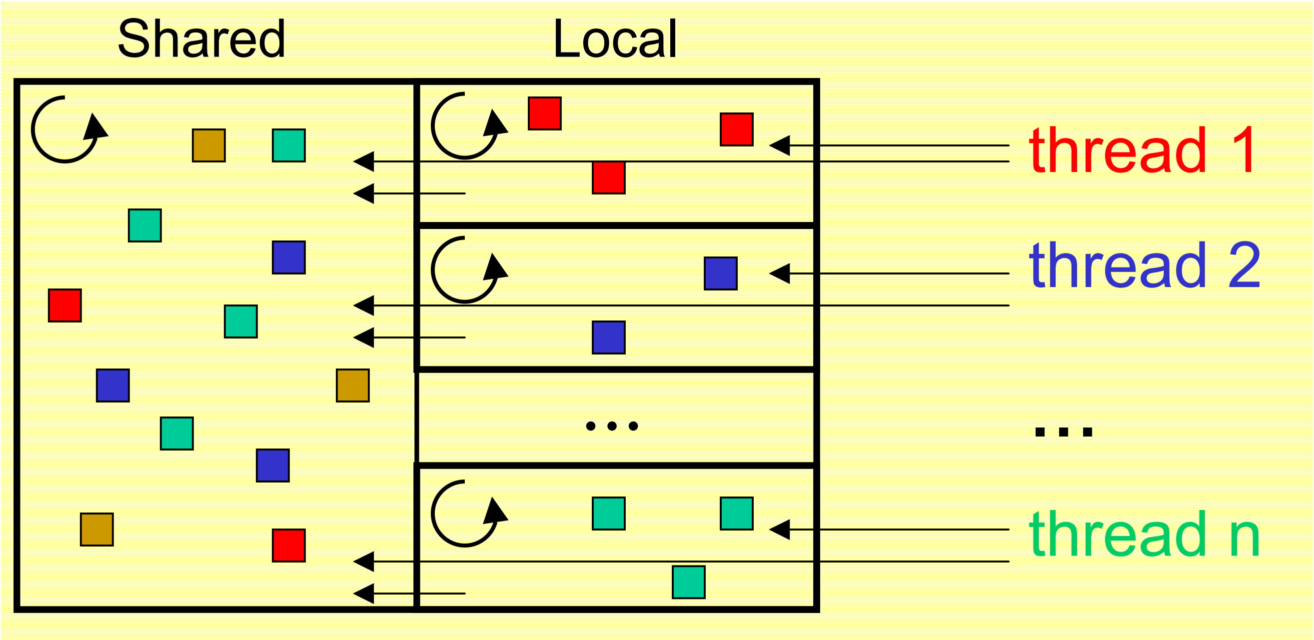
Why it works

There is one shared heap with implicit structure



- spawn preserves structure because of shareabilities
- each thread accesses only its “color”
- lock acquire/release changes some objects’ “color”

Why it works, continued



- objects changing color are not being mutated
- so no data races occur
- basis for a formal proof for an abstract machine
- structure is for the proof – colors/boxes don't "exist"

The plan from here

- Goals for the type system
- Safe multithreading
- Region-based memory management
- Evaluation (single-threaded)
- Related work
- Future directions

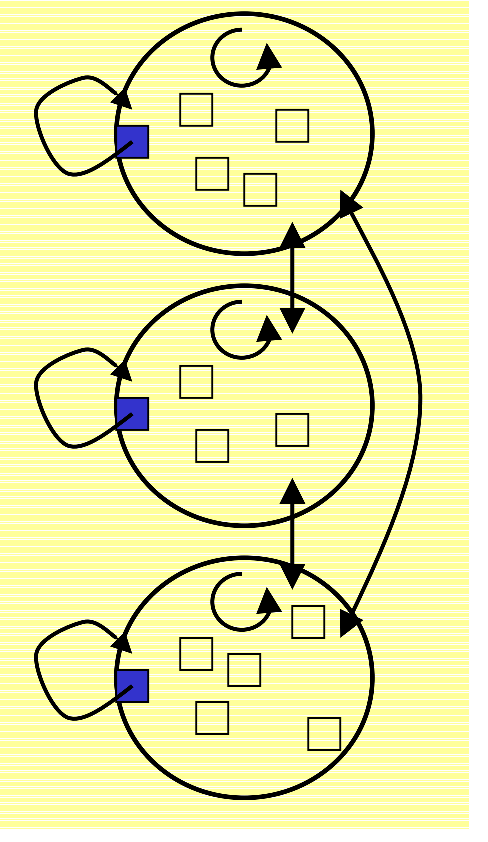
Memory reuse: the problem

Dereferencing dangling pointers breaks safety:

```
void f() {
    int* x;
    {
      int  y = 0;
      x = &y; // x not dangling
    } // x dangling
    {
      int* z = NULL;
      *x = 123;
      ...
    }
}
```

Regions

- a.k.a. zones, arenas, …
- Each object is in exactly one region
- Allocation via a region *handle*
- Deallocate an entire region simultaneously
  (cannot **`free`** an object)
- Type system [Tofte/Talpin]
  - types for handles and pointers use region names
  - should sound familiar



Region contributions [PLDI 02]

1. Integrate heap, stack, and user-defined regions
   – RC [Gay/Aiken 01] not for heap or stack

2. Usable source language
   – MLKit: compiler intermediate language
   – Walker et al.: assembly language

3. New approach to abstract types

4. Subtyping based on "outlives"

Cyclone regions

- 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

Region names

- Annotate all pointer types with a *region name*
- `int*`r` means "pointer into the region named `` `r ``"
  - heap has name `` `H ``
  - `l:`… has name `` `l ``
  - `{region r; s}` has name `` `r ``
    `r` has type `region_t<`r>`

Safety via scoping (almost)

```
void f() {
 int*__    x;

 l:{int     y;
     int*`l p = &y;
     x = p;
   }
  ...
}
```

- What region name for type of x ?
- `` `l `` is not in scope at allocation point
- scoping insufficient in general
- But system is equivalent to "scoping rule" unless you use first-class abstract types

Power via quantified types

- Universal quantification lets code take stack, region, and heap pointers

- Example: swap *exactly* like in C

```
void swap(int*`r1 x, int*`r2 y);
```

- Existential types and type constructors too

A common framework

- `void f(lock_t<`L>, int*`L);`
- `void f(region_t<`r>,int*`r);`
- `void f(bound_t<`i>, int*`i);`
- `void f(void g(`a), `a);`

- Quantified types express invariants while permitting code reuse
- No hidden fields or checks
- Use flow analysis and alias information when invariants are too strong

The plan from here

- Goals for the type system
- Safe multithreading
- Region-based memory management
- Evaluation (single-threaded)
- Related work
- Future directions

Status

Cyclone really exists (except threads underway)

- >130K lines of Cyclone code, including the compiler
- gcc back-end (Linux, Cygwin, OSX, …)
- User's manual, mailing lists, …
- Still a research vehicle
- More features: exceptions, datatypes, …

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 <1 minute
- 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

- MediaNet [Hicks et al, OPENARCH2003]:
  - multimedia overlay network
  - servers written in Cyclone
  - needs quick data filtering
- Open Kernel Environment [Bos/Samwel, OPENARCH2002]
  - runs partially trusted code in a kernel
  - extended compiler, e.g., resource limits
  - uses regions for safe data sharing
- Windows device driver (6K lines)
  - 100 lines still in C (vs. 2500 in Vault [Fähndrich/DeLine])
  - unclear what to do when a run-time check fails
  - still many ways to crash a kernel (fewer with Vault)

The plan from here

- Goals for the type system
- Safe multithreading
- Region-based memory management
- Evaluation (single-threaded)
- Related work
- Future directions

Related work: higher and lower

- Adapted/extended ideas:
  - universal quantification [ML, Haskell, GJ, …]
  - existential quantification [Mitchell/Plotkin, …]
  - region types [Tofte/Talpin, Walker et al., …]
  - lock types [Flanagan et al., Boyapati et al.]
  - safety via dataflow [Java, …]
  - controlling data representation [Ada, Modula-3, …]
- Safe lower-level languages [TAL, PCC, …]
  - engineered for machine-generated code
    (TIC 2000, WCSSS 1999)

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
    - less user burden
    - less memory management, single-threaded
  - RTC [Yong/Horwitz]
- Splint [Evans], Metal [Engler]: unsound, but very useful
- SFI [Wahbe, Small, ...]: sandboxing via binary rewriting

Plenty left to do

- Resource exhaustion
  (e.g., stack overflow)
- User-specified aliasing properties
  (e.g., all aliases are known)
- More “compile-time arithmetic”
  (e.g., array initialization)
- Better error messages
  (not a beginner’s language)

Integrating more approaches

- My work uses types, flow analysis, and run-time checks for low-level safety

- Integrate and compare: model checking, metacompilation, type qualifiers, pointer logics, code rewriting, theorem proving, ...
  - many tools assume memory safety

- Cross-fertilization for languages, tools, and compilers

Beyond C code

A safe C-level language is only part of the battle

- Language interoperability
- Distributed, cross-platform, embedded computing
- Let programmers treat "code as data"
  - sensible tools for querying and transforming code
  - examples: modern linkers, data mining

Language research for managing heterogeneity

Summary

- Memory safety is essential, but the world relies on C
- Cyclone is a safe C-level language
- Today:
  - types to prevent races and dangling pointers
  - safe, powerful, scalable, usable
  - justified with design insight and empirical evidence

- To learn more:
  - http://www.cs.cornell.edu/projects/cyclone
  - write some code!

[Presentation ends here –
some auxiliary slides follow]

Our work cut out for us

To guarantee safety, we must address all sources of safety violations

Some of my favorites:

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

Example in Cyclone

```
void f(int@{`j} p, bound_t<`i> i, int v
          ; `i < `j){
     p[i] = v;
}
```

- @ for not-NULL
- regions and locks use implicit defaults (live and accessible)

Using existential types – arrays

```
struct SafeArr {<`i>
  bound_t<`i> len;
  int*{`i} arr;
};

// p has type struct SafeArr*
let SafeArr{<`i>.len=bd, .arr=a} = *p;
if(bd > i)
  a[i]=42;

// p2 can be longer or shorter
*p=*p2;

// e has type int*{37}
*p=SafeArr{.len=37, .arr=e};
```

Using existential types – locks

```
struct LkInt {<`L>
  lock_t<`L> lk;
  int*`L i;
};
// p has type struct LkInt*,
// `L not in scope
let LkInt{<`L>.lk=lk, .i=val} = *p;
sync lk { *val = 42; }
// p2 can use a different lock
*p=*p2;
// e has type int*loc
*p=SafeArr{.lk=nonlock, .i=e};
```

Using locks

```
∃`L. lock_t<`L> lk = newlock();
let nlk<`L1> = lk; // `L1 not in scope
int*`L1 p = e;
sync nlk {/* use *p */}
```

Not-null pointers

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

- Subtyping: `t@ < t*` but `t@@ ≮ 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

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)

Flow-Analysis 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 to uninitialized memory
  analysis computes must-points-to information
- under-defined evaluation order
  conservatively approximate all orders

Empirical results – numeric

Example	LOC	diff total	bugs	execution time (C=1)	faster	execution time (C=1)
grobner	3260	+257 –190 (41)	1	1.94x	+ 336 – 196	1.51x
cacm	340	+ 16 – 1 (13)		1.22x	+21 – 6	1.18x
cfrac	4057	+120 –105		2.12x	+ 141 – 110	1.93x
tile	1345	+108 – 94 (13)	1	1.32x	+ 110 – 95 3 in C	1.25

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

Empirical results – CCured-Olden

Example	LOC	diff total	bugs	execution time (C=1)	faster	execution time (C=1)
bisort	514	+11 – 9		1.03x		
treeadd	370	+21 –21		1.89x	+21 –21 nogc	1.15x
tsp	565	+ 9 – 9		1.11x		
mst	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

CCured results

- As published in POPL02
- I have not run CCured

Program	Ccured time (C=1)	Cyclone time (C=1)
bisort	1.03x	1.03x
treeadd	1.47x	1.89x / 1.15x
tsp	1.15x	1.11x
mst	2.05x	1.93x / 1.39x

"Incidental"

- `new` is a Cyclone keyword
- extraneous semicolons (`void f() {...};`)
- C missing prototypes (or omitting argument types)

- Nonincidental: distinguishing arrays of unknown length from those of length 1 (majority of changes)
  - other default would "improve" results

Why conservative GC?

- Would be inefficient because we:
  - Don’t tag data
  - Have full-width integers
  - Allow instantiating **`` `a ``** with **`int`**
- We allow dangling pointers
- We have not noticed false-pointer problems (yet?)
- Could compact with a mostly-copying scheme

Enforcing thread-local

- A possible type for spawn: spawn(<f>,<p>,<sz>)

```
void spawn(void f(`a*loc ;{}), `a*`L,
                    sizeof_t<`a> ;{`L});
```

- But not any `` `a `` will do – local must stay local!

- We already have different *kinds* of types:
  L for lock names
  A for (conventional) types

- Examples: `loc::L`, ``int*`L::A``, `struct T :: A`

Enforcing loc cont'd

- Enrich kinds with *shareabilities*, `S` or `U`
- `loc::LU`
- `newlock()` has type `∃`L::LS. lock_t<`L>`
- <u>A type is shareable only if every part is shareable</u>
- Unshareable is the default (allows every type)

```
void spawn<`a::AS>(void f(`a*;{}),
                    `a*`L,
                    sizeof_t<`a>
                    ;{`L});
```

// `` `a::AS `` means `` `a `` is a shareable type

Abstract Machine

Program state:

(H, L0, (L1,s1), ..., (Ln,sn))

- One heap H mapping variables to values
    (local vs. shared not a run-time notion)
- Li are disjoint lock sets: a lock is available (L0) or held by some thread
- A thread has held locks (Li) and control state (si)

Thread scheduling non-deterministic

– any thread can take the next primitive step

Dynamic semantics

- Single-thread steps can:
  - change/create a heap location
  - acquire/release/create a lock
  - spawn a thread
  - rewrite the thread's statement (control-flow)
- Mutation takes two steps. Roughly:

H maps x to v,	x=v′ ; s	$\Rightarrow$
H maps x to junk(v′),	x=junk(v′); s	$\Rightarrow$
H maps x to v′,	s	

- Data races can lead to stuck threads

Type system – source

- Type-checking (right) expressions:

$$\Delta;\Gamma;\varepsilon \vdash e : \tau$$

  Δ : lock names and their shareabilities

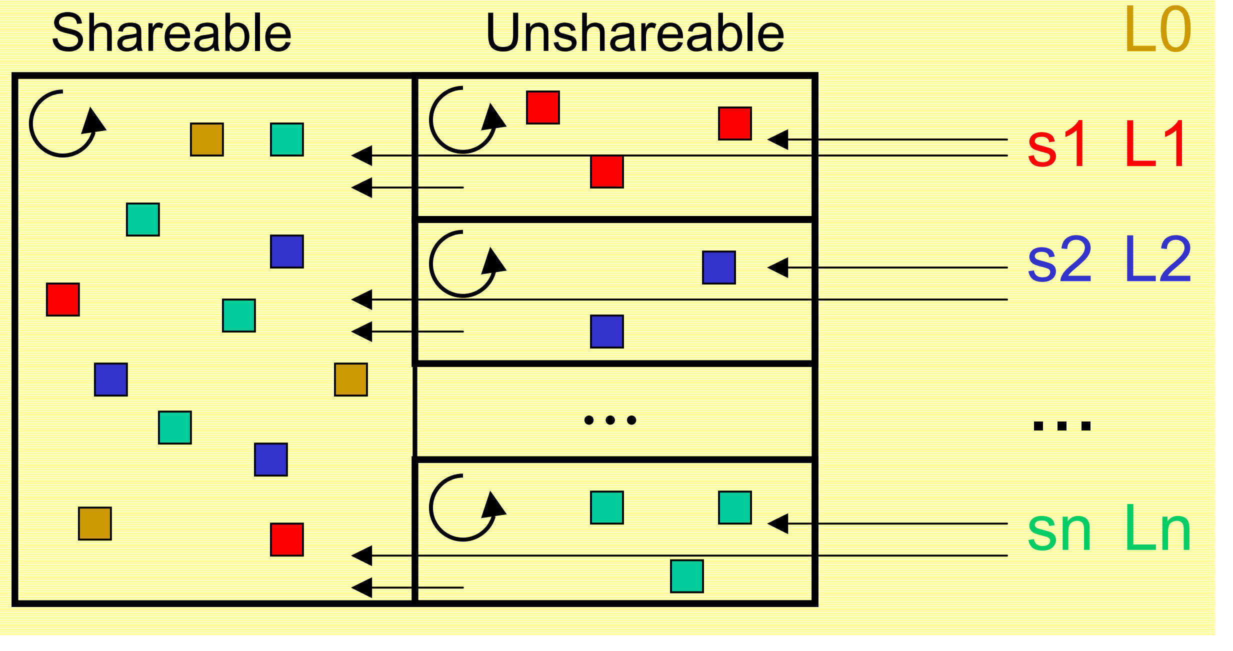
  Γ : term variables and their types & lock-names

  ε : names of locks that we know must be held

- junk expressions never appear in source programs
- Largely conventional

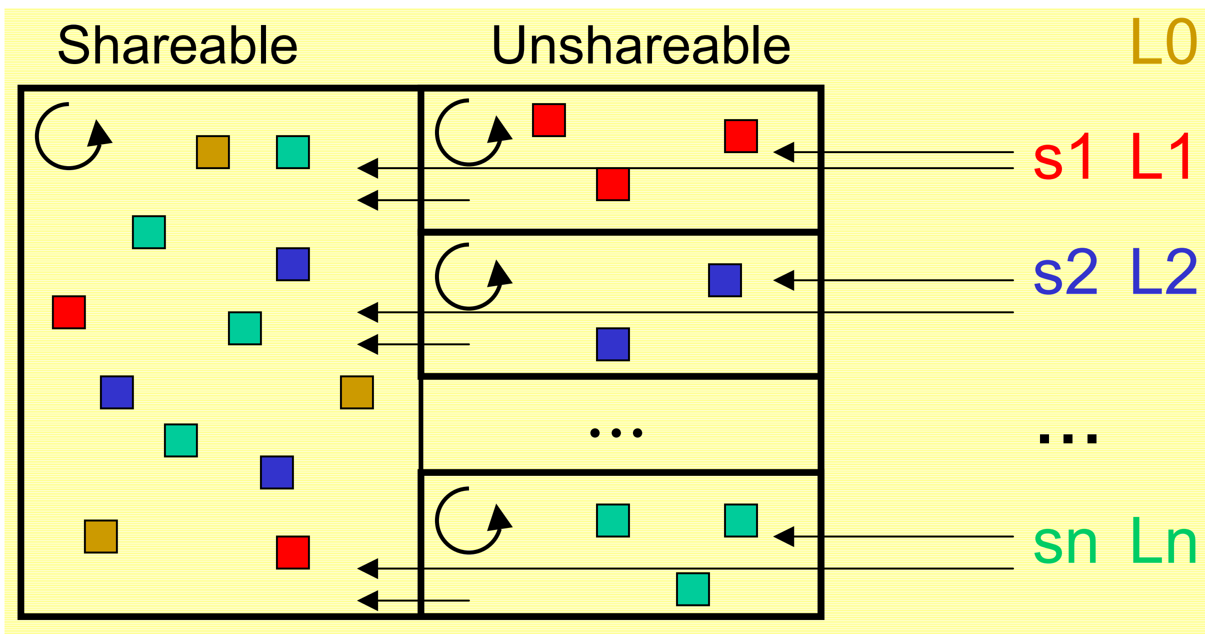
Type system – program state

- Evaluation preserves implicit structure on the heap
- spawn preserves the invariant because of its shareability restriction
- Lock acquire/release "recolors" the shared heap



No data races

- Invariant on where junk(v) appears:
  - Color has 1 junk if thread is mutating a location
  - Else color has no junk
- So no thread gets stuck due to junk



Formalism summary

- One “flat” run-time heap
- Machine models the data-race problem
- Straightforward type system for source programs (graduate student does the proof once)
- Proof requires understanding how the type system imposes structure on the heap...
- ... which helps explain “what’s really going on”

First proof for a system with thread-local data

Power via quantified types

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

int*`r newptr(region_t<`r> r, int x){
   return rnew(r,x);
}
```

- Default: different region name for each omission
- Existential types and type constructors too

To learn more:

- Cyclone: http://www.research.att.com/projects/cyclone
- My work: http://www.cs.cornell.edu/home/danieljg
- Cyclone publications
  - overview: USENIX 2002
    [Jim,Morrisett,Grossman,Hicks,Cheney,Wang]
  - existential types: ESOP 2002
    [Grossman]
  - regions: PLDI 2002
    [Grossman,Morrisett,Jim,Hicks,Wang,Cheney]
  - threads: TLDI 2003
    [Grossman]
- Write some code

Related: ownership types

Boyapati et al. concurrently developed similar techniques for locks, regions, and encapsulation in Java

- Cyclone
  - nonlock
  - no run-time type passing
  - support for parametric polymorphism
  - rigorous proof
- Ownership types
  - deadlock prevention
  - support for OO subtyping
  - object migration
  - “existentials” easier syntactically