

---

# Type-Safe Multithreading in Cyclone

Dan Grossman  
Cornell University  
TLDI 2003  
18 Jan 2003

---

# Cyclone + threads = ?

---

- Cyclone is a **safe** language at the **C** level
- Target domains often use **threads**
- Races are a notorious source of errors
  - Automated help would be welcome
- **Data races** can violate safety
  - Safety guarantee requires prevention

# Data races break safety

---

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

1. Pointer update must be atomic
  - shared-memory MP semantics must be sane
  - source pointers must translate to addresses
2. Writing addresses atomically insufficient

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

```
if (p->len > i)      ||| *p=*p2 // p2 longer  
    (p->arr) [i]=42; |||
```

# Preventing data races

---

- Dynamic
  - Detect races as they occur
  - Control scheduling and preemption
  - ...
- 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
  - ...

# Lock types

---

The type system ensures that:

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

- Flanagan, Abadi, Freund, Qadeer
  - invented basic approach
  - found real Java bugs
- Boyapati, Rinard, Lee
  - reuse code for shared and local data
  - advanced features I have not adapted

# Contributions

---

1. Adapt the approach to a C-level language
2. Integrate with parametric polymorphism
3. Integrate/compare with region-based memory management
4. Kinds to explain thread-local data and code reuse
5. Type-safety result for 1, 2, and 4 for an abstract machine where data races violate safety

# The plan from here

---

- Multithreading language
  - terms
  - types
  - kinds
- Limitations
- Formalism: insight into why it's safe
- Related work

# Multithreading terms

---

- **spawn** (**f**, **p**, **sz**) run **f** (**p2**) in a thread where **\*p2** is a shallow copy of **\*p** and **sz** is **sizeof(\*p)**
  - new thread starts with no locks held
  - new thread terminates when **f** returns
  - allows **\*p** to be thread-local
- **sync e s** acquire lock **e**, run **s**, release lock
- **newlock()** create a new lock
- **nonlock** a pseudolock for thread-local data

*Only sync requires language support  
(others are C terms with Cyclone types)*

# Simple examples (w/o types)

---

- Suppose `*p1` is shared (lock `lk`) and `*p2` is local
- Caller-locks:

```
void f(int *p) { /* use *p */ }  
sync lk { f(p1) };  
f(p2);
```

- Callee-locks:

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

# Types

---

- Obligation: Each shared memory *location* has a lock that is acquired before access
- Key: *Lock names* (types of kind **L**) in pointer types and lock types
  - `int* `L` is a type for pointers to locations guarded by a lock with type `lock_t<`L>`
  - mutual exclusion b/c `lock_t<`L>` is a *singleton*
- Thread-local locations use lock name `loc`
- `newlock()` has type `∃`L. lock_t<`L>`
- `nonlock` has type `lock_t<loc>`

# Access rights

---

- Obligation: Each shared memory location has a lock that is *acquired before access*
- Key: Each program point has a set of lock names
  - using location guarded by ``L` requires ``L` in set (`loc` is always in set)
  - `sync e s` adds ``L` if `e` has type `lock_t<`L>`
  - functions have explicit preconditions (default is caller locks)
- Lexical scope on `sync` convenient but nonessential

# Examples, with types

---

- Suppose `*p1` is shared (lock `lk`) and `*p2` is local
- Caller-locks:

```
void f(int* `L p ;{`L}) { /* use *p */  
    sync lk { f(p1) };  
    f(p2);  
}
```

- Callee-locks:

```
void g(int* `L p, lock_t<`L> l ;{ }) {  
    sync l { /*use *p */ }  
}  
g(p1, lk);  
g(p2, nonlock);
```

# Second-order lock types

---

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

```
struct LkInt {<`L> int*`L; lock_t<`L>;};
```

(same race problem as SafeArr example)

- Type constructors for reusing locks

```
struct Lst<`L> {  
    int*`L hd;  
    struct Lst<`L>*`L tl;  
};
```

- Easy to add because Cyclone had second-order types

# The plan from here

---

- Multithreading language
  - terms
  - types
  - kinds
- Limitations
- Formalism: insight into why it's safe
- Related work

*No data races only if local data is really local*

# Enforcing loc

---

- A possible type for spawn:

```
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:
  - `L` for locks
  - `A` for all (non-lock) types
- Examples: `loc :: L`, `int*`L :: A`, `struct T :: A`

# Enforcing loc cont'd

---

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

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

*Keeps local data local*

# Threads shortcomings

---

- Global variables need top-level locks
- 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

---

- Multithreading language
  - terms
  - types
  - kinds
- Limitations
- **Formalism: insight into why it's safe**
- Related work

# Abstract Machine

---

Program state:  $(H, L0, (L1,s1), \dots, (Ln,sn))$

- One heap (local vs. shared not a run-time notion)
- $L_i$  are disjoint lock sets: a lock is available ( $L0$ ) or held by some thread
- A thread has held locks ( $L_i$ ) and control state ( $s_i$ )

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-stack)
- Mutation takes two steps. Informally:
$$H[x \rightarrow v], \quad x = v' ; s \quad \Rightarrow$$
$$H[x \rightarrow \text{junk}(v')], x = \text{junk}(v') ; s \Rightarrow$$
$$H[x \rightarrow v'], \quad s$$
- Data races exist and can lead to stuck threads

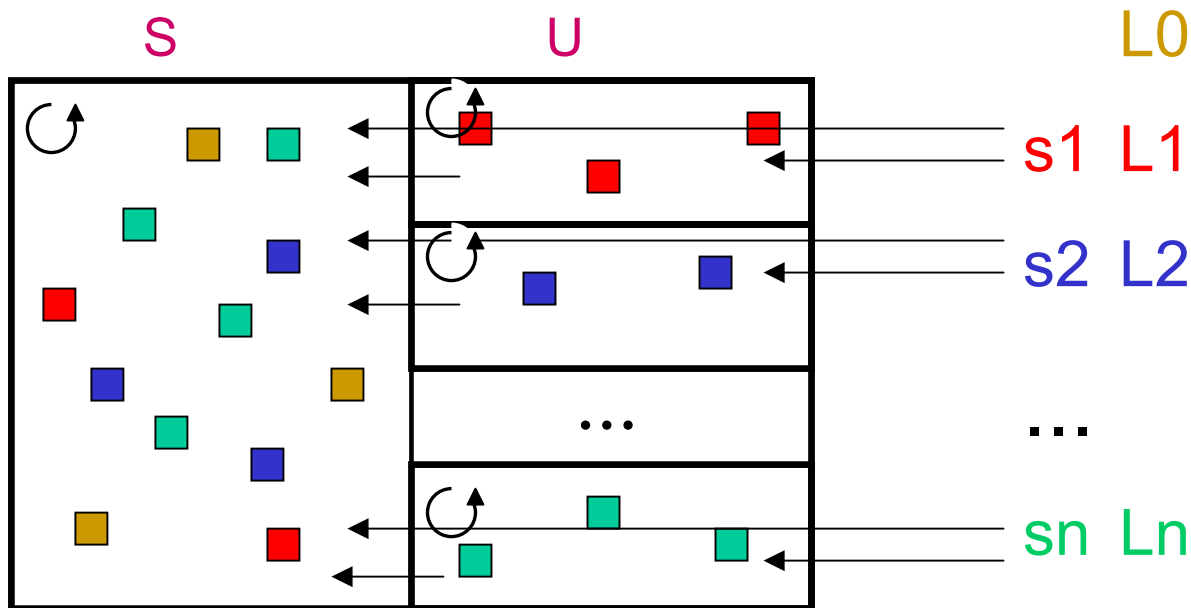
# Static semantics – source

---

- Distinguishes statements and left/right expressions (as does dynamic semantics and C)
- Type-checking right-expressions:  $\Delta; \Gamma; \gamma; \varepsilon \vdash e : \tau$ 
  - $\Delta$  : type variables and their kinds
  - $\Gamma$  : term variables and their types & lock-names
  - $\gamma$  : effect constraints (for polymorphism)
  - $\varepsilon$  : effect (lock names currently allowed)
- junk expressions never well-typed in source
- Largely conventional – no surprises

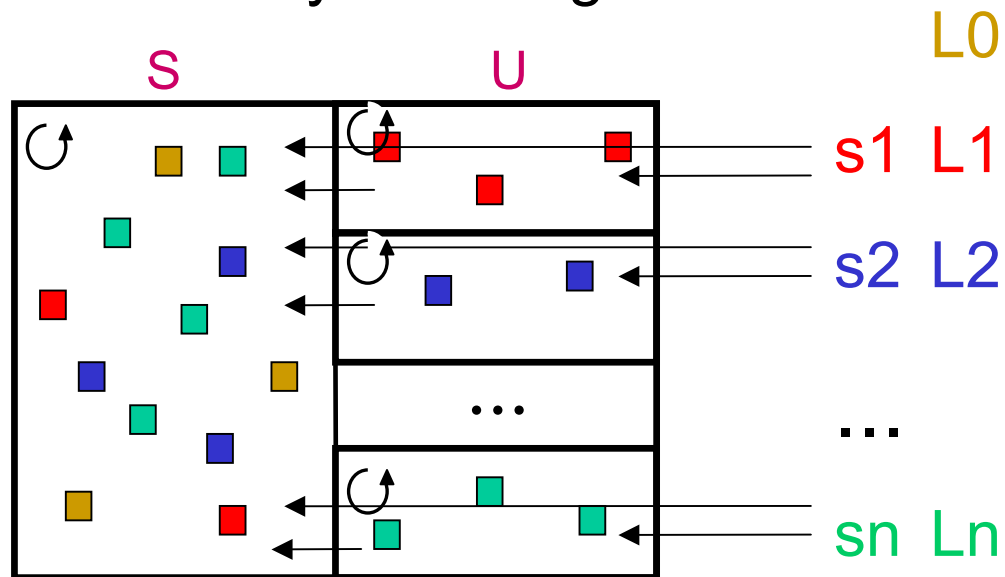
# Static semantics – program state

- Evaluation preserves **implicit structure on the heap**
- spawn preserves the invariant because of the kind restriction on its argument
- Acquiring/releasing a lock “recolors” the shared heap



# No data races

- Invariant on where junk(v) can appear:
  - Color has one junk if  $s_i$  is mutating an element
  - Else color is junk-free
- So no thread gets stuck due to junk
- Theorem: thread stuck only if waiting on lock  
(can deadlock)



# Formalism summary

---

- One run-time heap (colors and boxes for the proof)
- A trick for making data races a problem
- Straightforward type system for source programs
- Syntactic safety proof requires understanding how the type system imposes structure on the heap...
- ... which was invaluable in understanding, “what’s really going on” especially with spawn
  
- First proof for a system with thread-local data

# Related work

---

- This work in line of Flanagan, Boyapati, et al.
- Guava (race-free Java, rigid local/shared distinction)
- Bug-finding tools (ESC/Java, Warlock, ...)
- Dynamic race detection (novel code and run-time)
  
- Other safe low-level languages (CCured, Vault, PCC, TAL, ...) single-threaded

*Cannot implement an array-bounds check in the presence of data races*

# Conclusions

---

- Data races and safe C do not mix well
- Static support for lock-based mutual exclusion becoming well understood
- Designed and formalized multithreading for Cyclone
- May need more bells and whistles for realistic multithreaded programs
- Implementation high on the to-do list

---

[End of Presentation – an auxiliary slide follows]

# Important extensions (see the paper)

---

- Parametric polymorphism
  - What locks are needed to access an `a`
  - How do we ensure these locks are acquired while allowing polymorphic code
- Region-based memory management
  - How do we prevent one thread from accessing objects another thread has deallocated
  - Type systems for regions and locks very similar, so what do the differences teach us