Type-Safety, Concurrency, and Beyond: Programming-Language Technology for Reliable Software

Dan Grossman
University of Washington

15 February 2005
PL for Better Software

• Software is part of society’s critical infrastructure
  Where we learn of security lapses:
  bboards → tech news → business-page → front-page

• PL is uniquely positioned to help. “We own”:
  – The build process and run-time
  – Intellectual tools to prove program properties

• But solid science/engineering is key
  – The UMPLFAP* solution is a non-starter
  – Crisp problems and solutions

*Use My Perfect Language For All Programming
Better low-level code

My focus for the last \( n \) years:  

*bring type-safety to low-level languages*

- For some applications, C remains the best choice (!)
  - Explicit data representation
  - Explicit memory management
  - Tons of legacy code
- But C without the dangerous stuff is too impoverished
  - No arrays, threads, null-pointers, varargs, …
- Cyclone: a safe, modern language at the C-level
  - A necessary but insufficient puzzle piece
Beyond low-level type safety

0. Brief Cyclone overview
   – Synergy of types, static analysis, dynamic checks (example: not-NULL pointers)
   – The need for more (example: data races)

1. Better concurrency primitives (AtomCAML)

   Brief plug for:

2. A C-level module system (CLAMP)

3. Better error messages (SEMINAL)

   Research that needs doing and needs eager, dedicated, clever people
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
- **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”
Status

Cyclone really exists (except memory-safe threads)
• >150K lines of Cyclone code, including the compiler
  – Compiles itself in 30 seconds
• Targets gcc
  (Linux, Cygwin, OSX, OpenBSD, Mindstorm, Gameboy, …)
• User’s manual, mailing lists, …
• Still a research vehicle
Example 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)

- Always looking for systems projects that would benefit from Cyclone

www.research.att.com/projects/cyclone
Not-null pointers

<table>
<thead>
<tr>
<th>t*</th>
<th>pointer to a t value or NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>t@</td>
<td>pointer to a t value</td>
</tr>
</tbody>
</table>

- Subtyping: t@ < t* but t@@ ≠ t*@

- 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");
    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 safely just via types
It’s always aliasing

```c
void f(int**@p) {
    if(*p != NULL) {
        g();
        **p = 42; //inserts 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)
It’s always aliasing

```c
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)
Data-race example

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

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

```c
data example
15 February 2005 Dan Grossman
15

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

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

change
p1->len
to 5

change
p1->arr

*p1 = *p2;

change p1->len to 5

change p1->arr
```
Data-race example

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

If `p1->len` is greater than 4:
- Check `p1->len > 4`.

Change `p1->len` to 5:
- Check `p1->len`.

Change `p1->arr`.

* `p1 = *p2`.

Change `p1->len` to 5.
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, Boyapati et al]

- Adaptation to Cyclone works out
  - See my last colloquium talk (March 2003)
  - But locks are the wrong thing for reliable concurrency
Cyclone summary

Achieving memory safety a key first step, but

1. Locks for memory safety is really weak (applications always need to keep multiple objects synchronized)
   – Solve the problem for high-level PLs first

2. A million-line system needs more modularity than “no buffer overflows”

3. Fancy types mean weird error messages and/or buggy compiler

Good news: 3 new research projects
Atomicity overview

• Why “atomic” is better than mutual-exclusion locks
  – And why it belongs in a language

• How to implement atomic on a uniprocessor

• How to implement atomic on a multiprocessor
  – Preliminary ideas that use locks cleverly

Foreshadowing:
  – hard part is efficient implementation
  – key is cheap logging and rollback
Threads in PL

• Positive shift: Threads are a C library and a Java language feature

• But: Locks are an error-prone, low-level mechanism that is a poor match for much programming
  – Java programs/libraries full of races and deadlocks
  – Java 1.5 just provides more low-level mechanisms

• Target domain: Apps that use threads to mask I/O latency and provide responsiveness (e.g., GUls)
  – Not high-performance scientific computing
Atomic

An easier-to-use and harder-to-implement primitive:

```java
void deposit(int x) {
    synchronized(this) {
        int tmp = balance;
        tmp += x;
        balance = tmp;
    }
}
```

semantics:
lock acquire/release

```java
atomic {
    int tmp = balance;
    tmp += x;
    balance = tmp;
}
```

semantics:
(behave as if)
no interleaved execution

No fancy hardware, code restrictions, deadlock, or unfair scheduling (e.g., disabling interrupts)
6.5 ways atomic is better

1. Atomic makes deadlock less common

```java
transfer(Acct that, int x){
  synchronized(this){
    synchronized(that){
      this.withdraw(x);
      that.deposit(x);
    }
  }
}
```

- Deadlock with parallel "untransfer"
  - Sun JDK had this for buffer append!
- Trivial deadlock if locks not re-entrant
- 1 lock at a time \(\Rightarrow\) race with "total funds available"
6.5 ways atomic is better

2. Atomic allows modular code evolution
   – Race avoidance: global object → lock mapping
   – Deadlock avoidance: global lock-partial-order

   // x, y, and z are globals
   void foo() {
     synchronized(???){
       x.f1 = y.f2 + z.f3;
     }
   }

   • Want to write foo to be race and deadlock free
     – What locks should I acquire? (Are y and z immutable?)
     – In what order?
6.5 ways atomic is better

3. Atomic localizes errors
   (Bad code messes up only the thread executing it)

```java
void bad1() {
    x.balance = -1000;
}

void bad2() {
    synchronized(lk) {
        while(true) ;
    }
}
```

- Unsynchronized actions by other threads are invisible to atomic
- Atomic blocks that are too long may get starved, but won’t starve others
  - Can give longer time slices
6.5 ways atomic is better

4. Atomic makes abstractions thread-safe without committing to serialization

```java
class Set { // synchronization unknown
  void insert(int x) {...}
  bool member(int x) {...}
  int size() {...}
}
```

To wrap this with synchronization:
Grab the same lock before any call. But:
- Unnecessary: no operations run in parallel (even if member and size could)
- Insufficient: implementation may have races
6.5 ways atomic is better

5. Atomic **is usually what programmers want**
   
   Flanagan, Qadeer, Freund

   - Vast majority of Java methods marked `synchronized` are actually atomic
   - Of those that aren’t, vast majority of races are application-level bugs
   - `synchronized` is an implementation detail
     - does not belong in interfaces (atomic does)!

```
interface I { synchronized int m(); }
class A { synchronized int m() { // an I
    <<call code with races>>
} }
class B { int m() { return 3; } } // not an I
```
6.5 ways atomic is better

6. Atomic can efficiently implement locks

```java
class Lock {
    bool b = false;
    void acquire() {
        while(true) {
            while(b) /*spin*/;
            atomic {
                if(b) continue;
                b = true;
                return;
            }
        }
    }
    void release() {
        b = false;
    }
}
```

- Cute O/S homework problem
- In practice, implement locks like you always have
- Atomic and locks peacefully co-exist
  - Use both if you want
6.5 ways atomic is better

6.5 Concurrent programs have the granularity problem:

- Too little synchronization:
  non-determinism, races, bugs
- Too much synchronization:
  poor performance, sequentialization

Example: Should a chaining hashtable have one lock, one lock per bucket, or one lock per entry?

**atomic** doesn’t solve the problem, but makes it easier to mix coarse-grained and fine-grained operations
Atomicity overview

• Why “atomic” is better than mutual-exclusion locks
  – And why it belongs in a language

• How to implement atomic on a uniprocessor

• How to implement atomic on a multiprocessor
Interleaved execution

The “uniprocessor” assumption:

*Threads communicating via shared memory don't execute in “true parallel”*

Actually more general than uniprocessor: threads on different processors can pass messages

An important special case:

- Many language implementations make this assumption
- Many concurrent apps don’t need a multiprocessor (e.g., a document editor)
- If uniprocessors are dead, where’s the funeral?
Implementing atomic

Key pieces:

• Execution of an atomic block logs writes

• If scheduler pre-empts a thread in an atomic block, rollback the thread

• Duplicate code so non-atomic code is not slowed down by logging/rollback

• In an atomic block, buffer output and log input – Necessary for rollback but may be inconvenient
Logging example

```c
int x=0, y=0;
void f() {
    int z = y+1;
    x = z;
}
void g() {
    y = x + 1;
}
void h() {
    atomic {
        y = 2;
        f();
        g();
    }
}
```

- Executing atomic block in `h` builds a LIFO log of old values:

```
y:0  z:?  x:0  y:2
```

Rollback on pre-emption:
- Pop log, doing assignments
- Set program counter and stack to beginning of atomic

On exit from atomic: drop log
Keeping the log small:
- Don’t log reads (key uniprocessor optimization)
- Don’t log memory allocated after atomic was entered (in particular, local variables like \( z \))
- No need to log an address after the first time
  - To keep logging fast, only occasionally “trim”
- Tell programmers non-local writes cost more

Keeping logging fast: Simple resizing or chunked array
Duplicate code so callees know to log or not:

- For each function $f$, compile $f_{\text{atomic}}$ and $f_{\text{normal}}$
- Atomic blocks and atomic functions call atomic functions
- Function pointers (e.g., vtables) compile to pair of code pointers

Cute detail: compiler erases any atomic block in $f_{\text{atomic}}$
Qualitative evaluation

- Non-atomic code executes unchanged
- Writes in atomic block are logged (2 extra writes)
- Worst case code bloat of 2x

- Thread scheduler and code generator must conspire

- Still have to deal with I/O
  - Atomic blocks probably shouldn’t do much
Handling I/O

• Buffering sends (output) is easy and necessary
• Logging receives (input) is easy and necessary
  – And may as well rollback if the thread blocks
• But may miss subtle non-determinism:

```c
void f() {
    write_file_foo(); // flushed?
    read_file_foo();
}

void g() {
    atomic {f();} // read won't see write
    f();          // read may see write
}
```

• Alternative: receive-after-send-in-atomic throws exception
Prototype

• AtomCAML: modified OCaml bytecode compiler
• Advantages of mostly functional language
  – Fewer writes (don’t log object initialization)
  – To the front-end, atomic is just a function

\[
\text{atomic : (unit -> 'a) -> 'a}
\]

• Key next step: port applications that use locks
  – Planet active network from UPenn
  – MetaPRL logical framework from CalTech
Atomicity overview

• Why “atomic” is better than mutual-exclusion locks
  – And why it belongs in a language

• How to implement atomic on a uniprocessor

• How to implement atomic on a multiprocessor
A multiprocessor approach

• Give up on zero-cost reads
• Give up on safe, unsynchronized accesses
  – All shared-memory access must be within atomic
    (conceptually; compiler can insert them)
• But: Try to minimize inter-thread communication

Strategy: Use locks to implement atomic
• Each shared object guarded by a readers/writer lock
  – Key: many objects can share a lock
• Logging and rollback to prevent deadlock
Example redux

int x=0, y=0;
void f() {
    int z = y+1;
    x = z;
}
void g() {
    y = x + 1;
}
void h() {
    atomic {
        y = 2;
        f();
        g();
    }
}

• Atomic code acquires lock(s) for x and y (1 or 2 locks)
• Release locks on rollback or completion
• Avoid deadlock automatically. Possibilities:
  – Rollback on lock-unavailable
  – Scheduler detects deadlock, initiates rollback
• Only 1 problem…
What locks what?

There is little chance any compiler in my lifetime will infer a decent object-to-lock mapping

– More locks = more communication
– Fewer locks = less parallelism
What locks what?

There is little chance any compiler in my lifetime will infer a decent object-to-lock mapping

- More locks = more communication
- Fewer locks = less parallelism
- Programmers can’t do it well either, though we make them try
What locks what?

There is little chance any compiler in my lifetime will infer a decent object-to-lock mapping

When stuck in computer science, use 1 of the following:

a. Divide-and-conquer
b. Locality
c. Level of indirection
d. Encode computation as data
e. An abstract data-type
Locality

Hunch: Objects accessed in the same atomic block will likely be accessed in the same atomic block again

• So while holding their locks, change the object-to-lock mapping to share locks
  – Conversely, detect false contention and break sharing

• If hunch is right, future atomic block acquires fewer locks
  – Less inter-thread communication
  – And many papers on heuristics and policies 😊
Related Work on Atomic

Old ideas:
• Transactions in databases and distributed systems
  – Different trade-offs and flexibilities
• Rollback for various recoverability needs
• Atomic sequences to implement locks [Bershad et al]
• Atomicity via restricted sharing [ARGUS]

Rapid new progress:
• Atomicity via shadow-memory & versioning [Harris et al]
• Checking for atomicity [Qadeer et al]
• Transactional memory in SW [Herlihy et al] or HW [tcc]

PLDI03, OOPSLA03, PODC03, ASPLOS04, …
Beyond low-level type safety

0. Brief Cyclone overview
   – Synergy of types, static analysis, dynamic checks
   – The need for more
1. Better concurrency primitives

Brief plug for:
2. A C-level module system (CLAMP)
3. Better error messages (SEMINAL)

Research that needs doing and needs eager, dedicated, clever people
Clamp

Clamp is a C-like Language for Abstraction, Modularity, and Portability (it holds things together)

Go beyond Cyclone by using a module system to encapsulate low-level assumptions, e.g.,:
- Module X assumes big-endian 32-bit words
- Module Y uses module X
- Do I need to change Y when I port?

(Similar ideas in Modula-3 and Knit, but no direct support for the data-rep levels of C code.)

Clamp doesn’t exist yet; there are many interesting questions
What happens:

1. A researcher implements an elegant new analysis in a compiler that is great for correct programs.

2. But the error messages are inscrutable, so the compiler gets hacked up:
   - Pass around more state
   - Sprinkle special cases and strings everywhere
   - Slow down the compiler
   - Introduce compiler bugs

Recently I fixed a dangerous bug in Cyclone resulting from not type-checking $e \rightarrow f$ as $(*e).f$
A new approach

• One solution: 2 checkers, trust the fast one, use the other for messages
  – Hard to keep in sync; slow one no easier to write
• SEMINAL*: use fast one as a subroutine for search:
  – Human speed (1-2 seconds)
  – Find a similar term (with holes) that type-checks
    • Easier to read than types
    • Offer multiple ranked choices
• Example: “f(e1,e2,e3) doesn’t type-check, but f(e1,_,e3) does and f(e1,e2->foo,e3) does”
• Help! (PL, compilers, AI, HCI, …)

*Searching for Error Messages in Advanced Languages
Summary

• We must make it easier to build large, reliable software
  – Current concurrency technology doesn’t
  – Current modules for low-level code doesn’t
  – Type systems are hitting the error-message wall

• Programming-languages research is fun
  – Ultimate blend of theory and practice
  – Unique place in “tool-chain control”
  – Core computer science with much work remaining
Acknowledgments

• Cyclone is joint work with Greg Morrisett (Harvard), Trevor Jim (AT&T Research), Michael Hicks (Maryland)
  – Thanks: Ben Hindman for compiler hacking

• Atomicity is joint work with Michael Ringenburg
  – Thanks: Cynthia Webber for some benchmarks
  – Thanks: Manuel Fähndrich and Shaz Qadeer (MSR) for motivating us

• For updates and other projects: www.cs.washington.edu/research/progsys/wasp/