Programming-Language Motivation, Design, and Semantics for Software Transactions

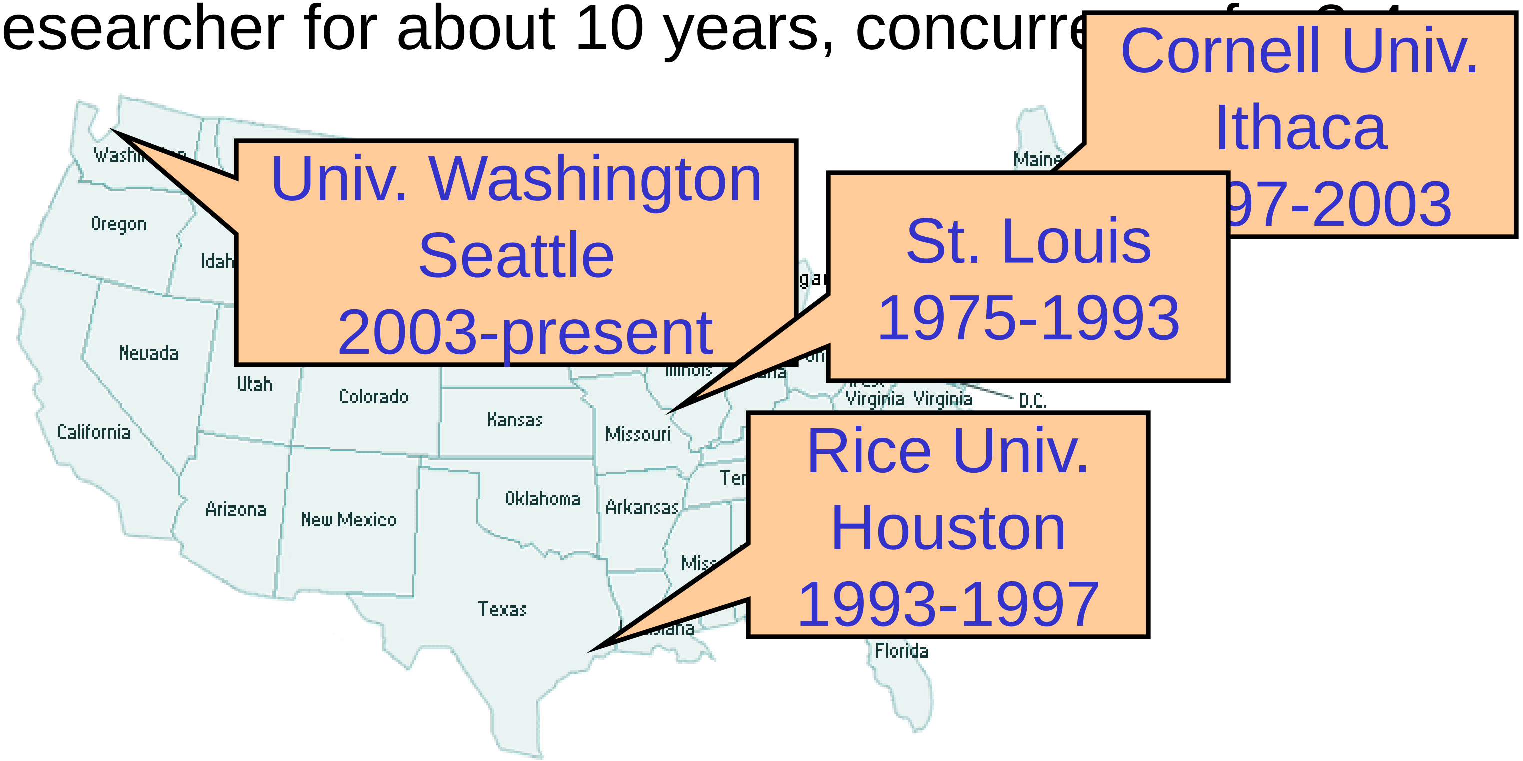
Dan Grossman
University of Washington

June 2008

Me in 2 minutes

Excited to be here & give “my PL view on transactions”

A PL researcher for about 10 years, concurre



Univ. Washington
Seattle
2003-present

Cornell Univ.
Ithaca
97-2003

St. Louis
1975-1993

Rice Univ.
Houston
1993-1997

Atomic

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

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

lock acquire/release

```
void deposit(int x){
atomic {
  int tmp = balance;
  tmp += x;
  balance = tmp;
}}
```

(behave as if)
no interleaved computation

PL Perspective

Complementary to lower-level implementation work

Motivation:

- The essence of the advantage over locks

Language design:

- Rigorous high-level semantics
- Interaction with rest of the language

Language implementation:

- Interaction with modern compilers
- New optimization needs

Answers urgently needed for the multicore era

My tentative plan

1. Basics: language constructs, implementation intuition (Tim next week)

2. Motivation: the TM/GC Analogy

3. Strong vs. weak atomicity
   - And optimizations relevant to strong

4. Formal semantics for transactions / proof results
   - Including formal-semantics tutorial

5. Brief mention: memory-models, continuations

Time not evenly divided among these topics

Related work

- Many fantastic papers on transactions
  - And related topics
- *Lectures* borrow heavily from my research and others
  - Examples from papers and talks I didn’t write
  - Examples from work I did with others
- See my papers and TM Online for proper citation
  - Purpose here is to prepare you to understand the literature
  - **`www.cs.wisc.edu/trans-memory/`**

Basics

- Basic semantics

- Implementation intuition
  - Many more details/caveats from Tim

- Interaction with other language features

Informal semantics

```
atomic {
    s // some statement
}
```

- `atomic{s}` runs `s` “all-at-once with no interleaving”
  - isolation and atomicity
  - syntax unimportant (maybe a function or an expression or an annotation or …)
- `s` can do almost anything
  - read, write, allocate, call, throw, …
  - Ongoing research: I/O and thread-spawn

Parallelism

- Performance “guarantee” rarely in language specs
  - But programmers need informal understanding

- Transactions (atomic blocks) can run in parallel if there are no *memory conflicts*
  - Read and write of “same memory”
  - Write and write of “same memory”

- *Granularity* matters
  - word vs. object vs. cache line vs. hashing
  - *false sharing* ⇒ unpredictable performance

Easier fine-grained parallelism

- Fine-grained locking
  - lots of locks, hard to get code right
  - but hopefully more parallel critical sections
  - *pessimistic:* acquire lock if might access data
- Course-grained locking
  - Fewer locks, less parallelism
- Transactions:
  - parallelism based on dynamic memory accessed
  - *optimistic:* abort/retry when conflict detected
    - should be hidden from programmers

Retry

```
class Queue {
  Object[] arr    = …;
  int      front  = …;
  int      back   = …;
  boolean isFull() { return front==back; }
  boolean isEmpty(){ return … }
  void enqueue(Object o) {
    atomic {
      if(isFull())
        retry;
      …
    }
  }
  // dequeue similar with isEmpty()
}
```

Retry

- Let programmers cause retry
  - great for waiting for conditions
- Compare to *condition variables*
  - *retry* serves role of *wait*
  - No explicit *signal* (*notify)*
    - Implicit: something transaction read is updated
    - Performance: best not to retry transaction until something has changed (?)
      - not supported by all current implementations
  - Drawback: no signal vs. broadcast (notifyAll)

Basics

- Basic semantics
- Implementation intuition
  - Many more details/caveats from Tim
- Interaction with other language features

Track what you touch

High-level ideas:

- Maintain transaction's *read set*
  - so you can *abort* if another thread writes to it before you *commit* (detect *conflicts*)
- Maintain transaction's *write set*
  - again for *conflicts*
  - also to *commit* or *abort* correctly

Basic trade-offs

- Two approaches to writes
  - *Eager update*
    - update in place, “own until commit” to prevent access by others
    - log previous value; undo update if abort
    - if owned by another thread, abort to prevent *deadlock* (livelock is possible)
  - *Lazy update*
    - write to private buffer
    - reads must check buffer
    - abort is trivial
    - commit is fancy to ensure “all at once”

Basic Trade-offs

- Reads
  - May read an *inconsistent value*
    - detect with version numbers and such
    - inconsistent read requires an abort
    - but can detect & abort lazily, allowing *zombies*
    - implementation must be careful about zombies

```
    initially x=0, y=0
atomic {          atomic {
  while(x!=y)       ++x;
    ;               ++y;
}                 }
```

Basics

- Basic semantics

- Implementation intuition
  - Many more details/caveats from Tim

- Interaction with other language features

Language-design issues

- Interaction with exceptions
- Interaction with native-code
- Closed nesting (flatten vs. partial rollback)
- Escape hatches and open nesting
- Multithreaded transactions
- The **`orelse`** combinator
- **`atomic`** as a first-class function

Exceptions

If code in `atomic` raises exception caught outside `atomic`, does the transaction abort and/or retry?

I say no! (others disagree)

- atomic = “no interleaving until control leaves”
- Else `atomic` changes meaning of 1-thread programs

```
int x = 0;
try {
  atomic { ++x; f(); }
} catch (Exception e) {}
assert(x==1);
```

Other options

Alternative semantics

1. Abort + retry transaction
   - Easy for programmers to encode (& vice-versa)

```
atomic {
  try { s }
  catch (Throwable e) { retry; }
}
```

2. Undo transaction’s memory updates, don’t retry
   - Transfer to catch-statement instead
   - Makes little sense:
     - Transaction “didn’t happen”
     - What about the exception object itself?

Handling I/O

- Buffering sends (output) easy and necessary
- Logging receives (input) easy and necessary

- But input-after-output still doesn’t work

```
void f(){
 write_file_foo();
 …
 read_file_foo();
}
void g(){
  atomic{f();} //read won’t see write
  f();         //read may see write
}
```

- I/O one instance of native code …

Native mechanism

- Most current systems: halt program on native call
  - Should at least not fail on zombies
- Other imperfect solutions
  - Raise an exception
  - Make the transaction “irrevocable” (unfair)
- A pragmatic partial solution: Let the C code decide
  - Provide 2 functions (in-atomic, not-in-atomic)
  - in-atomic can call not-in-atomic, raise exception, cause retry, or do something else
  - in-atomic can *register* commit- & abort- actions
    - sufficient for buffering

Language-design issues

- Interaction with exceptions
- Interaction with native-code
- Closed nesting (flatten vs. partial rollback)
- Escape hatches and open nesting
- Multithreaded transactions
- The **`orelse`** combinator
- **`atomic`** as a first-class function

Closed nesting

One transaction inside another has no effect!

```
void f() { … atomic { … g() … } }
void g() { … h() … }
void h() { … atomic { … } }
```

- Flattened “nesting”: treat inner `atomic` as a no-op
  - – Retry aborts outermost (never prevents progress)
- Retry to innermost (“partial rollback”) could avoid some recomputation via extra bookkeeping
  - – *May* be more efficient

Partial-rollback example

(Contrived) example where aborting inner transaction:

- is useless
- only aborting outer can lead to commit

Does this arise “in practice”?

```
atomic {
  y = 17;
  if(x > z)
    atomic {
      if (x > y)
        retry;
      …
    }
}
```

- Inner cannot succeed until **x** or **y** changes
- But **x** or **y** changing “dooms” outer

Language-design issues

- Interaction with exceptions
- Interaction with native-code
- Closed nesting (flatten vs. partial rollback)
- Escape hatches and open nesting
- Multithreaded transactions
- The **`orelse`** combinator
- **`atomic`** as a first-class function

Escape hatch

```
atomic { … escape { s; } … }
```

Escaping is a total cheat (a “back door”)

- Reads/writes don’t count for outer’s conflicts
- Writes happen even if outer aborts

Arguments against:

- It’s not a transaction anymore!
- Semantics poorly understood
- May make implementation optimizations harder

Arguments for:

- Can be correct at application level and more efficient
- Useful for building a VM (or O/S) with only **`atomic`**

Example

I am not a fan of language-level escape hatches (too much unconstrained power!)

But here is a (simplified) canonical example:

```
class UniqueId {
  private static int g = 0;
  private int myId;
  public UniqueId() {
    escape { atomic { myId = ++g; } }
  }
  public boolean compare(UniqueId i){
    return myId == i.myId;
  }
}
```

The key problem (?)

- *Write-write conflicts* between outer transaction and escape
  - Followed by abort

```
atomic {
   … ++x; …
   escape { … ++x; … }
   … ++x; …
}
```

- Such code is likely “wrong” but need some definition
  - *False sharing* even more disturbing
- Read-write conflicts are more sensible??

Open nesting

```
atomic { … open { s; } … }
```

Open nesting is quite like escaping, except:

- Body is itself a *transaction* (isolated from others)

Can encode if `atomic` is allowed within `escape` :

```
atomic { … escape { atomic { s; } … }}
```

Language-design issues

- Interaction with exceptions
- Interaction with native-code
- Closed nesting (flatten vs. partial rollback)
- Open nesting (back-door or proper abstraction?)
- Multithreaded transactions
- The `orelse` combinator
- `atomic` as a first-class function

Multithreaded Transactions

- Most implementations assume single-threaded transactions
  - Thread-creation a dynamic error within transaction
  - But isolation and parallelism are orthogonal
  - And Amdahl’s Law will strike with manycore

- So what does thread-spawn within a transaction mean?
- 2 useful answers (programmer picks for each spawn):
  1. Spawn delayed until/unless transaction commits
  2. Transaction commits only after spawnee completes
     - Now want “real” nested transactions…

Example

Pseudocode (to avoid spawn boilerplate)

```
atomic {
        Queue   q    = newQueue();
        boolean done = false;

  while(moreWork)   ||   while(true) {
    q.enqueue(…);   ||    atomic {
  atomic {          ||     if(done)
    done=true;      ||       return;
  }                 ||    }
                    ||    x=q.dequeue();
                    ||    … process x …
                    ||   }
}
```

Note: **enqueue** and **dequeue** also use nested **atomic**

Language-design issues

- Interaction with exceptions
- Interaction with native-code
- Closed nesting (flatten vs. partial rollback)
- Open nesting (back-door or proper abstraction?)
- Multithreaded transactions
- The **`orelse`** combinator
- **`atomic`** as a first-class function

Why orelse?

- Sequential composition of transactions is easy:

```
void f() { atomic { … } }
void g() { atomic { … } }
void h() { atomic { f(); g(); } }
```

- But what about alternate composition
- Example: “get something from either of two buffers, retrying only if both are empty”

```
void get(Queue buf){
  atomic{ if(empty(buf)) retry; …}
}
void get2(Queue buf1, Queue buf2) {???}
```

orelse

- Only “solution” so far is to break abstraction
  - The greatest programming sin
- Better: `orelse`
  - Semantics: On retry, try alternative, if it also retries, the whole thing retries
  - Allow ≥ 0 “orelse branches” on atomic

```
void get2(Queue buf1, Queue buf2){
  atomic { get(buf1); }
  orelse { get(buf2); }
}
```

One cool ML thing

As usual, languages with convenient higher-order functions avoid syntactic extensions

To the front-end, `atomic` is just a first-class function

```
Thread.atomic : (unit -> `a) -> `a
```

So yes, you can pass it around (useful?)

Like every other function, it has two run-time versions

- For outside of a transaction (start one)
- For inside of a transaction (just call the function)
  - Flattened nesting
- But this is just an implementation detail

Language-design issues

- Interaction with exceptions
- Interaction with native-code
- Closed nesting (flatten vs. partial rollback)
- Open nesting (back-door or proper abstraction?)
- Multithreaded transactions
- The **`orelse`** combinator
- **`atomic`** as a first-class function

Overall lesson: Language design is essential and nontrivial (key role for PL to play)

My tentative plan

1. Basics: language constructs, implementation intuition (Tim next week)

2. Motivation: the TM/GC Analogy

3. Strong vs. weak atomicity
   - And optimizations relevant to strong

4. Formal semantics for transactions / proof results
   - Including formal-semantics tutorial

5. Brief mention: memory-models, continuations

Advantages

So `atomic` "sure feels better than locks"

But the crisp reasons I've seen are all (great) examples

- Account transfer from Flanagan et al.
    - See also Java's `StringBuffer append`
- Double-ended queue from Herlihy

Code evolution

```
void deposit(…)  { synchronized(this) { … }}
void withdraw(…) { synchronized(this) { … }}
int  balance(…)  { synchronized(this) { … }}
```

Code evolution

```
void deposit(…)  { synchronized(this) { … }}
void withdraw(…) { synchronized(this) { … }}
int  balance(…)  { synchronized(this) { … }}
void transfer(Acct from, int amt) {

  if(from.balance()>=amt && amt < maxXfer) {
    from.withdraw(amt);
    this.deposit(amt);
  }

}
```

Code evolution

```
void deposit(…)  { synchronized(this) { … }}
void withdraw(…) { synchronized(this) { … }}
int  balance(…)  { synchronized(this) { … }}

void transfer(Acct from, int amt) {
  synchronized(this) {
   //race
   if(from.balance()>=amt && amt < maxXfer) {
     from.withdraw(amt);
     this.deposit(amt);
   }
  }
}
```

Code evolution

```
void deposit(…)  { synchronized(this) { … }}
void withdraw(…) { synchronized(this) { … }}
int  balance(…)  { synchronized(this) { … }}

void transfer(Acct from, int amt) {
  synchronized(this) {
  synchronized(from) { //deadlock (still)
   if(from.balance()>=amt && amt < maxXfer) {
     from.withdraw(amt);
     this.deposit(amt);
   }
  }}
}
```

Code evolution

```
void deposit(…)  { atomic { … }}
void withdraw(…) { atomic { … }}
int  balance(…)  { atomic { … }}
```

Code evolution

```
void deposit(…)  { atomic { … }}
void withdraw(…) { atomic { … }}
int  balance(…)  { atomic { … }}

void transfer(Acct from, int amt) {

   //race
   if(from.balance()>=amt && amt < maxXfer) {
      from.withdraw(amt);
      this.deposit(amt);
   }

}
```

Code evolution

```
void deposit(…)  { atomic { … }}
void withdraw(…) { atomic { … }}
int  balance(…)  { atomic { … }}

void transfer(Acct from, int amt) {
  atomic {
    //correct and parallelism-preserving!
   if(from.balance()>=amt && amt < maxXfer) {
     from.withdraw(amt);
     this.deposit(amt);
   }
  }
}
```

It really happens

Example [JDK1.4, version 1.70, Flanagan/Qadeer PLDI2003]

```
synchronized append(StringBuffer sb) {
 int len = sb.length();
 if(this.count + len > this.value.length)
   this.expand(…);
 sb.getChars(0,len,this.value,this.count);
 …
}
// length and getChars are synchronized
```

Documentation addition for Java 1.5.0:

“This method synchronizes on **`this`** (the destination) object but does not synchronize on the source (**`sb`**).”

Advantages

So **`atomic`** “sure feels better than locks”

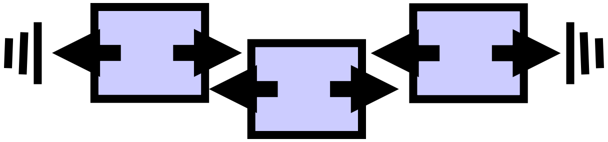
But the crisp reasons I’ve seen are all (great) examples

- Account transfer from Flanagan et al
  - See also Java’s **`StringBuffer append`**
- Double-ended queue from Herlihy

Double-ended queue

Operations

```
void enqueue_left(Object)
void enqueue_right(Object)
obj  dequeue_left()
obj  dequeue_right()
```



Correctness

- Behave like a queue, even when ≤ 2 elements
- Dequeuers wait if necessary, but can’t “get lost”

Parallelism

- Access both ends in parallel, except when ≤ 1 elements (because ends overlap)

Good luck with that…

- One lock?
  - No parallelism

- Locks at each end?
  - Deadlock potential
  - Gets very complicated, etc.

- Waking blocked dequeuers?
  - Harder than it looks

Actual Solution

- A clean solution to this apparent "homework problem" would be a publishable result?
  - In fact it was: [Michael & Scott, PODC 96]

- So locks and condition variables are not a "natural methodology" for this problem

- Implementation with transactions is trivial
  - Wrap 4 operations written sequentially in **`atomic`**
    - With **`retry`** for dequeuing from empty queue
  - Correct and parallel

Advantages

So `atomic` “sure feels better than locks”

But the crisp reasons I’ve seen are all (great) examples

- Account transfer from Flanagan et al
  - See also Java’s **`StringBuffer append`**
- Double-ended queue from Herlihy

- *… probably many more …*

But can we generalize

But what is the essence of the benefit?

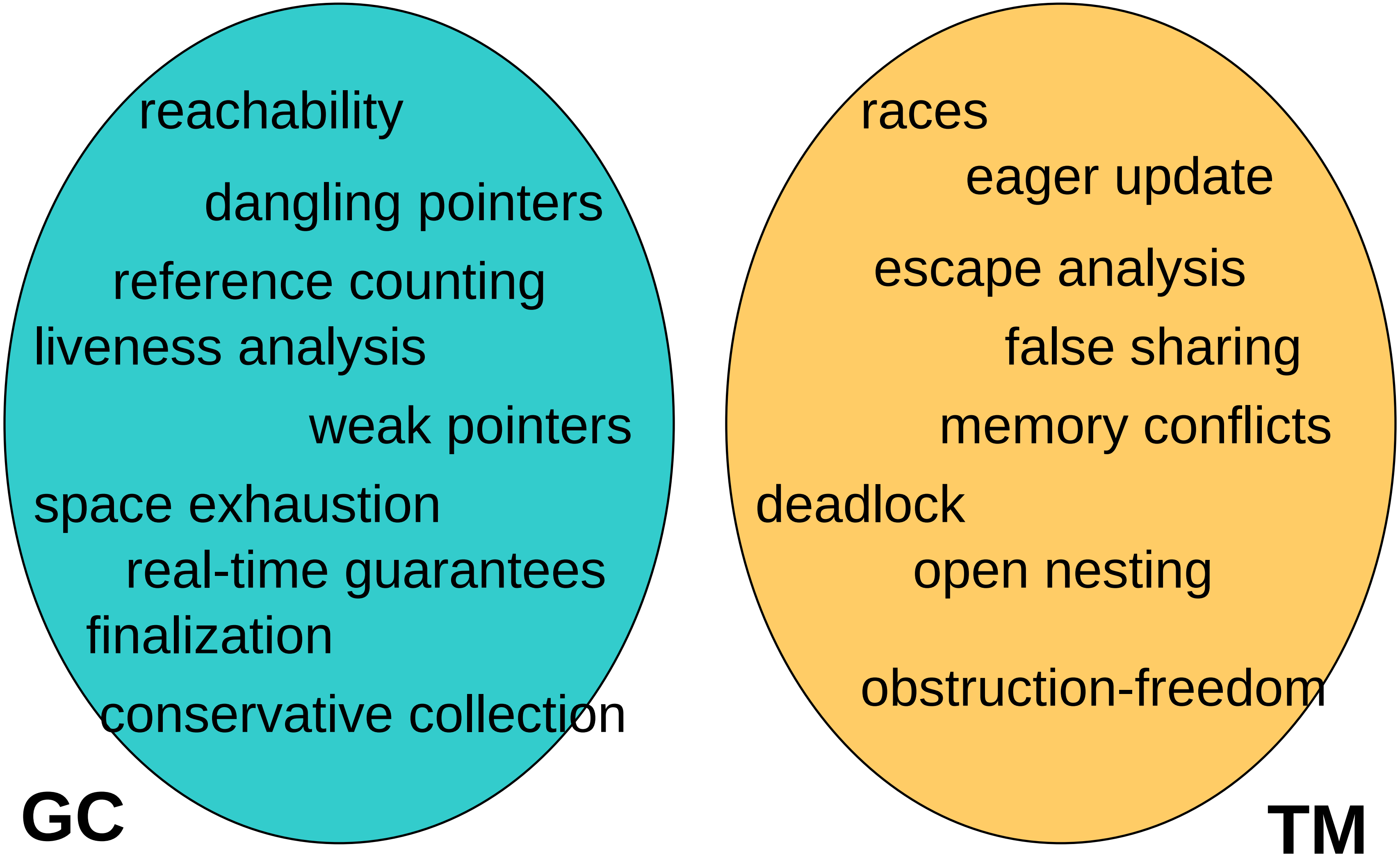
Transactional Memory (TM) is to
shared-memory concurrency
as
Garbage Collection (GC) is to
memory management

Explaining the analogy

"TM is to shared-memory concurrency as
GC is to memory management"

- Why an analogy helps
- Brief overview of GC
- The core technical analogy (but read the essay)
  - And why concurrency is still harder
- Provocative questions based on the analogy

Two bags of concepts



reachability

dangling pointers

reference counting

liveness analysis

weak pointers

space exhaustion

real-time guarantees

finalization

conservative collection

GC

races

eager update

escape analysis

false sharing

memory conflicts

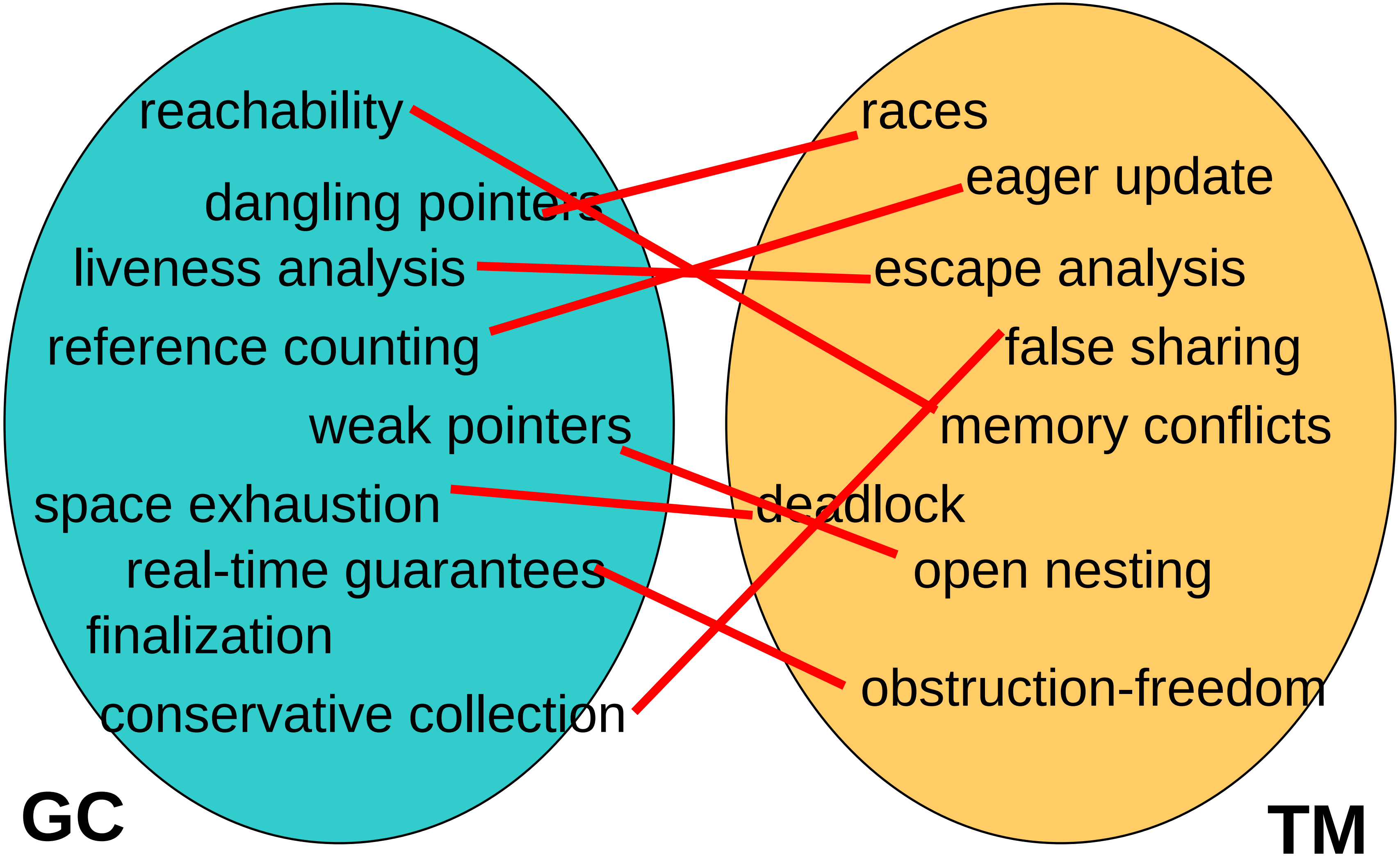
deadlock

open nesting

obstruction-freedom

TM

Interbag connections



reachability

dangling pointers

liveness analysis

reference counting

weak pointers

space exhaustion

real-time guarantees

finalization

conservative collection

GC

races

eager update

escape analysis

false sharing

memory conflicts

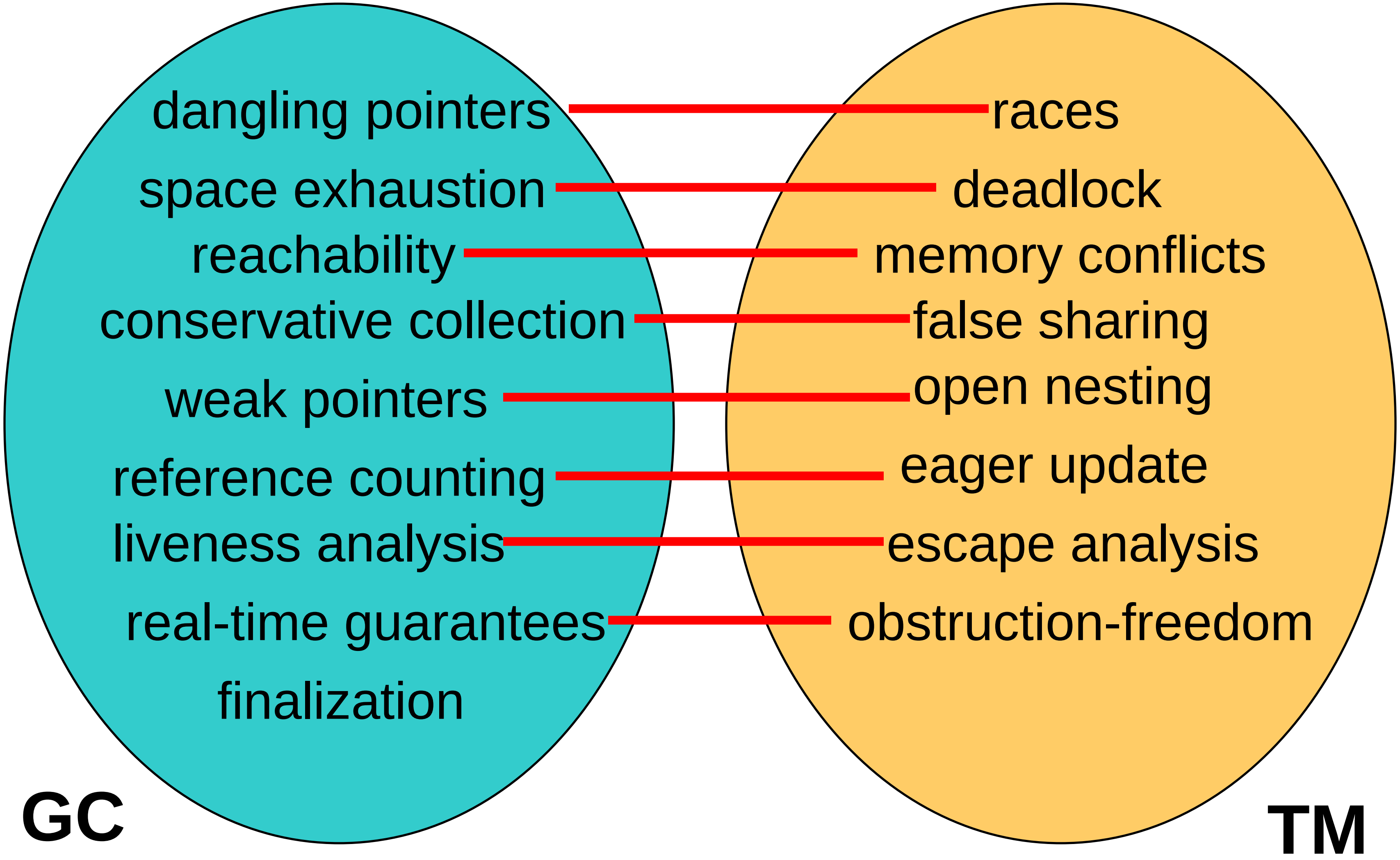
deadlock

open nesting

obstruction-freedom

TM

Analogies help organize



GC	TM
dangling pointers	races
space exhaustion	deadlock
reachability	memory conflicts
conservative collection	false sharing
weak pointers	open nesting
reference counting	eager update
liveness analysis	escape analysis
real-time guarantees	obstruction-freedom
finalization	

So the goals are…

- Leverage the design trade-offs of GC to guide TM
  - And vice-versa?

- Identify open research

- Motivate TM
  - TM improves concurrency as GC improves memory
  - GC is a huge help *despite its imperfections*
  - So TM is a huge help *despite its imperfections*

Explaining the analogy

“TM is to shared-memory concurrency as
GC is to memory management”

- Why an analogy helps
- Brief overview of GC
- The core technical analogy (but read the essay)
  - And why concurrency is still harder
- Provocative questions based on the analogy

Memory management

Allocate objects in the heap

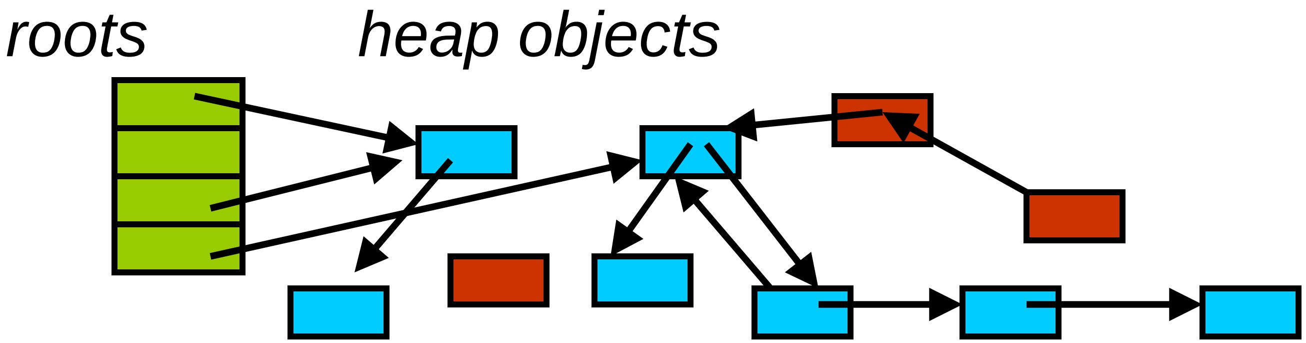
Deallocate objects to reuse heap space

- If too soon, dangling-pointer dereferences
- If too late, poor performance / space exhaustion

GC Basics

Automate deallocation via *reachability* approximation

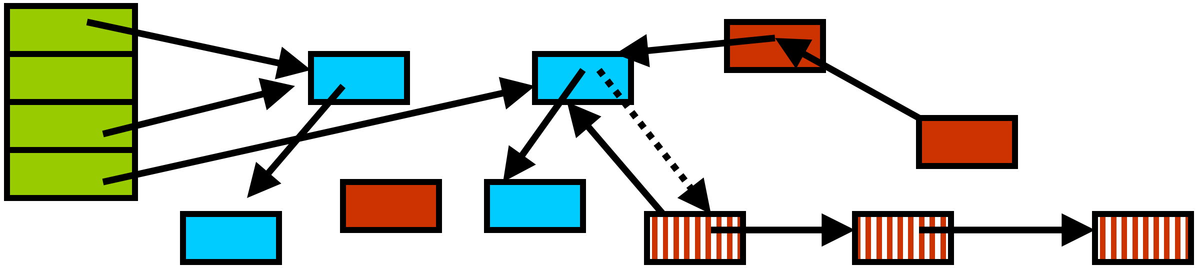
– Approximation can be terrible in theory



- Reachability via *tracing* or *reference-counting*
  – Duals [Bacon et al OOPSLA04]

- Lots of bit-level tricks for simple ideas
  – And high-level ideas like a *nursery* for new objects

A few GC issues



- *Weak pointers*
  - Let programmers overcome reachability approx.
- *Accurate* vs. *conservative*
  - Conservative can be unusable (only) in theory
- *Real-time* guarantees for responsiveness

GC Bottom-line

Established technology with widely accepted benefits

- Even though it can perform terribly in theory
- Even though you can't always ignore how GC works (at a high-level)
- Even though an active research area after 40+ years

Explaining the analogy

“TM is to shared-memory concurrency as
GC is to memory management”

- Why an analogy helps
- Brief separate overview of GC and TM
- The core technical analogy (but read the essay)
  - And why concurrency is still harder
- Provocative questions based on the analogy

The problem, part 1

Why ~~memory management~~ concurrent programming is hard:

Balance correctness (avoid ~~dangling pointers~~ race conditions)

And performance (~~space waste~~ loss of parallelism or ~~exhaustion~~ deadlock)

Manual approaches require whole-program protocols

Example: Manual ~~reference count~~ lock for each object

- Must avoid ~~garbage~~ lock acquisition cycles

The problem, part 2

synchronization

Manual ~~memory-management~~ is non-modular:

- Caller and callee must know what each other access or ~~deallocate~~ to ensure right ~~memory is live~~

  release locks are held

- A small change can require wide-scale code changes
  - Correctness requires knowing what data ~~subsequent~~ computation will access

    concurrent

The solution

Move whole-program protocol to language implementation

- One-size-fits-most implemented by experts
  - Usually combination of compiler and run-time

- TM ~~GC~~ system uses subtle invariants, e.g.:
  - Object header-word bits
  - No thread-shared ~~unknown mature~~ pointers to thread-local ~~nursery~~ objects

- In theory, optimistic concurrency ~~object relocation~~ can improve performance by increasing ~~spatial locality~~ parallelism
  - In practice, some performance loss worth convenience

Two basic approaches

- ~~Tracing~~ update-on-commit: assume all data is ~~live~~ conflict-free, detect ~~garbage~~ conflicts later
- ~~Reference counting~~ update-in-place: can detect ~~garbage~~ conflicts immediately
  - Often defer some ~~counting~~ conflict-detection to trade immediacy for performance (e.g., ~~trace the stack~~ optimistic reads)

So far…

	memory management	**concurrency**
correctness	dangling pointers	races
performance	space exhaustion	deadlock
automation	garbage collection	transactional memory
new objects	nursery data	thread-local data
eager approach	reference-counting	update-in-place
lazy approach	tracing	update-on-commit

Incomplete solution

GC a bad idea when “reachable” is a bad approximation of “cannot-be-deallocated”

Weak pointers overcome this fundamental limitation

- Best used by experts for well-recognized idioms (e.g., software caches)

In extreme, programmers can encode

manual memory management on top of GC

- Destroys most of GC’s advantages…

Circumventing GC

```
class Allocator {
  private SomeObjectType[] buf   = …;
  private boolean[]        avail = …;

  Allocator() {
    // initialize arrays
  }
  void malloc() {
    // find available index
  }
  void free(SomeObjectType o) {
    // set corresponding index available
  }
}
```

Incomplete solution

~~GC~~ TM a bad idea when "~~reachable~~" memory conflict is a bad approximation of "cannot-be-~~deallocated~~" run-in-parallel

~~Weak pointers~~ Open nested txns overcome this fundamental limitation

- Best used by experts for well-recognized idioms (e.g., ~~software caches~~ unique id generation)

In extreme, programmers can encode ~~manual memory management~~ locking on top of ~~GC~~ TM

- Destroys most of ~~GC~~ TM's advantages…

Circumventing ~~GC~~ TM

```
class SpinLock {
  private boolean b = false;

  void acquire() {
    while(true)
      atomic {
        if(b) continue;
        b = true;
        return;
       }
  }
  void release() {
    atomic { b = false; }
  }
}
```

Programmer control

For performance and simplicity, ~~GC~~ (some) TM treats entire objects as ~~reachable~~ accessed, which can lead to ~~more space~~ less parallelism

~~Space~~ Parallelism-conscious programmers can reorganize data accordingly

But with ~~conservative collection~~ coarser granularity (e.g., cache lines), programmers cannot completely control what appears ~~reachable~~ conflicting

- – Arbitrarily bad in theory

So far…

	memory management	**concurrency**
correctness	dangling pointers	races
performance	space exhaustion	deadlock
automation	garbage collection	transactional memory
new objects	nursery data	thread-local data
eager approach	reference-counting	update-in-place
lazy approach	tracing	update-on-commit
key approximation	reachability	memory conflicts
manual circumvention	weak pointers	open nesting
uncontrollable approx.	conservative collection	false memory conflicts

More

- I/O: output after input ~~of pointers~~ in transactions can cause incorrect behavior due to ~~dangling pointers~~ irreversible actions
- ~~Real-time guarantees~~ Obstruction-freedom doable but costly
- Static analysis can avoid overhead
  - Example: ~~liveness~~ escape analysis for fewer ~~root locations~~ potential conflicts
  - Example: remove write-barriers on ~~nursery~~ thread-local data

Too much coincidence!

	memory management	concurrency
correctness	dangling pointers	races
performance	space exhaustion	deadlock
automation	garbage collection	transactional memory
new objects	nursery data	thread-local data
eager approach	reference-counting	update-in-place
lazy approach	tracing	update-on-commit
key approximation	reachability	memory conflicts
manual circumvention	weak pointers	open nesting
uncontrollable approx.	conservative collection	false memory conflicts
more…	I/O of pointers	I/O in transactions
	real-time	obstruction-free
	liveness analysis	escape analysis
	…	…

Explaining the analogy

*“TM is to shared-memory concurrency as
GC is to memory management”*

- Why an analogy helps
- Brief separate overview of GC and TM
- The core technical analogy (but read the essay)
  - And why concurrency is still harder
- Provocative questions based on the analogy

Concurrency is hard!

I ***never said*** *the analogy means*
TM parallel programming is as easy as
GC sequential programming

By moving low-level protocols to the language run-time, TM lets programmers just declare where critical sections should be

But that is still very hard and – by definition – unnecessary in sequential programming

Huge step forward ≠ panacea

Non-technical conjectures

I can defend the technical analogy on solid ground

Then push things (perhaps) too far …

1. Many used to think GC was too slow without hardware
2. Many used to think GC was "about to take over" (decades before it did)
3. Many used to think we needed a "back door" for when GC was too approximate

Motivating you

Push the analogy further or discredit it

- Generational GC?
- Contention management?
- Inspire new language design and implementation

Teach programming with TM as we teach programming with GC

Find other useful analogies

My tentative plan

1. Basics: language constructs, implementation intuition (Tim next week)

2. Motivation: the TM/GC Analogy

3. Strong vs. weak atomicity
   - And optimizations relevant to strong

4. Formal semantics for transactions / proof results
   - Including formal-semantics tutorial

5. Brief mention: memory-models, continuations

The Naïve View

```
atomic { s }
```

Run ***s*** as though no other computation is interleaved?

May not be “true enough”:

- Races with nontransactional code can break isolation
  - *Even when similar locking code is correct*
- Restrictions on what ***s*** can do (e.g., spawn a thread)
  - *Even when similar locking code is correct*

"Weak" isolation

```
initially y=0
```

```
atomic {
  y = 1;
  x = 3;
  y = x;
}
```

```
x = 2;
print(y); //1? 2? 666?
```

Widespread misconception:

"Weak" isolation violates the "all-at-once" property only if corresponding lock code has a race

(May still be a bad thing, but smart people disagree.)

A second example

We’ll go through many examples like this:

```
initially x=0, y=0, b=false
```

```
atomic {
  if(b)
    ++x;
  else
    ++y;
}
```

```
r = x; //race
s = y; //race
assert(r+s<2);
```

```
atomic {
  b=true;
}
```

Assertion can’t fail under the naïve view (or with locks??)

Assertion can fail under some but not all STMs

– Must programmers know about retry?

The need for semantics

- A high-level language ***must*** define whether our example’s assertion can fail

- Such behavior was unrecognized 3 years ago
  - A rigorous semantic definition helps us “think of everything” (no more surprises)
  - Good news: We can define sufficient conditions under which naïve view is correct and prove it

- Why not just say, “if you have a data race, the program can do anything”?
  - A couple reasons…

The “do anything” non-starter

In safe languages, it must be possible to write secure code, *even if other (untrusted) code is broken*

```
class Secure {
  private String pwd = “topSecret”;
  private void withdrawBillions(){…}
  public check(String s){
    if(s.equals(pwd))
      withdrawBillions();
  }
}
```

Unlike C/C++, a buffer overflow, race condition, or misuse of atomic in another class can’t corrupt **`pwd`**

The “what’s a race” problem

“Banning race conditions” requires defining them

Does this have a race?

```
initially x=0, y=0, z=0
```

```
atomic {
  if(x<y)
    ++z;
}
```

```
atomic {
  ++x;
  ++y;
}
```

```
r = z; //race?
assert(r==0);
```

Dead code under “naïve view” isn’t dead with many STMs

Adapted from [Abadi et al POPL2008]

So…

Hopefully you're convinced high-level language semantics is needed for transactions to succeed

First focus on various notions of *isolation*

- A taxonomy of ways weak isolation can surprise you
- Ways to avoid surprises
  - Strong isolation (enough said?)
  - Restrictive type systems

Then formal semantics for high-level definitions & correctness proofs

Notions of isolation

- Strong-isolation: A transaction executes as though no other *computation* is interleaved
- Weak-isolation?
  - Single-lock ("weak-sla"): A transaction executes as though no other *transaction* is interleaved
  - Single-lock + abort ("weak undo"): Like weak-sla, but a transaction can retry, undoing changes
  - Single-lock + lazy update ("weak on-commit"): Like weak-sla, but buffer updates until commit
  - Real contention: Like "weak undo" or "weak on-commit", but multiple transactions can run at once
  - Catch-fire: Anything can happen if there's a race

Strong-Isolation

Strong-isolation is clearly the simplest semantically, and we've been working on getting scalable performance

Arguments against strong-isolation:

1. Reads/writes outside transactions need expensive extra code (including synchronization on writes)
   - *Optimize common cases, e.g., thread-local data*
2. Reads/writes outside transactions need extra code, so that interferes with precompiled binaries
   - *A nonissue for managed languages (bytecodes)*
3. Blesses subtle, racy code that is bad style
   - *Every language blesses bad-style code*

Taxonomy of Surprises

Now let’s use examples to consider:

- strong vs. weak-sla (less surprising; same as locks)
- strong vs. weak undo
- strong vs. weak on-commit
- strong vs. real contention (undo or on-commit)

Then:

- Static partition (a.k.a. segregation) to avoid surprises
- Formal semantics for proving the partition correct

strong vs. weak-sla

Since weak-sla is *like* a global lock, the “surprises” are the expected data-race issues

Dirty read:

non-transactional read between transactional writes

```
initially x=0
```

```
atomic {
  x=1;
  x=2;
}
```

```
r = x;
```

can **r==1**?

strong vs. weak-sla

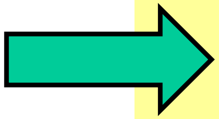
Since weak-sla is *like* a global lock, the “surprises” are the expected data-race issues

Non-repeatable read:

non-transactional write between transactional reads

```
initially x=0
```

```
atomic {
  r1=x;
  r2=x;
}
```



```
x=1;
```

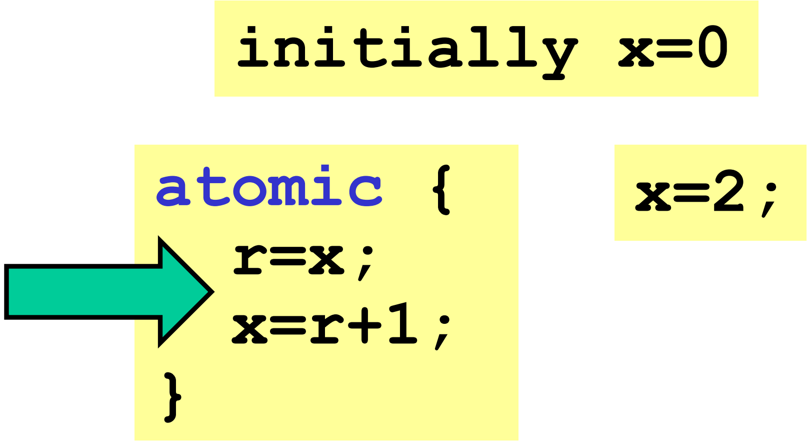
can **r1!=r2**?

strong vs. weak-sla

Since weak-sla is *like* a global lock, the “surprises” are the expected data-race issues

Lost update:

non-transactional write after transactional read



```
initially x=0
```

```
atomic {
  r=x;
  x=r+1;
}
```

```
x=2;
```

can **x==1**?

Taxonomy

- strong vs. weak-sla (not surprising)
  - *dirty read, non-repeatable read, lost update*
- strong vs. weak undo
  - *weak, plus…*
- strong vs. weak on-commit
- strong vs. real contention

strong vs. weak undo

With eager-update and undo, races can interact with speculative (aborted-later) transactions

Speculative dirty read:

non-transactional read of speculated write

```
initially x=0, y=0
```

```
atomic {
  if(y==0){
   x=1;
   retry;
  }
}
```

```
if(x==1)
  y=1;
```

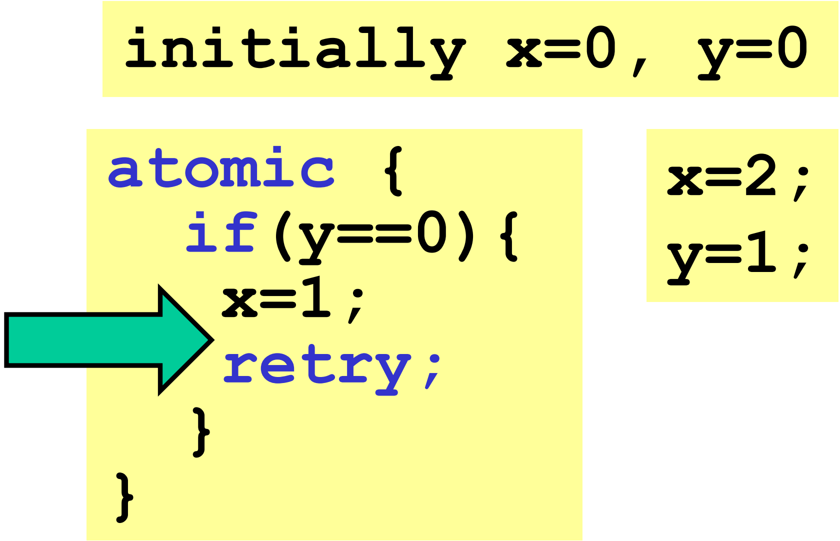
an early example was also a speculative dirty read

can `y==1`?

strong vs. weak undo

With eager-update and undo, races can interact with speculative (aborted-later) transactions

Speculative lost update: non-transactional write between transaction read and speculated write



```
initially x=0, y=0
```

```
atomic {
  if(y==0){
   x=1;
   retry;
  }
}
```

```
x=2;
y=1;
```

can **x==0**?

strong vs. weak undo

With eager-update and undo, races can interact with speculative (aborted-later) transactions

Granular lost update:

lost update via different fields of an object

```
initially x.g=0, y=0
```

```
atomic {
  if(y==0){
   x.f=1;
   retry;
  }
}
```

```
x.g=2;
y=1;
```

can **`x.g==0`**?

Taxonomy

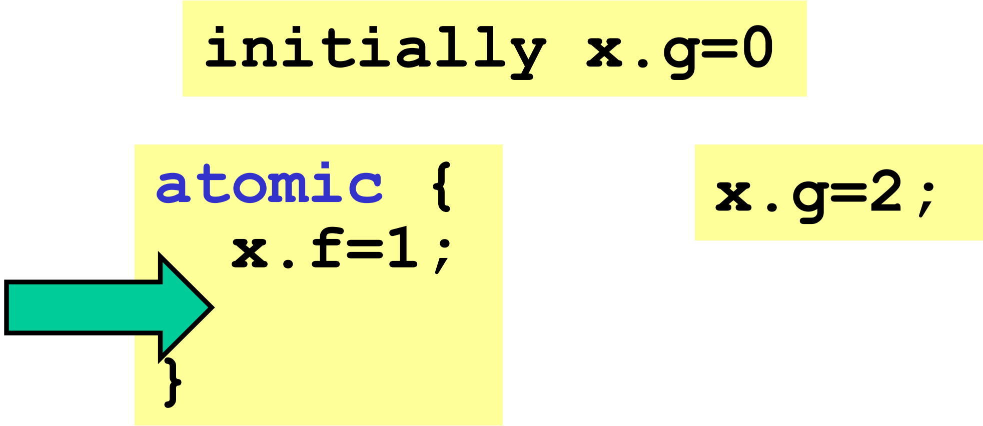
- strong vs. weak-sla (not surprising)
  - *dirty read, non-repeatable read, lost update*
- strong vs. weak undo
  - *weak, plus speculative dirty reads & lost updates, granular lost updates*
- strong vs. weak on-commit
- strong vs. real contention

strong vs. weak on-commit

With lazy-update and undo, speculation and dirty-read problems go away, but problems remain…

Granular lost update:

lost update via different fields of an object



```
initially x.g=0
```

```
atomic {
  x.f=1;

}
```

```
x.g=2;
```

can `x.g==0`?

strong vs. weak on-commit

With lazy-update and undo, speculation and dirty-read problems go away, but problems remain…

Reordering: transactional writes exposed in wrong order

```
initially x=null, y.f=0
```

```
atomic {
  y.f=1;
  x=y;

}
```

```
r=-1;
if(x!=null)
  r=x.f;
```

can `r==0`?

Technical point: x should be volatile (need reads ordered)

Taxonomy

- strong vs. weak-sla (not surprising)
  - *dirty read, non-repeatable read, lost update*
- strong vs. weak undo
  - *weak, plus speculative dirty reads & lost updates, granular lost updates*
- strong vs. weak on-commit
  - *weak (minus dirty read), plus granular lost updates, reordered writes*
- strong vs. real contention (with undo or on-commit)

strong vs. real contention

Some issues require multiple transactions running at once

Publication idiom unsound

```
initially ready=false, x=0, val=-1
```

```
atomic {
 tmp=x;
 if(ready)
   val=tmp;
}
```

```
x=1;
atomic {
 ready=true;
}
```

can `val==0`?

Adapted from [Abadi et al POPL2008]

strong vs. real contention

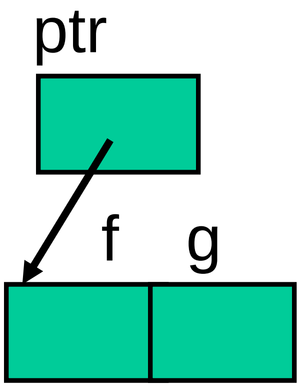
Some issues require multiple transactions running at once

Privatization idiom unsound

```
initially ptr.f = ptr.g
```

```
atomic {
  r   = ptr;
  ptr = new C();
}
assert(r.f!=r.g)
```

```
atomic {
  ++ptr.f;
  ++ptr.g;
}
```



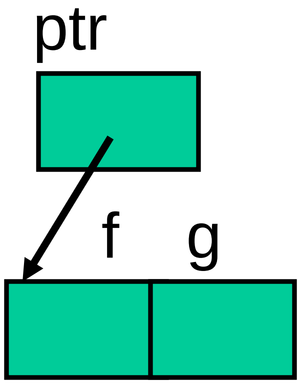
Adapted from [Rajwar/Larus] and [Hudson et al.]

More on privatization

```
initially ptr.f = ptr.g
```

```
atomic {
  r = ptr;
  ptr = new C();
}
assert(r.f!=r.g)
```

```
atomic {
  ++ptr.f;
  ++ptr.g;
}
```



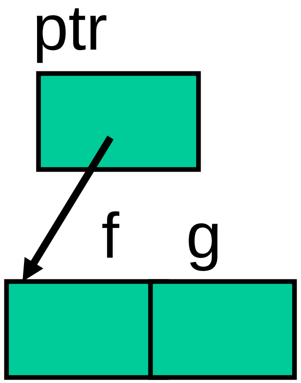
- With undo, assertion can fail after right thread does one update and before it aborts due to conflict

More on privatization

```
initially ptr.f = ptr.g
```

```
atomic {
  r = ptr;
  ptr = new C();
}
assert(r.f!=r.g)
```

```
atomic {
  ++ptr.f;
  ++ptr.g;
}
```



- With undo, assertion can fail after right thread does one update and before it aborts due to conflict
- With on-commit, assertion can fail if right thread commits first, but updates happen later
(racing with assertion)

Taxonomy

- strong vs. weak-sla (not surprising)
  - *dirty read, non-repeatable read, lost update*
- strong vs. weak undo
  - *weak, plus speculative dirty reads & lost updates, granular lost updates*
- strong vs. weak on-commit
  - *weak (minus dirty read), plus granular lost updates, and reordered writes*
- strong vs. real contention (with undo or on-commit)
  - *the above, plus publication and privatization*

“Weak” isolation in practice

“Weak” really means nontransactional code bypasses the transaction mechanism…

Imposes correctness burdens on programmers that locks do not…

… and what the burdens are depends on the details of the TM implementation

- If you got lost in some examples, imagine mainstream programmers

Does it matter?

- These were simple-as-possible examples
  - to define the issues
- If “nobody would ever write that” maybe you’re unconvinced
  - PL people know better than to use that phrase
  - Publication, privatization are common idioms
  - Issues can also arise from compiler transformations

Taxonomy of Surprises

Now let’s use examples to consider:

- strong vs. weak-sla (less surprising; same as locks)
- strong vs. weak undo
- strong vs. weak on-commit
- strong vs. real contention (undo or on-commit)

Then:

- Static partition (a.k.a. segregation) to avoid surprises
- Formal semantics for proving the partition correct

Partition

Surprises arose from the same mutable locations being used inside & outside transactions by different threads

Hopefully sufficient to forbid that

- But unnecessary and probably too restrictive
  - Bans publication and privatization
- cf. STM Haskell [PPoPP05]

For each allocated object (or word), require *one* of:

1. Never mutated
2. Only accessed by one thread
3. Only accessed inside transactions
4. Only accessed outside transactions

Static partition

Recall our “what is a race” problem:

```
initially x=0, y=0, z=0
```

```
atomic {
  if(x<y)
    ++z;
}
```

```
atomic {
  ++x;
  ++y;
}
```

```
r = z; //race?
assert(r==0);
```

So “accessed on valid control paths” is not enough

– Use a *type system* that conservatively assumes all paths are possible

Type system

Part of each variable's type is how it may be used:

1. Never mutated (not on left-hand-side)
2. Thread-local (not pointed-to from thread-shared)
3. Inside transactions (+ in-transaction methods)
4. Outside transactions

Part of each method's type is where it may be called:

1. Inside transactions (+ other in-transaction methods)
2. Outside transactions

Will formalize this idea in the remaining lectures

Example

Our example does not type-check because **`z`** has no type

```
initially x=0, y=0, z=0
```

```
atomic {
  if(x<y)
    ++z;
}
```

```
atomic {
  ++x;
  ++y;
}
```

```
r = z; //race?
assert(r==0);
```

Formalizing the type system and extending to method calls is a totally standard *type-and-effect system*

My tentative plan

1. Basics: language constructs, implementation intuition (Tim next week)

2. Motivation: the TM/GC Analogy

3. Strong vs. weak atomicity
   - And optimizations relevant to strong

4. Formal semantics for transactions / proof results
   - Including formal-semantics tutorial

5. Brief mention: memory-models, continuations

Strong performance problem

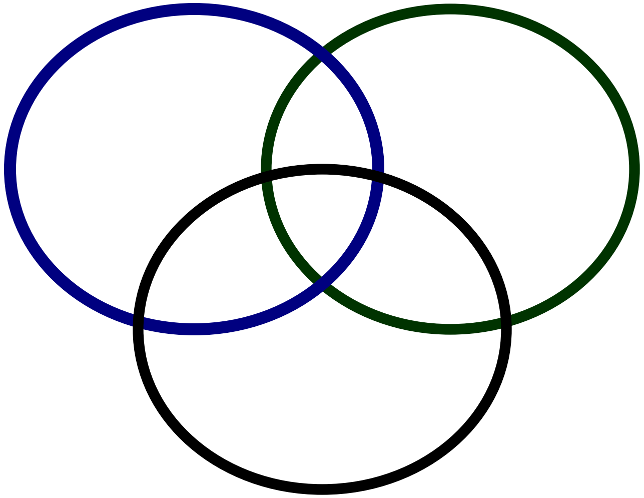
The overhead of transactions:

	not in atomic	in atomic
read	none iff weak	some
write	none iff weak	some

Expect:

- Most code runs outside transactions
- Aborts are rare

Optimizing away strong's cost



Thread local

Not accessed in transaction

Immutable

New: static analysis for not-accessed-in-transaction …

Not-accessed-in-transaction

Revisit overhead of not-in-atomic for strong isolation, given information about how data is used in atomic

	not in atomic			in atomic
	no atomic access	no atomic write	atomic write	
read	none	none	some	some
write	none	some	some	some

Yet another client of pointer-analysis

Analysis details

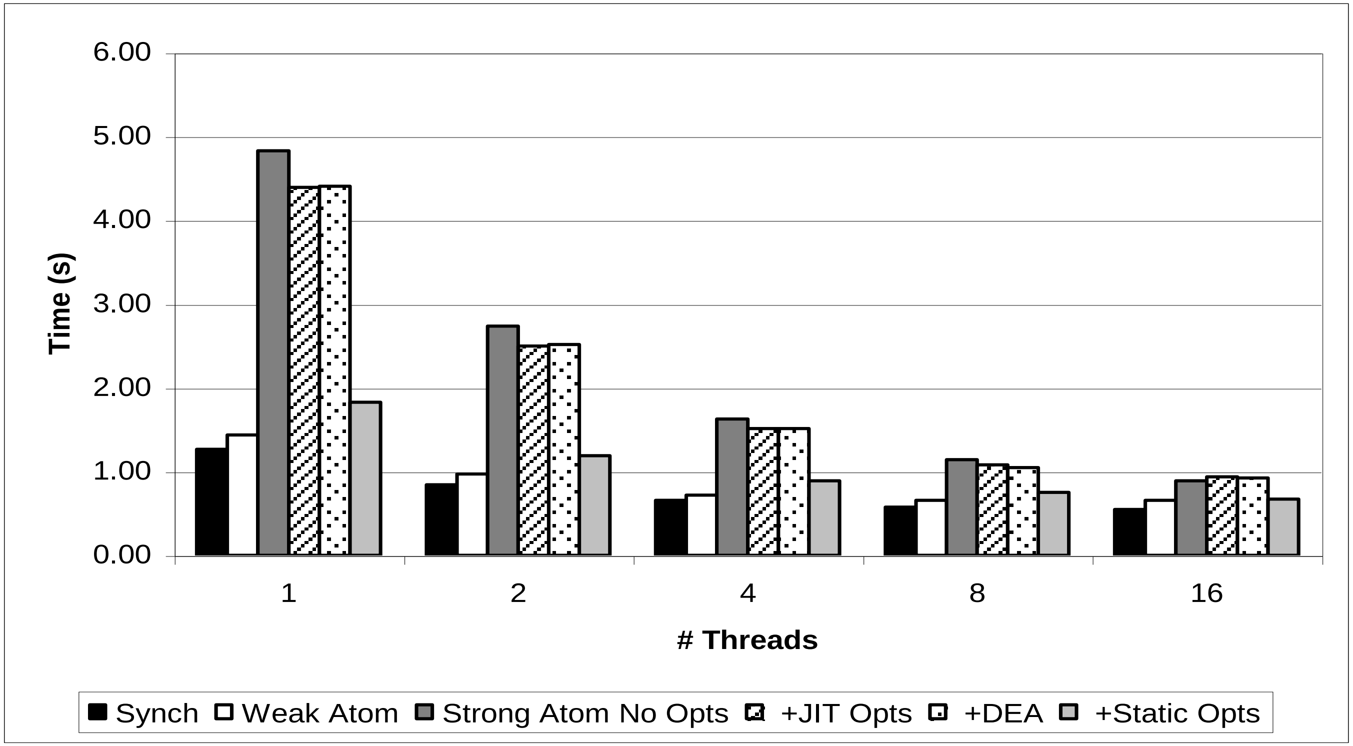
- Whole-program, context-insensitive, flow-insensitive
  - Scalable, but needs whole program
- Given pointer information, just two more passes
  1. How is an “abstract object” accessed transactionally?
  2. What “abstract objects” might a non-transactional access use?
- Can be done before method duplication
  - Keep lazy code generation without losing precision

Collaborative effort

- UW: static analysis using pointer analysis
  - Via Paddle/Soot from McGill
- Intel PSL: high-performance STM
  - Via compiler and run-time

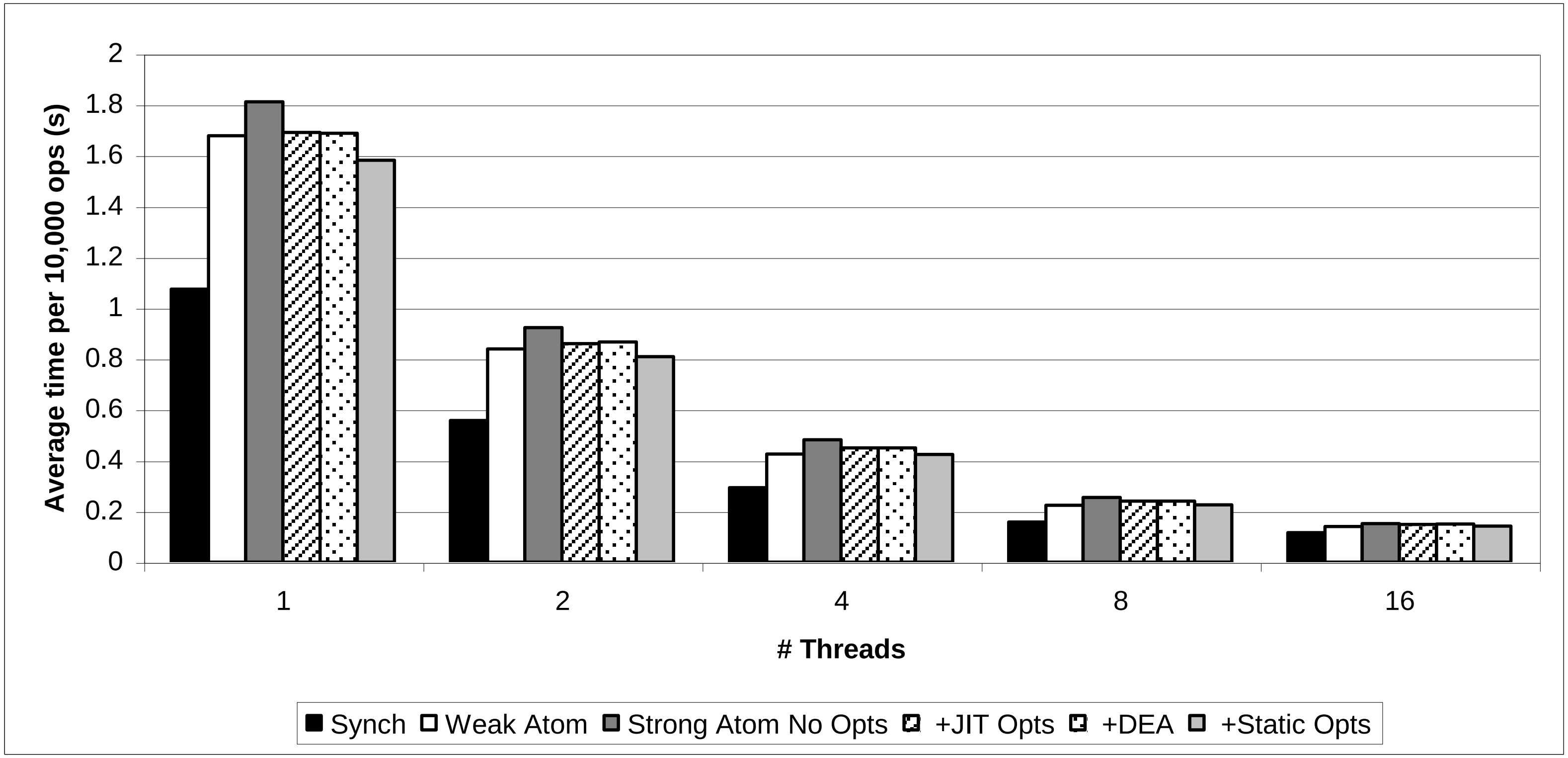
Static analysis annotates bytecodes, so the compiler back-end knows what it can omit

Benchmarks



Tsp

Benchmarks



JBB

My tentative plan

1. Basics: language constructs, implementation intuition (Tim next week)

2. Motivation: the TM/GC Analogy

3. Strong vs. weak atomicity
   - And optimizations relevant to strong

4. Formal semantics for transactions / proof results
   - Including formal-semantics tutorial

5. Brief mention: memory-models, continuations

Outline

1. Lambda-calculus / operational semantics tutorial

2. Add threads and mutable shared-memory

3. *Add transactions; study weak vs. strong isolation*

4. Simple type system

5. *Type (and effect system) for strong = weak*
   - *And proof sketch*

Lambda-calculus in about an hour

- To decide “what concurrency means” we must start somewhere
- One popular *sequential* place: a lambda-calculus
- Can define:
  - *Syntax* (abstract)
  - *Semantics* (operational, small-step, call-by-value)
  - *Types* (filter out “bad” programs)
    - Will add *effects* later (have many uses)

Syntax

Syntax of an *untyped lambda-calculus*

Expressions: `e ::= x | λx. e | e e | c | e + e`
Constants: `c ::= … | -1 | 0 | 1 | …`
Variables: `x ::= … | x1 | x′ | y | …`
Values: `v ::= λx. e | c`

Defines a set of trees (ASTs)

Conventions for writing these trees as strings:

- `λx. e1 e2` is `λx. (e1 e2)`, not `(λx. e1) e2`
- `e1 e2 e3` is `(e1 e2) e3`, not `e1 (e2 e3)`
- Use parentheses to disambiguate or clarify

Semantics

- One computation step rewrites the program to something “closer to the answer”

$$e \rightarrow e'$$

- Inference rules describe what steps are allowed

$$\frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ e2} \qquad \frac{e2 \rightarrow e2'}{v\ e2 \rightarrow v\ e2'} \qquad \frac{}{(\lambda x.e)\ v \rightarrow e\{v/x\}}$$

$$\frac{e1 \rightarrow e1'}{e1+e2 \rightarrow e1'+e2} \qquad \frac{e2 \rightarrow e2'}{v+e2 \rightarrow v+e2'} \qquad \frac{\text{“}c1+c2=c3\text{”}}{c1+c2 \rightarrow c3}$$

Notes

- These are rule schemas
  - Instantiate by replacing metavariables consistently
- A derivation tree justifies a step
  - A proof: "read from leaves to root"
  - An interpreter: "read from root to leaves"
- Proper definition of substitution requires care
- Program evaluation is then a sequence of steps

  `e0` → **`e1`** → **`e2`** → …

- Evaluation can "stop" with a value (e.g., **`17`**) or a "stuck state" (e.g., **`17`** **λ*x*. *x***)

More notes

- I chose left-to-right call-by-value
  - Easy to change by changing/adding rules
- I chose to keep evaluation-sequence deterministic
  - Easy to change
- I chose small-step operational
  - Could spend a year on other approaches
- This language is Turing-complete
  - Even without constants and addition
  - Infinite state-sequences exist

Adding pairs

$$\mathtt{e} \;::=\; \ldots \mid (\mathtt{e},\mathtt{e}) \mid \mathtt{e}.1 \mid \mathtt{e}.2$$
$$\mathtt{v} \;::=\; \ldots \mid (\mathtt{v},\mathtt{v})$$

$$\frac{\mathtt{e1} \rightarrow \mathtt{e1'}}{(\mathtt{e1},\mathtt{e2}) \rightarrow (\mathtt{e1'},\mathtt{e2})} \qquad \frac{\mathtt{e2} \rightarrow \mathtt{e2'}}{(\mathtt{v},\mathtt{e2}) \rightarrow (\mathtt{v},\mathtt{e2'})}$$

$$\frac{\mathtt{e} \rightarrow \mathtt{e'}}{\mathtt{e}.1 \rightarrow \mathtt{e'}.1} \qquad \frac{\mathtt{e} \rightarrow \mathtt{e'}}{\mathtt{e}.2 \rightarrow \mathtt{e'}.2}$$

$$\frac{}{(\mathtt{v1},\mathtt{v2}).1 \rightarrow \mathtt{v1}} \qquad \frac{}{(\mathtt{v1},\mathtt{v2}).2 \rightarrow \mathtt{v2}}$$

Outline

1. Lambda-calculus / operational semantics tutorial
2. Add threads and mutable shared-memory
3. *Add transactions; study weak vs. strong isolation*
4. Simple type system
5. *Type (and effect system) for strong = weak*
   - *And proof sketch*

Adding concurrency

- Change our syntax/semantics so:
  - A program-state is *n* threads (top-level expressions)
  - Any one might "run next"
  - Expressions can fork (a.k.a. spawn) new threads

Expressions: e `::=` … `|` **`spawn`** e

States: T `::=` . `|` $e;T$

Exp options: o `::=` **`None`** `|` **`Some`** e

Change $e \rightarrow e'$ to $e \rightarrow e',o$

Add $T \rightarrow T'$

Semantics

$$\frac{e1 \rightarrow e1', o}{e1\ e2 \rightarrow e1'\ e2, o} \qquad \frac{e2 \rightarrow e2', o}{v\ e2 \rightarrow v\ e2', o} \qquad \frac{}{(\lambda x.e)\ v \rightarrow e\{v/x\}, \text{None}}$$

$$\frac{e1 \rightarrow e1', o}{e1+e2 \rightarrow e1'+e2, o} \qquad \frac{e2 \rightarrow e2', o}{v+e2 \rightarrow v+e2', o} \qquad \frac{\text{“c1+c2=c3”}}{c1+c2 \rightarrow c3, \text{None}}$$

$$\frac{}{\texttt{fork}\ e \rightarrow 42, \text{Some}\ e}$$

$$\frac{ei \rightarrow ei', \text{None}}{e1;\ldots;ei;\ldots;en;. \rightarrow e1;\ldots;ei';\ldots;en;.} \qquad \frac{ei \rightarrow ei', \text{Some}\ e0}{e1;\ldots;ei;\ldots;en;. \rightarrow e0;e1;\ldots;ei';\ldots;en;.}$$

Notes

In this simple model:

- At each step, exactly one thread runs
- “Time-slice” duration is “one small-step”
- Thread-scheduling is non-deterministic
  - – So the operational semantics is too?
- Threads run “on the same machine”
- A “good final state” is some v1;…;vn;.
  - – Alternately, could “remove done threads”:

$$\frac{}{e1;...;ei;\ v;\ ej;\ ...\ ;en;. \rightarrow e1;...;ei;\ ej;\ ...;en;.}$$

Not enough

- These threads are really uninteresting
  - They can’t communicate
  - One thread’s steps can’t affect another
  - At most one final state is reachable
- One way: *mutable shared memory*
  - Many other communication mechanisms to come!
- Need:
  - Expressions to create, access, modify locations
  - A map from locations to values in program state

Changes to old stuff

Expressions: $e ::= \ldots \mid \texttt{ref}\ e \mid e1 := e2 \mid !e \mid l$

Values: $v ::= \ldots \mid l$

Heaps: $H ::= . \mid H, l \to v$

Thread pools: $T ::= . \mid e;T$

States: H, T

Change $e \to e', o$ to $H, e \to H', e', o$

Change $T \to T'$ to $H, T \to H', T'$

Change rules to modify heap (or not). 2 examples:

$$\frac{H, e1 \to H', e1', o}{H, e1\ e2 \to H', e1'\ e2, o} \qquad \frac{\text{“c1+c2=c3”}}{H, c1+c2 \to H, c3, \text{None}}$$

New rules

$$\frac{\mathtt{l} \text{ not in } \mathtt{H}}{\mathtt{H}, \mathtt{ref}\ v \rightarrow \mathtt{H}, \mathtt{l}\rightarrow\mathtt{v}, \mathtt{l}, \text{None}} \qquad \frac{}{\mathtt{H}, \mathtt{!}\ \mathtt{l} \rightarrow \mathtt{H}, \mathtt{H}\ (\mathtt{l})\ , \text{None}}$$

$$\frac{}{\mathtt{H}, \mathtt{l} := \mathtt{v} \rightarrow \mathtt{H}, \mathtt{l}\rightarrow\mathtt{v}, \mathtt{v}, \text{None}}$$

$$\frac{\mathtt{H}, e \rightarrow \mathtt{H}', e', o}{\mathtt{H}, \mathtt{!}\ e \rightarrow \mathtt{H}', \mathtt{!}\ e', o} \qquad \frac{\mathtt{H}, e \rightarrow \mathtt{H}', e', o}{\mathtt{H}, \mathtt{ref}\ e \rightarrow \mathtt{H}', \mathtt{ref}\ e', o}$$

$$\frac{\mathtt{H}, e \rightarrow \mathtt{H}', e', o}{\mathtt{H}, e1 := e2 \rightarrow \mathtt{H}', e1' := e2, o} \qquad \frac{\mathtt{H}, e \rightarrow \mathtt{H}', e', o}{\mathtt{H}, \mathtt{v} := e2 \rightarrow \mathtt{H}', \mathtt{v} := e2', o}$$

Now we can do stuff

We could now write “interesting examples” like

- Fork 10 threads, each to do a different computation
- Have each add its answer to an accumulator `l`
- When all threads finish, `l` is the answer
  - Increment another location to signify “done”

Problem: races

Races

```
l := !l + e
```

Just one interleaving that produces the wrong answer:

- Thread 1 reads `l`
- Thread 2 reads `l`
- Thread 1 writes `l`
- Thread 2 writes `l` – “forgets” thread 1’s addition

Communicating threads must synchronize

Languages provide synchronization mechanisms, e.g., locks or transactions

Outline

1. Lambda-calculus / operational semantics tutorial

2. Add threads and mutable shared-memory

3. *Add transactions; study weak vs. strong isolation*

4. Simple type system

5. *Type (and effect system) for strong = weak*
   - *And proof sketch*

Changes to old stuff

Expressions: `e ::= …| atomic e | inatomic e`

(No changes to values, heaps, or thread pools)

Atomic bit: `a ::=` ○ | ●

States: a, H, T

Change $H, e \rightarrow H', e', o$ to $a, H, e \rightarrow a, H', e', o$

Change $H, T \rightarrow H', T'$ to $a, H, T \rightarrow a, H', T'$

Change rules to modify atomic bit (or not). Examples:

$$\frac{a, H, e1 \rightarrow a', H', e1', o}{a, H, e1\ e2 \rightarrow a', H', e1'\ e2, o} \qquad \frac{\text{“c1+c2=c3”}}{a, H, c1+c2 \rightarrow a, H, c3, \text{None}}$$

The atomic-bit

- Intention of is to model “at most one transaction at a time”
  - ○ = No thread currently in transaction
  - ● = Exactly one thread currently in transaction

- Not how transactions are implemented
- But a good semantic definition for programmers
- Enough to model some (not all) weak/strong problems
  - Multiple small-steps within transactions
  - Unecessary just to define strong

Using the atomic-bit

Start a transaction, *only if no transaction is running*

$$\frac{}{\circ, H, \texttt{atomic}\ e \rightarrow \bullet, H, \texttt{inatomic}\ e, \text{None}}$$

End a transaction, *only if you have a value*

$$\frac{}{\bullet, H, \texttt{inatomic}\ v \rightarrow \circ, H, v, \text{None}}$$

Inside a transaction

$$\frac{a, H, e \rightarrow a', H', e', \text{None}}{\bullet, H, \texttt{inatomic}\ e \rightarrow \bullet, H, \texttt{inatomic}\ e', \text{None}}$$

- Says spawn-inside-transaction is dynamic error
  - Have also formalized other semantics
- Using unconstrained a and a’ is essential
  - A key technical “trick” or “insight”
  - For allowing closed-nested transactions
  - For allowing heap-access under strong
    - see next slide

Heap access

$$\frac{}{\circ, H, \,!\,l \rightarrow \circ, H, H(l), \text{None}}$$

$$\frac{}{\circ, H, l := v \rightarrow \circ, H, l \rightarrow v, 42, \text{None}}$$

Strong atomicity: If a transaction is running, no other thread may access the heap or start a transaction

- Again, just the semantics
- Again, unconstrained ***a*** lets the running transactions access the heap (previous slide)

Heap access

$$\frac{}{a, H, !\,l \rightarrow a, H, H(l), \text{None}}$$

$$\frac{}{a, H, l := v \rightarrow a, H, l \rightarrow v, v, \text{None}}$$

Weak-sla: If a transaction is running, no other thread may start a transaction

- A different semantics by changing four characters

A language family

- So now we have two languages
  - Same syntax, different semantics
- How are they related?
  - Every result under “strong” is possible under “weak”
    - Proof: Trivial induction (use same steps)
  - “Weak” has results not possible under “strong”
    - Proof: Example and exhaustive list of possible executions

Example

Distinguish “strong” and “weak”

- Let `a` be ○
- Let `H` map `l1` to 5 and `l2` to 6
- Let thread 1 be `atomic(l2:=7; l1:=!l2)`
  - sequencing (`e1; e2`) can be desugared as `(λ_. e2) e1`
- Let thread 2 be `l2:=4`

This example is not surprising

- Next language models some “surprises”

Weak-undo

Now: 3rd language modeling nondeterministic rollback

- Transaction can choose to rollback at any point
- Could also add explicit `retry` (but won't)

- Eager-update with an explicit undo-log
  - Lazy-update a 4th language we'll skip

- Logging requires still more additions to our semantics

Changes to old stuff

Expressions: $e ::= \ldots \mid \texttt{inatomic}(a,e,L,e0)$

$\mid \texttt{inrollback}(L,e0)$

Logs: $L ::= . \mid L,l \to v$

States (no change): a,H,T

Change $a,H,e \to a,H',e',o,L$

Overall step (no change) $a,H,T \to a,H',T'$

Change rules to “pass up” log. Examples:

$$\frac{a,H,e1 \to a',H',e1',o,L}{a,H,e1\ e2 \to a',H',e1'e2,o,L} \qquad \frac{\text{“c1+c2=c3”}}{a,H,c1+c2 \to a,H,c3,\text{None},.}$$

Logging writes

Reads are unchanged; writes log old value

- Orthogonal change from weak vs. strong

$$\frac{}{a, H, \, !\, l \rightarrow a, H, H(l), \text{None}, \; .}$$

$$\frac{}{a, H, l := v \rightarrow a, H, l \rightarrow v, v, \text{None}, \; ., l \rightarrow H(l)}$$

Start / end transactions

Start transactions *with an empty log and remembering initial expression (and no nested transaction)*

$$\frac{}{\circ, H, \texttt{atomic}\ e \rightarrow \bullet, H, \texttt{inatomic}(\circ, e, ., e0)\ \text{None}, .}$$

End transactions by passing up your whole log

$$\frac{}{\bullet, H, \texttt{inatomic}(\circ, v, L, e0)\ v \rightarrow \circ, H, v, \text{None}, L}$$

Inside a transaction

$$\frac{a, H, e \rightarrow a', H', e', \text{None}, L2}{\bullet, H, \mathtt{inatomic}(a, e, L1, e0) \rightarrow \bullet, H, \mathtt{inatomic}(a', e', L1@L2, e0), \text{None}, .}$$

- “Catches the log”
  - Keeps it as part of transaction state
  - Log only grows
  - Appends to a stack (see also rollback)
- Inner atomic-bit tracked separately
  - Still unconstrained, but need to know what it is for rollback (next slide)

Starting rollback

Start rollback provided no nested transaction

- Else you would forget the inner log!

$$\frac{}{\begin{array}{l}\bullet, H, \texttt{inatomic}(\circ, \texttt{e}, L1, \texttt{e0}) \rightarrow \\ \bullet, H, \texttt{inrollback}(L1, \texttt{e0}), \text{None}, \cdot\end{array}}$$

Rolling back

Pop off the log, restoring heap to what it was

● , *H*, inrollback(*L1*,l→*v* ,e0) →
● , *H*,l→*v*, inrollback(*L1*,e0), None , .

When log is empty, ready to restart

– no transaction is running

● , *H*, inrollback(. ,e0) →
○ , *H*, atomic e0, None , .

A bigger language family

- So now we have three languages
  - Same *source-language* syntax, different semantics
- How are they related?
  - Every result under “weak” is possible under “weak-undo”
    - Proof: Trivial induction (never rollback)
  - “Weak-undo” has results not possible under “weak”
    - Proof: Example and exhaustive list of possible executions
    - This is surprising
    - Examples use conditionals (easy to add)

Example (intuition)

```
initially l1=0, l2=0, l3=0
```

```
atomic {
  if(l1==1)
   l2=1;
  else
   l3=1;
}
```

```
l1=1;
l4=!l2;
l5=!l3;
```

can `!l4==1` and `!l5==1`

This is a speculative dirty read example we can model

Where are we

- Three increasingly permissive semantics

- Can we define a restriction on source programs sufficient to make languages equivalent?
  - More permissive than “reject all programs”
    - But likely still too restrictive in practice
  - Rigorously defined
  - Equivalence rigorously proven

Outline

1. Lambda-calculus / operational semantics tutorial

2. Add threads and mutable shared-memory

3. *Add transactions; study weak vs. strong isolation*

4. Simple type system

5. *Type (and effect system) for strong = weak*
   - *And proof sketch*

Types

A 2nd judgment $\Gamma \vdash \mathtt{e1} : \tau$ gives types to expressions

- – No derivation tree means “does not type-check”
- – Use a context to give types to variables in scope

“Simply typed lambda calculus” a starting point

Types: $\tau ::= \mathtt{int} \mid \tau \rightarrow \tau \mid \tau * \tau \mid \mathtt{ref}\ \tau$

Contexts: $\Gamma ::= . \mid \Gamma, \mathtt{x} : \tau$

$$\frac{}{\Gamma \vdash \mathtt{c} : \mathtt{int}} \qquad \frac{\Gamma \vdash \mathtt{e1} : \mathtt{int} \quad \Gamma \vdash \mathtt{e2} : \mathtt{int}}{\Gamma \vdash \mathtt{e1+e2} : \mathtt{int}} \qquad \frac{}{\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})}$$

$$\frac{\Gamma, \mathtt{x} : \tau 1 \vdash \mathtt{e} : \tau 2}{\Gamma \vdash (\lambda \mathtt{x}.\mathtt{e}) : \tau 1 \rightarrow \tau 2} \qquad \frac{\Gamma \vdash \mathtt{e1} : \tau 1 \rightarrow \tau 2 \quad \Gamma \vdash \mathtt{e2} : \tau 1}{\Gamma \vdash \mathtt{e1}\ \mathtt{e2} : \tau 2}$$

Notes

- Our declarative rules "infer" types, but we could just as easily adjust the syntax to make the programmer tell us

- These rules look arbitrary but have deep logical connections

The rest of the rules

$$\frac{\Gamma \vdash e1:\tau 1 \quad \Gamma \vdash e2:\tau 2}{\Gamma \vdash (e1,e2) : \tau 1*\tau 2} \qquad \frac{\Gamma \vdash e:\tau 1*\tau 2}{\Gamma \vdash e.1:\tau 1} \qquad \frac{\Gamma \vdash e:\tau 1*\tau 2}{\Gamma \vdash e.2:\tau 2}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{ref}\ e : \mathbf{ref}\ \tau} \qquad \frac{\Gamma \vdash e : \mathbf{ref}\ \tau}{\Gamma \vdash !e:\tau}$$

$$\frac{\Gamma \vdash e1 : \mathbf{ref}\ \tau \quad \Gamma \vdash e2:\tau}{\Gamma \vdash e1 := e2 : \tau} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{atomic}\ e : \tau}$$

Soundness

Reason we defined rules how we did:

If $\cdot \vdash e : \tau$ and after some number of steps

.;e becomes a;H';e1;…;en, then either all threads have terminated or at least one can take a step

An infinite number of different type systems have this property for our language, but want to show at least ours is one of them

Note: we wrote the semantics, so we chose what the "bad" states are. Extreme example: every type system is sound if any a;H;e1;…;en can step to a;H;42

Showing soundness

Soundness theorem is true, but how would we show it:

1. Extend our type system to program states (heaps and expressions with labels) *only for the proof*
2. Progress: Any well-typed program state has an expression that is a value or can take one step
3. Preservation: If a well-typed program state takes a step, the new state is well-typed

Perspective: “is well-typed” is just an induction hypothesis (preservation) with a property (progress) that describes what we want (e.g., don’t do !42)

Not really what we want

- Preservation gives us an invariant
  - Execution preserves well-typedness
- But want equivalence for strong, weak, weak-undo
  - Need a stronger invariant
  - Reject more source programs
  - Prove stronger invariant is preserved
- Our choice for stronger invariant:
  - Every heap location is accessed only inside transactions or only outside transactions
  - Formalize with a stricter type system that uses effects

Hard theorems

Use this new to-be-defined type system that ensures any mutable memory is either:

- Only accessed in transactions
- Only accessed outside transactions

(Helper) theorem: Preservation holds for this type system

Theorem: *If a program type-checks*, it has the same possible behaviors under strong and weak

Theorem: *If a program type-checks*, it has the same possible behaviors under weak and weak-undo

- *Almost: extra garbage and intermediate states under weak-undo*

Outline

1. Lambda-calculus / operational semantics tutorial
2. Add threads and mutable shared-memory
3. *Add transactions; study weak vs. strong isolation*
4. Simple type system
5. *Type (and effect system) for strong = weak*
   - *And proof sketch*

Effects

Code can be used inside transactions, outside transactions, or both

Each memory location can be accessed only inside transactions or only outside transactions

$$p ::= \mathit{ot} \mid \mathit{wt}$$

$$\varepsilon ::= p \mid \mathit{both}$$

$$\tau ::= \texttt{int} \mid \tau \xrightarrow{\varepsilon} \tau \mid \tau \texttt{ * } \tau \mid \texttt{ref(p)}\ \tau$$

$$\Gamma ::= \bullet \mid \Gamma, x{:}\tau$$

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

“Under context Γ, e has type τ and stays on the side of the partition required by effect ε”

Effects in general

- Type-and-effect systems describe computations
  - What does evaluating some expression do
  - Examples: memory accessed, locks acquired, messsages sent, time consumed, etc.
- They are all remarkably similar
  - Values have the “empty effect”
  - Function types have a “latent effect”
  - Subeffecting for function calls and conditionals
  - “Effect masking” in the “interesting rules”

General rules

Almost every effect system looks like this:

$$\frac{}{\Gamma;\varepsilon \vdash \texttt{c}:\texttt{int}} \qquad \frac{\Gamma;\varepsilon \vdash \texttt{e1}:\texttt{int} \quad \Gamma;\varepsilon \vdash \texttt{e2}:\texttt{int}}{\Gamma;\varepsilon \vdash \texttt{e1+e2}:\texttt{int}} \qquad \frac{}{\Gamma;\varepsilon \vdash \texttt{x}:\Gamma\texttt{(x)}}$$

$$\frac{\Gamma,\texttt{x}:\tau\texttt{1};\varepsilon' \vdash \texttt{e}:\underset{\varepsilon'}{\tau\texttt{2}}}{\Gamma;\varepsilon \vdash (\lambda\texttt{x.e}):\tau\texttt{1}\rightarrow\tau\texttt{2}} \qquad \frac{\Gamma;\varepsilon \vdash \texttt{e1}:\tau\texttt{1}\rightarrow\tau\texttt{2} \quad \Gamma;\varepsilon \vdash \texttt{e2}:\tau\texttt{1}}{\Gamma;\varepsilon \vdash \texttt{e1 e2}:\tau\texttt{2}}$$

$$\frac{\Gamma;\varepsilon' \vdash \texttt{e}:\tau \quad \varepsilon' \leq \varepsilon}{\Gamma;\varepsilon \vdash \texttt{e}:\tau} \qquad \frac{}{\varepsilon \leq \varepsilon} \qquad \frac{\varepsilon 1 \leq \varepsilon 2 \quad \varepsilon 2 \leq \varepsilon 3}{\varepsilon 1 \leq \varepsilon 3}$$

Plug in special stuff

- “Effect masking” in rule for **`atomic`** (needed for nesting)
- Allocation need not constrain ε
  - not thread-shared (yet)

$$\frac{\Gamma;\varepsilon \vdash \texttt{e1}:\texttt{ref}(\varepsilon)\,\tau \qquad \Gamma;\varepsilon \vdash \texttt{e2}:\tau}{\Gamma;\varepsilon \vdash \texttt{e1:=e2} : \tau} \qquad \frac{\Gamma;\varepsilon \vdash \texttt{e}:\tau}{\Gamma;\varepsilon \vdash \texttt{!e}: \tau}$$

$$\frac{\Gamma;\varepsilon \vdash \texttt{e}:\tau}{\Gamma;\varepsilon \vdash \texttt{ref e}: \texttt{ref}(p)\,\tau} \qquad \frac{\Gamma;wt \vdash \texttt{e}:\tau}{\Gamma;\varepsilon \vdash \texttt{atomic e} : \tau}$$

$$\frac{\Gamma;ot \vdash \texttt{e}:\tau}{\Gamma;ot \vdash \texttt{spawn e}:\tau} \qquad \frac{}{both \lesssim wt} \qquad \frac{}{both \lesssim ot}$$

Now what

- Extend to program states
  - (add rules for inatomic, labels, etc.)
  - Needed to prove preservation
- Prove preservation
  - Needed to prove equivalence
- Prove equivalence
  - Long, but can sketch the key ideas…

The proof

The proofs are dozens of pages and a few person-months (lest a skipped step hold a surprise)

But the high-level picture is illuminating…

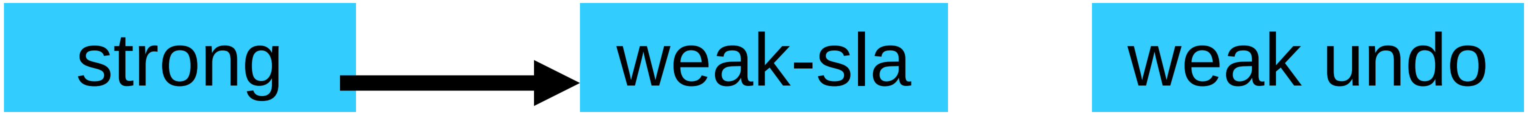
The proof

The proofs are dozens of pages and a few person-months (lest a skipped step hold a surprise)

But the high-level picture is illuminating…

If possible in strong, then possible in weak-sla

- trivial: don’t ever violate isolation



strong

weak-sla

weak undo

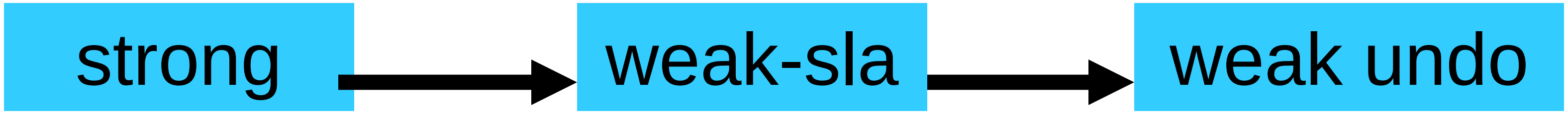
The proof

The proofs are dozens of pages and a few person-months (lest a skipped step hold a surprise)

But the high-level picture is illuminating…

If possible in weak-sla, then possible in weak undo

- trivial: don't ever abort



strong → weak-sla → weak undo

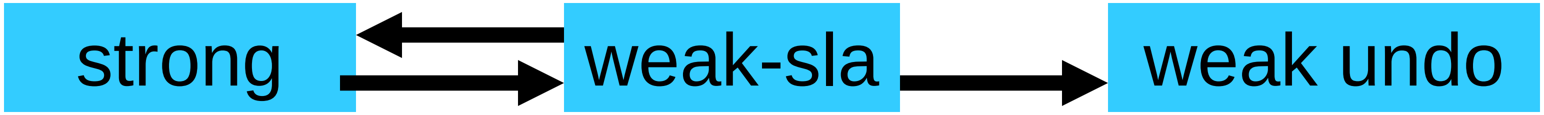
The proof

The proofs are dozens of pages and a few person-months (lest a skipped step hold a surprise)

But the high-level picture is illuminating…

If possible in weak-sla, then possible in strong

- Current transaction is serializable thanks to the type system (can permute with other threads)
- Earlier transactions serializable by induction



A little more detail…

Strengthened induction hypothesis (always the key!):

- If *e* type-checks and can reach state *S* in *n* steps under weak-sla, then can reach state *S* in *n* steps under strong
- And if *S* has a transaction running, then the *n* steps in strong can end with the transaction not interleaved with anything else

Proof: By induction on n…

Serialization

- So weak took n-1 steps to *S′* and then 1 step to *S*
  - By induction strong can take n-1 steps to *S′*
- Hard case: A transaction is running but 1 more step is by another thread
  - So need to show that in strong sequence this step could be taken *before the transaction started*
  - Follows from step-commutation lemmas that rely on program states type-checking
  - And program states type-check due to Preservation
- A “classic” serialization argument, relying at the crucial moment on our type-and-effect system

The proof

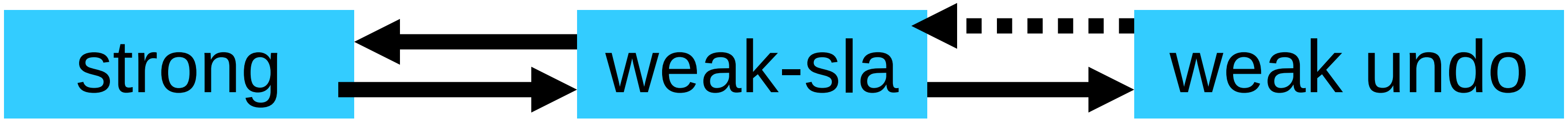
The proofs are dozens of pages and a few person-months (lest a skipped step hold a surprise)

But the high-level picture is illuminating…

If possible in weak undo, then possible in weak-sla?

– Really need that abort is correct

– And that’s hard to show, especially with interleavings from weak isolation…



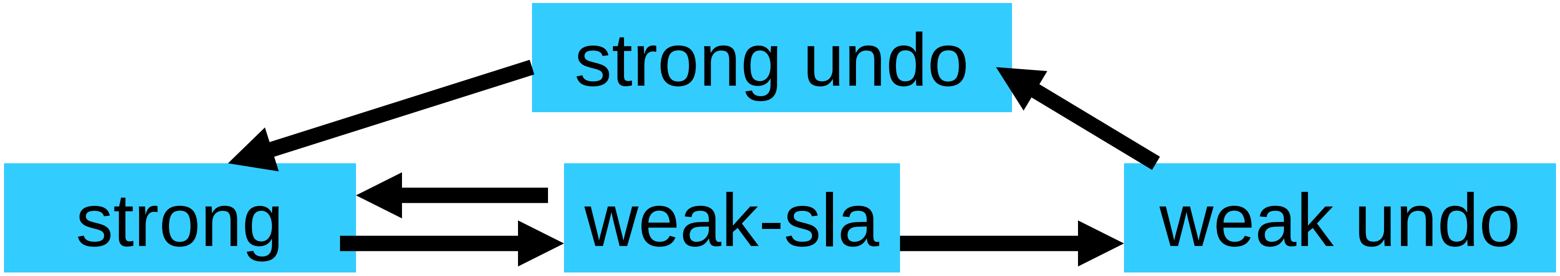
The proof

The proofs are dozens of pages and a few person-months (lest a skipped step hold a surprise)

But the high-level picture is illuminating…

If possible in weak undo, then possible in weak-sla?

- Define strong undo for sake of the proof
- Can show abort is correct without interleavings



Some more details

- weak-undo to strong-undo proof is just like weak-sla to strong
  - With more cases for the rollback
- strong undo to strong has nothing to do with serialization
  - Just show rollback produces a “similar enough state”
    - Exactly the same, plus potential “garbage”
    - Because we don’t rollback allocations

Why we formalize, redux

Thanks to the formal semantics, we:

- Had to make precise definitions
- Know we did not skip cases (at least in the model)
- Learned the *essence of why* the languages are equivalent under partition
  - Weak interleavings are serializable
  - Rollback is correct
  - And these two arguments compose
    - good “proof engineering”

My tentative plan

1. Basics: language constructs, implementation intuition (Tim next week)

2. Motivation: the TM/GC Analogy

3. Strong vs. weak atomicity
   - And optimizations relevant to strong

4. Formal semantics for transactions / proof results
   - Including formal-semantics tutorial

5. Brief mention: memory-models, continuations

Relaxed memory models

Modern languages don't provide *sequential consistency*

> *"the results of any execution is the same as if the operations of all the [threads] were executed in some sequential order, and the operations of each individual [thread] appear in this sequence in the order specified by its program"*

Why not?

1. Lack of hardware support
2. Prevents otherwise sensible & ubiquitous compiler transformations (e.g., copy propagation)

Semi-digression

Compiler transformation may be wrong only because of shared-memory multithreading

“Threads cannot be implemented as a library”

[Boehm, PLDI05]

```
int x=0, y=0;
void f1(){ if(x) ++y; }
void f2(){ if(y) ++x; }
// main: run f1, f2 concurrently
// can compiler implement f2 as ++x; if(!y) --x;
```

Want to forbid compiler from introducing race conditions

– Transformation above would do this!

But allow reorderings

But most compilers do reorder reads and writes

- common subexpressions, redundant assignments, copy propagation, …
- and hardware does to (read from store buffers

So this code is broken

Requiring assertion to hold is deemed too restrictive on implementations

```
initially x=0, y=0
```

```
x = 1;
y = 1;
```

```
r = y;
s = x;
assert(s>=r);//invalid
```

Needing a memory model

So safe languages need two complicated definitions

1. What is “properly synchronized”?
   - Popular answer:
     “data-race free implies sequential consistency”
2. What can compiler & hardware do with “bad code”?

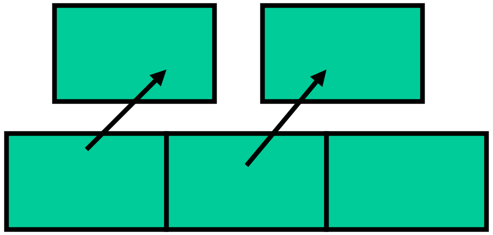
(Unsafe languages need (1) but not (2))

A flavor of simplistic ideas and the consequences…

Simplistic ideas

“Properly synchronized” ➔ All thread-shared mutable memory accessed in transactions

Consequence: *Data-handoff* code deemed “bad”



```
//Producer
tmp1=new C();
tmp1.x=42;
atomic {
 q.put(tmp1);
}
```

```
//Consumer
atomic {
 tmp2=q.get();
}
tmp2.x++;
```

Ordering

Can get “strange results” for bad code

- Need rules for what is “good code”

```
initially x=0, y=0
```

```
x = 1;

y = 1;
```

```
r = y;

s = x;
assert(s>=r);//invalid
```

Ordering

Can get “strange results” for bad code

- Need rules for what is “good code”

```
initially x=0, y=0
```

```
x = 1;
sync(lk){}
y = 1;
```

```
r = y;
sync(lk){} //same lock
s = x;
assert(s>=r);//valid
```

Ordering

Can get “strange results” for bad code

- Need rules for what is “good code”

```
initially x=0, y=0
```

```
x = 1;
atomic{}
y = 1;
```

```
r = y;
atomic{}
s = x;
assert(s>=r);//???
```

If this is good code, existing STMs are wrong

Ordering

Can get “strange results” for bad code

- Need rules for what is “good code”

```
initially x=0, y=0
```

```
x = 1;
atomic{z=1;}
y = 1;
```

```
r = y;
atomic{tmp=0*z;}
s = x;
assert(s>=r);//???
```

“Conflicting memory” a slippery ill-defined slope

Lesson

It is not clear when transactions are ordered, but languages need memory models

Corollary: This could/should delay adoption of transactions in well-specified languages

My tentative plan

1. Basics: language constructs, implementation intuition (Tim next week)

2. Motivation: the TM/GC Analogy

3. Strong vs. weak atomicity
   - And optimizations relevant to strong

4. Formal semantics for transactions / proof results
   - Including formal-semantics tutorial

5. Brief mention: memory-models, continuations

First-class continuations

- First-class continuations (“call-cc”) make call-stacks first class (store in fields, return from functions, etc.)
  - Unlike exceptions, you can “jump back in”
  - Very powerful for idioms like user-defined iterators or coroutines
- But what does jumping back into a completed transaction mean?
  - It depends
  - Need programmer to indicate somewhere
    - on transaction, on continuation creation, on continuation invocation
- See SCHEME 2007 paper

My tentative plan

1. Basics: language constructs, implementation intuition (Tim next week)
2. Motivation: the TM/GC Analogy
3. Strong vs. weak atomicity
   - And optimizations relevant to strong
4. Formal semantics for transactions / proof results
   - Including formal-semantics tutorial
5. Brief mention: memory-models, continuations

So …

What I hope you learned

- Transactions need integrating into modern complicated programming languages
  - This is not easy
  - Many language features have non-trivial interactions
  - Formal semantics has a role to play
  - Weak isolation is worse than you think
- Transactions a big step forward for shared-memory multithreading
  - But doesn't make parallel programming easy
  - Hopefully enough *easier* to make a difference