CSEP505: Programming Languages

Lecture 9: Fancier OOP; Concurrency

Dan Grossman

Spring 2006

Now what?

- That's basic class-based OOP
  - Not all OOPLs use classes
    (Javascript, Self, Cecil, …)
    (may show you some cool stuff time permitting)
- Now some "fancy" stuff
  - Typechecking
  - Multiple inheritance; multiple interfaces
  - Static overloading
  - Multimethods
  - Revenge of bounded polymorphism

Typechecking

We were sloppy:

talked about types without “what are we preventing”

1. In pure OO, stuck if v.m(v1,…,vn) and v has no m method (taking *n* args)
   - “No such method”
2. Also if ambiguous (multiple methods with same name; no best choice)
   - “No best match”

Multiple inheritance

Why not allow `C extends C1,…,Cn {…}`

(and `C`≤`C1`, …, `C`≤`Cn`)

What everyone agrees on: C++ has it, Java doesn't

We'll just consider some problems it introduces and how (multiple) interfaces avoids some of them

Problem sources:

1. Class hierarchy is a dag, not a tree
2. Type hierarchy is a dag, not a tree

Method-name clash

What if C extends C1, C2 which both define m?

Possibilities:

1. Reject declaration of C
   - Too restrictive with diamonds (next slide)
2. Require C overrides m
   - Possibly with *directed resends*
3. "Left-side" (C1) wins
   - Question: does cast to C2 change what m means?
4. C gets both methods (implies incoherent subtyping)
5. Other? (I'm just brainstorming)

Diamonds

- If C extends C1, C2 and C1, C2 have a common (transitive) superclass D, we have a diamond
  - Always have one with multiple inheritance and a topmost class (Object)
- If D has a field f, does C have one field or two?
  - C++ answer: yes ☺
- If D has a method m, see previous slide.
- If subsumption is coercive, we may be incoherent
  - Multiple paths to D

Implementation issues

- Multiple-inheritance semantics often muddied by wanting "efficient member lookup"
  - If "efficient" is compile-time offset from self pointer, then multiple inheritance means subsumption must "bump the pointer"
  - Roughly why C++ has different sorts of casts

- Preaching: Get the semantics right first

Digression: casts

A “cast” can mean too many different things (cf. C++):

Language-level:

- Upcast (no run-time effect)
- Downcast (failure or no run-time effect)
- Conversion (key question: *round-tripping*)
- “Reinterpret bits” (not well-defined)

Implementation level

- Upcast (see last slide)
- Downcast (check the tag)
- Conversion (run code to make a new value)
- “Reinterpret bits” (no effect)

Least supertypes

[Related to homework 4 extra credit]

For e1 ? e2 : e3

- e2 and e3 need the same type
- But that just means a common supertype
- But which one? (The least one)
  - But multiple inheritance means may not exist!

Solution:

- Reject without explicit casts

Multiple inheritance summary

1. Method clashes (same method m)
2. Diamond issues (fields in top of diamond)
3. Implementation issues (slower method lookup)
4. Least supertypes (may not exist)

Multiple interfaces (type without code or fields) have problems (3) and (4)

(and 3 isn't necessarily a problem)

Now what?

- That's basic class-based OOP
  - Not all OOPLs use classes
    (Javascript, Self, Cecil, …)
    (may show you some cool stuff time permitting)
- Now some "fancy" stuff
  - Typechecking
  - Multiple inheritance; multiple interfaces
  - Static overloading
  - Multimethods
  - Revenge of bounded polymorphism

Static overloading

- So far: Assume every method name unique
  - (same name in subclass meant override; required subtype)
- Many OO languages allow same name, different argument types:

```
A f(B b) {…}
C f(D d, E e) {…}
F f(G g, H h) {…}
```

- Changes method-lookup definition for e.m(e1,…en)
  - Old: lookup m via (run-time) class of e
  - New: lookup m via (run-time) class of e *and* (compile-time) types of e1,…,en

Ambiguity

Because of subtyping, multiple methods can match!

"Best match" rules are complicated. One rough idea:

- Fewer subsumptions is better match
- If tied, subsume to *immediate supertypes* & *recur*

Ambiguities remain

1. `A f(B)` or `C f(B)` (usually disallowed)
2. `A f(B)` or `A f(C)` and `f(e)` where `e` has a subtype of `B` and `C` but `B` and `C` are incomparable
3. `A f(B,C)` or `A f(C,B)` and `f(e1,e2)` where `e1` and `e2` have type `B` and `B` $\leq$ `C`

Now what?

- That's basic class-based OOP
  - Not all OOPLs use classes
    (Javascript, Self, Cecil, …)
    (may show you some cool stuff time permitting)
- Now some "fancy" stuff
  - Typechecking
  - Multiple inheritance; multiple interfaces
  - Static overloading
  - Multimethods
  - Revenge of bounded polymorphism

Multimethods

Static overloading just saves keystrokes via shorter method names

- Name-mangling on par with syntactic sugar

Multiple (dynamic) dispatch (a.k.a. multimethods) much more interesting: Method lookup based on (run-time) classes of all arguments

A natural generalization: “receiver” no longer special

So may as well write **`m(e1,…,en)`**

instead of **`e1.m(e2,…,en)`**

Multimethods example

```
class A { int f; }
class B extends A { int g; }
Bool compare(A x, A y) { x.f==y.f }
Bool compare(B x, B y) { x.f==y.f && x.g==y.g }
Bool f(A x, A y, A z) { compare(x,y) &&
                        compare(y,z) }
```

- `compare(x,y)` calls first version unless both arguments are B's
  - Could add "one of each" methods if you want different behavior

Pragmatics; UW

Not clear where multimethods should be defined

- No longer “belong to a class” because receiver not special

Multimethods are “more OO” because dynamic-dispatch is the essence of OO

Multimethods are “less OO” because without distinguished receiver the “analogy to physical objects” is less

UW CSE a multimethods leader for years (Chambers)

Revenge of ambiguity

- Like static overloading, multimethods have “no best match” problems
- Unlike static overloading, the problem does not arise until run-time!

Possible solutions:

1. Run-time exception
2. Always define a best-match (e.g., Dylan)
3. A conservative type system

Now what?

- That's basic class-based OOP
  - Not all OOPLs use classes
    (Javascript, Self, Cecil, …)
    (may show you some cool stuff time permitting)
- Now some "fancy" stuff
  - Typechecking
  - Multiple inheritance; multiple interfaces
  - Static overloading
  - Multimethods
  - Revenge of bounded polymorphism

Still want generics

OO subtyping no replacement for parametric polymorphism

So have both

Example:

```
/* 3 type constructors (e.g., Int Set a type) */
interface 'a Comparable { Int f('a,'a); }
interface 'a Predicate { Bool f('a); }
class 'a Set {
 …
 constructor('a Comparable x){…}
 unit add('a x){…}
 'a Set functional_add('a x){…}
 'a find('a Predicate x) {…}
}
```

Worse ambiguity

"Interesting" interaction with overloading or multimethods

```
class B {
 Int f(Int C x){1}
 Int f(String C x){2}
 Int g('a x) { self.f(x) }
}
```

Whether match is found depends on instantiation of `'a`

Cannot resolve static overloading at compile-time without code duplication

At run-time, need run-time type information

- Including instantiation of type constructors
- Or restrict overloading enough to avoid it

Wanting bounds

As expected, with subtyping and generics, want bounded polymorphism

Example:

```
interface I { unit print(); }
class ('a≤I) Logger {
 'a item;
 'a get() { item.print(); item }
}
```

w/o polymorphism, **`get`** would return an **`I`** (not useful)

w/o the bound, **`get`** could not **`send`** print to **`item`**

Fancy example

With forethought, can use bounds to avoid some subtyping limitations

"Not on the final"

(Example lifted from Abadi/Cardelli text)

```
/* Herbivore1 ≤ Omnivore1 unsound */
interface Omnivore1  { unit eat(Food); }
interface Herbivore1 { unit eat(Veg); }
/* T Herbivore2 ≤ T Omnivore2 sound for any T */
interface ('a≤Food) Omnivore2 { unit eat('a); }
interface ('a≤Veg)  Herbivore2 { unit eat('a); }
/* subtyping lets us pass herbivores to feed
   but only if food is a Veg */
unit feed('a food, 'a Omnivore animal) {
   animal.eat(food);
}
```

Concurrency

- PL support for concurrency a huge topic
- We'll just do explicit threads plus
  - Shared memory (locks and transactions)
  - Synchronous message-passing (CML)
  - Transactions last (wrong logic, but CML is hw 5)
- Skipped topics
  - Futures
  - Asynchronous methods (joins, tuple-spaces, …)
  - Data-parallel (vector) languages
  - …

Threads

High-level: “Communicating sequential processes”

Low-level: “Multiple stacks plus communication”

```
(* thread.mli; compile with -vmthread threads.cma
   ON THE LEFT *)
type t (* a thread handle *)
val create : ('a->'b) -> 'a -> t (*run new thread*)
val self   : unit -> t (*who am I? (not OO self)*)
```

Why? Any one of

1. Performance (multiprocessor or mask I/O latency)
2. Isolation (separate errors or responsiveness)
3. Natural code structure (1 stack not enough)

Running threads

- A closure: “object that can be run concurrently”
- **`Thread.create`**: “start a new empty stack”
  - Thread terminates on return/uncaught-exception

- OO languages usually make same distinction
  - **`new Thread(…)`** something that can be run
  - **`thr.run()`** start a new empty stack

- Alternative: start with copy of spawner’s stack
  - UNIX **`fork`**

Preemption

- We'll assume *pre-emptive scheduling*
  - Running thread can be stopped whenever
  - **`yield : unit->unit`** a semantic no-op (a "hint")
- Formalism easy:
  - Program state: H;e1;e2;…;en
    - For message-passing, see pi-calculus
  - Small-step:
    - Any ei might become ei' next with H now H'
    - create adds a new expression
    - A value in the "thread-pool" is removable
  - Nondeterministic

A “library”?

> Threads cannot be implemented as a library
> Hans-J. Boehm, PLDI2005

- Does not mean you need new language constructs
  - thread.mli, mutex.mli, condition.mli is fine
- Does mean the compiler must know threads exist
- (See paper for more compelling examples)

```
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; */
```

Communication

If threads did nothing other threads needed to see, we'd be done

- Best to do as little communication as possible
- E.g., don't mutate shared data unnecessarily

One way to communicate: Shared memory

- One thread writes to a ref, another reads it
- Sounds nasty with pre-emptive scheduling
- Hence synchronization control
  - Taught in O/S for historical reasons!
  - Fundamentally about restricting interleavings

Join

"Fork-join" parallelism

- Simple approach good for "farm out independent subcomputations, then merge results"

```
(*suspend caller until/unless arg terminates*)
val join : t -> unit
```

Common pattern (in C syntax; Caml also simple):

```
data_t data[N];
result_t results[N];
thread_t tids[N];
for(i=0; i < N; ++i)
  tids[i] = create(f,&data[i], &results[i]);
for(i=0; i < N; ++i)
  join(tids[i]);
// now use results (e.g., sum them)
```

Locks (a.k.a. mutexes)

```
(* mutex.mli *)
type t (* a mutex *)
val create : unit -> t
val lock   : t -> unit (* may block *)
val unlock : t -> unit
```

- Caml locks do not have two common features:
  - Reentrancy
    (changes semantics of **`lock`**)
  - Banning non-holder release
    (changes **`unlock`** semantics)
- Also want condition variables (see **`condition.mli`**)
  - also known as wait/notify

Example

```
type acct = { lk : Mutex.t;
              bal : float ref;
              avail : float ref}
let mkAcct () =
  {lk=Mutex.create(); bal=ref 0.0; avail=ref 0.0}

let get a f =
  Mutex.lock a.lk;
  (if(!(a.avail) > f)
  then (a.bal := !(a.bal) -. f;
        a.avail := !(a.avail) -.f));
  Mutex.unlock a.lk

let put a f =
  Mutex.lock a.lk;
  a.bal := !(a.bal) +. f;
  a.avail := !(a.avail) +.(if f<500. then f else 500.);
  Mutex.unlock a.lk
```

Getting it wrong

Errors sequential programs never have:

1. Races (logic error due to bad interleaving)
   - Data race: concurrent access to same shared variable and at least one access is a write
   - Higher-level race: inconsistent view of multiple variables, data structure, etc.
2. Deadlocks (cyclic waiting)
3. Priority inversion

Can happen with any synchronization mechanism

- Consider locks…

Getting it right

Races

- Results from too little synchronization
- Sound-but-conservative data-race prevention:
  - For every (thread-shared, mutable) object, there exists a lock held on every access
  - Lots of type-system and tools work in last 10 years
- Higher-level races much tougher for the PL to help

Deadlock

- Results from too much synchronization
- Sound-but-conservative deadlock protection:
  - Global partial-order on lock acquisition order
  - Some type-system and tools work in last 10 years

Non-modular

Even if you get it right today, tomorrow's code change can have drastic effects

- Every bank account has its own lock works great until you want an "atomic transfer" function
  - One lock at a time: race
  - Both locks first: deadlock with parallel untransfer
- Same idea in JDK1.4:

```
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 also synchronized
```

Atomicity is modular

Atomicity restricts interleaving, regardless of other code from yesterday or tomorrow

- More on this after CML

```
let xfer src dst x =
 Mutex.lock src.lk;
 Mutex.lock dst.lk;
 src.bal <- src.bal-x;
 dst.bal <- dst.bal+x;
 Mutex.unlock src.lk;
 Mutex.unlock dst.lk
```

```
atomic:
  (unit->α)->α

let xfer src dst x =
atomic (fun()->
 src.bal <- src.bal-x;
 dst.bal <- dst.bal+x
)
```

Where are we

- Thread creation
- Communication via shared memory
  - Synchronization w/ join, locks
- Message passing a la Concurrent ML
  - Very elegant
  - Can wrap synchronization abstractions to make new ones
  - In my opinion, quite under-appreciated
- Back to shared memory for software transactions

The basics

```
(* event.mli; Caml's version of CML *)
type 'a channel (* messages passed on channels *)
val new_channel : unit -> 'a channel

type 'a event (* when sync'ed on, get an 'a *)
val send : 'a channel -> 'a -> unit event
val receive : 'a channel -> 'a event
val sync : 'a event -> 'a
```

- Send and receive return "events" immediately
- Sync blocks until the "event happens"
- Separating these is key in a few slides

Simple version

Helper functions to define blocking sending/receiving

- Message sent when 1 thread sends, another receives
- One will block waiting for the other

```
let sendNow ch a = sync (send ch a) (* block *)
let recvNow ch = sync (receive ch)  (* block *)
```

Note: In SML, the CML book, etc:

send=sendEvt, receive=recvEvt,
sendNow=send, recvNow=recv

Example

Make a thread to handle changes to a bank account

- mkAcct returns 2 channels for talking to the thread

```
type action = Put of float | Get of float
type acct = action channel * float channel
let mkAcct () =
  let inCh  = new_channel() in
  let outCh = new_channel() in
  let bal   = ref 0.0       in (* state *)
  let rec loop () =
    (match recvNow inCh with (* blocks *)
      Put f -> bal := !bal +. f;
    | Get f -> bal := !bal -. f);(*allows overdraw*)
    sendNow outCh !bal; loop ()
  in Thread.create loop (); (inCh,outCh)
```

Example, continued

get and put functions use the channels

```
let get acct f =
  let inCh,outCh  = acct in
  sendNow inCh (Get f); recvNow outCh
let put acct f =
  let inCh,outCh  = acct in
  sendNow inCh (Put f); recvNow outCh
```

Outside the module, don't see threads or channels!

```
type acct
val mkAcct : unit -> acct
val get : acct->float->float
val put : acct->float->float
```

Simpler example

- A stream is an infinite set of values
  - Don't compute them until asked
  - Again we could hide the channels and thread

```
let squares = new_channel()
let rec loop i =
  sendNow squares (i*i);
  loop (i+1)
let _ = create loop 1

let one  = recvNow squares
let four = recvNow squares
let nine = recvNow squares
…
```

So far

- sendNow and recvNow let us do synchronous message passing
- Abstraction lets us hide concurrency behind interfaces
- But a thread picks a channel and blocks on it until the *rendezvous*
  - That's too restrictive
  - Example: add results from two channels

The cool stuff

```
type 'a event (* when sync'ed on, get an 'a *)
val send : 'a channel -> 'a -> unit event
val receive : 'a channel -> 'a event
val sync : 'a event -> 'a channel

val choose : 'a event list -> 'a event
val wrap : 'a event -> ('a -> 'b) -> 'b event
```

- **`choose`**: block until some event happens
- **`wrap`**: An event with the function as post-processing
  - Can wrap as many times as you want
- HW: Bank account with 2 in-channels instead of datatype

“And from or”

- Choose seems great for “until one happens”
- But a little coding trick gets you “until all happen”:

```
let add in1 in2 out =
  let ans = sync(choose[
   wrap (receive in1)
        (fun i -> sync (receive in2) + i);
   wrap (receive in2)
        (fun i -> sync (receive in1) + i)])
  in
  sync (send out ans)
```

Circuits

If you're an electrical engineer:

- send and receive are ends of a gate
- wrap wires up "what's next"
- choose is a multiplexer (no control over which)

So after you wire something up, you sync to say "wait for communication from the outside"

If you're a UNIX hacker:

- UNIX select is "sync of choose"
- A pain that they can't be separated

Remaining comments

- The ability to build bigger events from smaller ones is very powerful
- Synchronous message passing, well, synchronizes
- Also need a wrap_abort to support timeouts
- Homework shows you something fancy using all this
- By the way, Caml's implementation of CML is in terms of queues and locks
  - Works okay on a uniprocessor

Where are we

- Thread creation
- Communication via shared memory
  - Synchronization w/ join, locks
- Message passing a la Concurrent ML
  - Very elegant
  - Can wrap synchronization abstractions to make new ones
  - In my opinion, quite under-appreciated
- Back to shared memory for software transactions

Atomicity is modular

Atomicity restricts interleaving, regardless of other code from yesterday or tomorrow

- More on this after CML

```
let xfer src dst x =
 Mutex.lock src.lk;
 Mutex.lock dst.lk;
 src.bal <- src.bal-x;
 dst.bal <- dst.bal+x;
 Mutex.unlock src.lk;
 Mutex.unlock dst.lk
```

```
atomic:
  (unit->α)->α

let xfer src dst x =
atomic (fun()->
 src.bal <- src.bal-x;
 dst.bal <- dst.bal+x
)
```

Different viewpoints

Software transactions good for:

1. Software engineering (avoid races & deadlocks)
2. Performance (optimistic "no conflict" without locks)

key semantic decisions depend on emphasis

Research should be guiding:

A. New hardware with transactional support
B. Language implementation for expected platforms

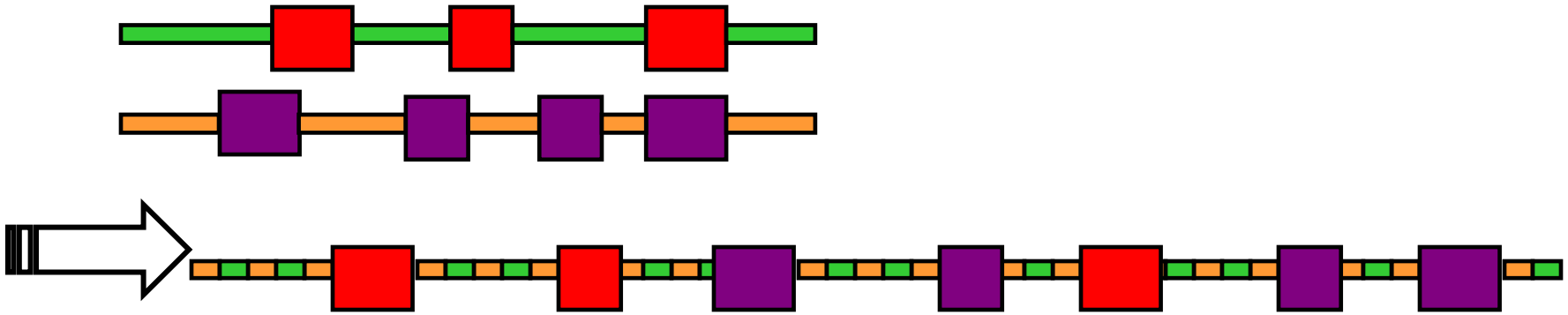
"is this a hw or sw question or both"

Full disclosure: I'm a "1B extremist"

Strong atomicity

(behave as if) no interleaved computation

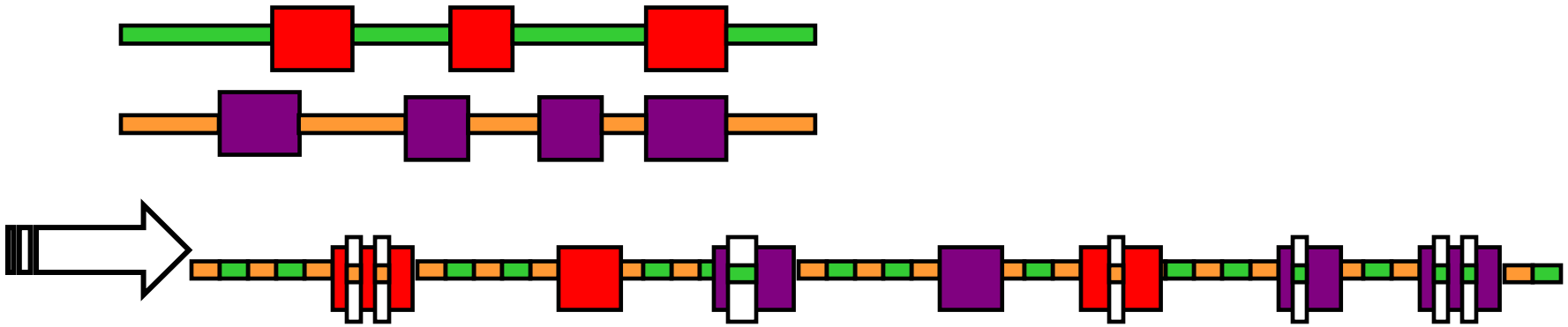
- Before a transaction "commits"
  - Other threads don't "read its writes"
  - It doesn't "read other threads' writes"
- This is just the semantics
  - Can "behave as though transaction started later"



Weak atomicity

(behave as if) no interleaved transactions

- Before a transaction "commits"
  - Other threads' transactions don't "read its writes"
  - It doesn't "read other threads' transactions' writes"
- This is just the semantics
  - Can "behave as though transaction started later"



Wanting strong

Software-engineering advantages of strong atomicity

1. Sequential reasoning in transaction
   - Strong: sound
   - Weak: only if all (mutable) data is not simultaneously accessed outside transaction
2. Transactional data-access a local code decision
   - Strong: new transaction "just works"
   - Weak: what data "is transactional" is global
3. Fairness: Long transactions don't starve others
   - Strong: true; no other code sees effects
   - Weak: maybe false for non-transactional code

Where are we

- Atomicity, especially strong sounds great
  - Another Dan mantra:

  "Atomic is to concurrency as garbage collection is to memory management"
- But there are some tough issues
  - Irreversible operations ("launch missiles")
- Now: a note on implementations
  - Like GC: It does the whole-program protocols!
  - Like GC: In practice, use low-level tricks

Implementing atomicity

Easier than you think (modulo performance):

Option 1:

- In a transaction, ensure exclusive owernship of everything you touch (avoid races)
- Rollback somebody on contention (avoid deadlock)
- Shameless plug: elegant and fast on a uniprocessor

Option 2:

- In a transaction, compute with private *version* of memory
- On commit, use a fancy protocol to ensure consistency and progress

The key point

There is data-level synchronization control, but now it's the compiler and run-time

- Programmer declaratively restricts interleavings

Language-implementation solutions scale because their implementation complexity does not depend on program size

But they are "one-size-fits-most"