A Type-Based Approach for Data-Centric Synchronization

Frank Tip

joint work with Mandana Vaziri, Julian Dolby and Jan Vitek

Synchronization is difficult in OO programs

- **Possibility of data races or deadlock**
  - Low-level data race: 2 concurrent access to a location (at least one write) with no synchronization between them
- **Locking discipline: non-local reasoning**
- **Granularity of locking**
  - Even if every shared access is protected, data may still end up in an inconsistent state (e.g., high-level races)

- **Objective of this work**
  - Data-centric mechanism for synchronization constraints
  - Leverage OO structure
  - Automated synchronization inference
  - Lightweight ownership type system that guarantees serializability and enables separate compilation
  - improved support for linked data structures

Example: High-Level Data Race in java.util.Vector

```
public class Vector<E> {
  protected Object[] elementData;
  protected int elementCount;

  ...

  public Vector(Collection<? extends E> c) {
    elementCount = c.size();
    elementData = new Object[(int)Math.min
  ((elementCount*110L)/100, Integer.MAX_VALUE)];
    c.toArray(elementData);
  }
  ...
}
```

Idea behind Data-Centric Synchronization

Synchronization is about preserving data consistency

So why not associate synchronization constraints directly with data?

Terminology

- synchronized block refers to set of locations accessed atomically
- part of a larger operation that preserves consistency

atomic set S

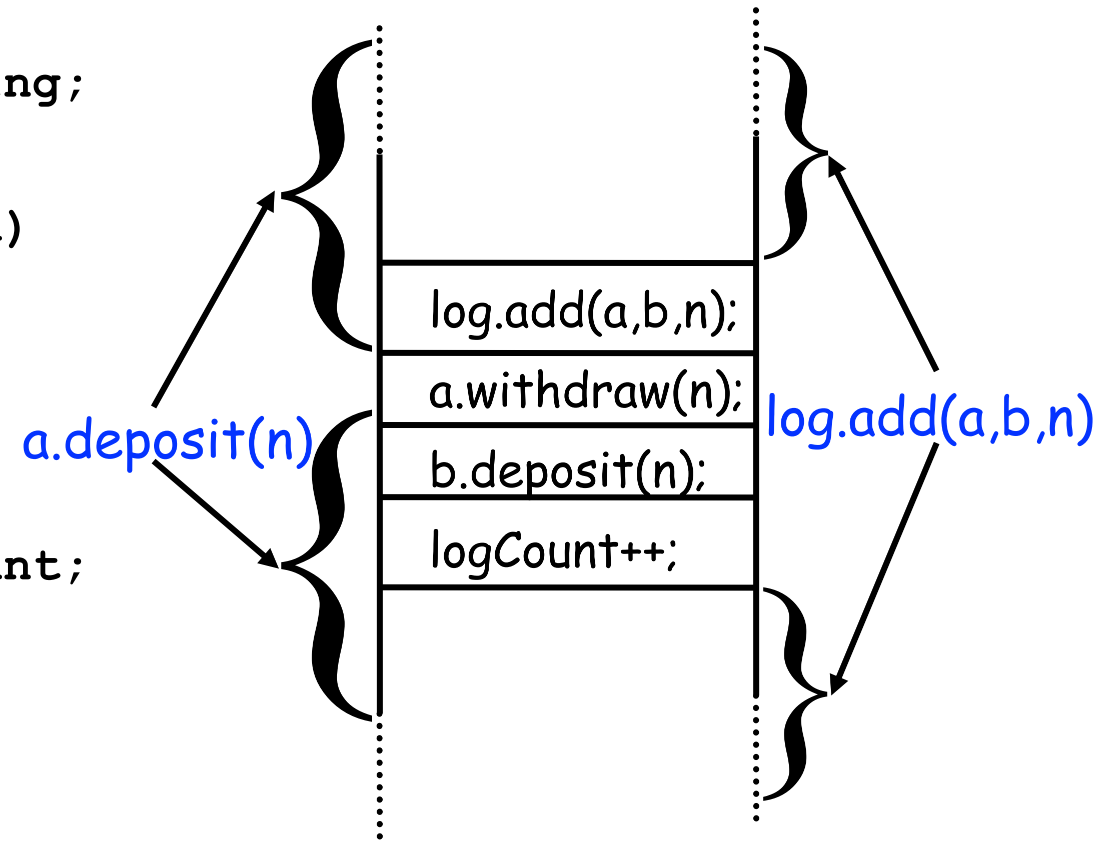
unit of work on S

preserves consistency of S when executed sequentially

```
class BankAccount {
   int checking, savings;
   int transferCount;

   void transfer(int amount) {
    synchronized(this) {
      checking -= amount;
      savings += amount;
    }
    transferCount++;
    …
  }
}
```

Language Construct: Atomic Sets

```
class Account {
  atomicset account;
  atomic(account) int checking;
  public void deposit(int n)
  { … }
  public void withdraw(int n)
  { … }
}

class Bank {
  atomicset logging;
  atomic(logging) Log log;
  atomic(logging) int logCount;

  void transfer(Account a,
                Account b,
                int n){
    log.add(a,b,n);
    a.withdraw(n);
    b.deposit(n);
    logCount++;
  }
}
```



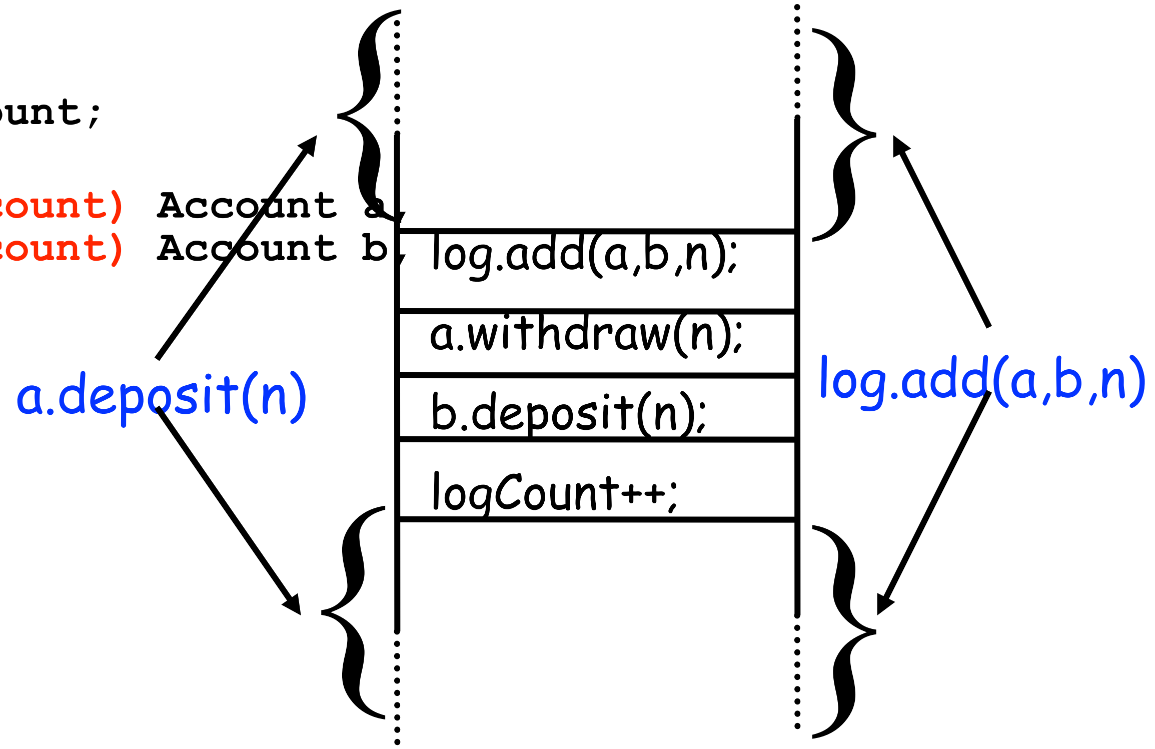
unit of work for this.logging

Language Construct: `unitfor`

```
class Account {
  atomicset account;
  atomic(account) int checking;
  public void deposit(int n){ … }
  public void withdraw(int n){ … }
}

class Bank {
  atomicset logging;
  atomic(logging) Log log;
  atomic(logging) int logCount;

  void transfer(unitfor(account) Account a,
                unitfor(account) Account b,
                int n){
    log.add(a,b,n);
    a.withdraw(n);
    b.deposit(n);
    logCount++;
  }
}
```

Example: unitfor

```
class Account {
  atomicset account;
  atomic(account) int checking;
  public void deposit(int n){ … }
  public void withdraw(int n){ … }
}

class Bank {
  atomicset logging;
  atomic(logging) Log log;
  atomic(logging) int logCount;

  void transfer(unitfor(account) Account a,
                unitfor(account) Account b,
                int n){
    log.add(a,b,n);
    a.withdraw(n);
    b.deposit(n);
    logCount++;
  }
}
```



unit of work for a.account, b.account, this.logging

Preventing the High-Level Data Race in Vector

```
public class Vector<E> {
  // atomicset L inherited from Collection
  atomic(L) protected Object[] elementData;
  atomic(L) protected int elementCount;

  ...

  public Vector(unitfor(L) Collection<? extends E> c) {
    elementCount = c.size();
    elementData = new Object[(int)Math.min((elementCount*110L)/100,
                                          Integer.MAX_VALUE)];
    c.toArray(elementData);
  }

  ...

}
```

Atomic sets that span multiple objects

- **Data structures often rely on other objects for their implementation**
  – The state in these objects belongs to that of the “owning” data structure

- **Examples**
  – LinkedList relies on LinkedList.Entry
  – HashMap relies on HashMap.Entry, a subtype of Map.Entry
  – Most collections rely on helper classes used to define iterators, views, ...

Aliasing and Internal Classes

- **Mechanism for merging atomic sets that span multiple objects**
- **syntax: annotation `|X=this.Y|` on declarations of fields, local variables, allocation sites**
- **Meaning:**
  - Atomic set `X` in the object pointed to by the annotated reference is merged with the atomic set `Y` in the object pointed to by `this`
  - All units of work for `Y` become units of work for `X` as well
- **`internal` classes**
  - must be aliased (i.e. atomic sets must be merged with owner)
  - access restricted to ensure that no reference to an internal class can leak outside of an atomic set
  - permits efficient implementation (no additional synchronization needed, because access is already protected by the class it is aliased with)

Example: Aliasing

the atomic set E in the object pointed to by header is merged with atomic set L in the object pointed to by this

```
class LinkedList extends ... {
  atomic(L) private Entry header|E=this.L|;
}
```

internal type: must be aliased, some restrictions on access, efficient implementation

```
internal class Entry {
  atomicset E;
  atomic(E) Object elem;
  atomic(E) Entry next|E=this.E|;
  atomic(E) Entry prev|E=this.E|;
}
```

the atomic sets E in the objects pointed to by next/prev are merged with atomic set E in the object pointed to by this

Example: simplified version of java.util.LinkedList

- **abstract class AbsList**
  - declares size(), iterator(), add(Object), addAll(AbsList), get(int)
- **class LinkedList**
  - concrete implementation of AbsList
- **class Entry**
  - helper class for representing list-entries
- **interface ListIterator**
  - declares methods hasNext(), next(), hasPrev(), prev(), set()
- **class ListItr**
  - helper class that provides an implementation of ListIterator
- **class Client**
  - small multi-threaded client of LinkedList

Example, continued

```
class LinkedList extends AbsList {
  atomic(L) private Entry header|E=this.L|;

  public LinkedList() {
    header = new Entry|E=this.L|(null, null, null);
    header.next = header.prev = header;
  }

  ...

  public ListIterator iterator() {
    return new ListItr|I=this.L|(this, this.header, 0);
  }
  public boolean addAll(unitfor(L) AbsList c) {
    boolean modified = false;
    ListIterator e = c.iterator();
    while (e.hasNext()) {
      add(e.next());
      modified = true;
    }
    return modified;
  }
}
```

Example, continued

```
class ListItr implements ListIterator  {
  atomicset I;
  atomic(I) private Entry lastReturned|E=this.I|;
  atomic(I) private Entry next|E=this.I|;
  atomic(I) private int nextIndex;
  atomic(I) final LinkedList list|L=this.I|;
  atomic(I) final Entry header|E=this.I|;

  ListItr(LinkedList l|L=this.I|, Entry h|E=this.I|, int index) {
    list = l;
    header = h;
    lastReturned = header;
    if (index < 0 || index > list.size())
      throw new IndexOutOfBoundsException();
    next = header.next;
    for (nextIndex = 0; nextIndex < index; nextIndex++)
      next = next.next;
  }

  // hasNext(), next(), hasPrev(), prev()

  public void set(Object o) {
    if (lastReturned == header)
      throw new IllegalStateException();
    lastReturned.elem = o;
  }
}
```

Example, continued

```
public class Client {
  public static void main(String[] args) throws Throwable {
    final AbsList x = new LinkedList();
    final AbsList y = new LinkedList(); y.add("a"); y.add("a");
    final AbsList z = new LinkedList(); z.add("b"); z.add("b");
    Thread t1 = new Thread(){
      public void run(){ x.addAll(y); }
    };
    Thread t2 = new Thread(){
      public void run(){ x.addAll(z); }
    };
    t1.start(); t2.start();
    t1.join(); t2.join();
    ListIterator it;
    for (it = x.iterator(); it.hasNext();){
      Object o = it.next();
      if (o.equals("b")) it.set("c");
    }
    // can print aacc or ccaa, but not acac, caca, caac, acca
    for ( ; it.hasPrev();)
      System.err.println(it.prev());
  }
}
```

Type System (Details in Technical Report)

- **Types are of the form: C or C|a=this.b|**
  - object of type **C** is in charge of its own synchronization
  - object of type **C|a=this.b|** has its synchronization managed by atomic set **this.b**

- **internal types**
  - must have alias information
  - internal object with alias **|a=this.b|** must be accessed from within a unit of work for **b**
  - (can be implemented efficiently, because synchronization is already ensured by owner)

- **Type rules ensure consistency of alias annotations and encapsulation**

- **Key properties establish atomic-set serializability:**
  - Units of work on each atomic set must be serializable.
  - Since units of work on S are mutually exclusive, and no access to S happens outside of a unit of work on S, all units of work on S happen serially.

Formalization

- **Give dynamic semantics as small-step operational semantics, including heaps and threads**
- **Define configuration as a heap and a series of threads**
  - each thread has a thread ID and a stack
- **Define well-formedness conditions on configurations**
  - configuration is well-formed if its heap and threads are well-formed
  - heap is well-formed if, for every object, the values of its fields are subtypes of the declared types of those fields
  - thread is well-formed if all the frames in its stack are well-typed
- **Prove type soundness (progress and preservation)**
- **Prove that internal objects with alias |a=this.b| are only accessed from within a unit of work for b**
- **Prove atomic-set-serializability**

Implementation

- **source-to-source translator implemented using Eclipse refactoring infrastructure**
  - atomic set annotations entered as Java comments
  - implementation handles Java subset (no generics, inner classes)
    - incl. annotations for arrays (e.g. `|this.M[]F=this.M|`)
  - two translation options supported:
    1. based on java.util.concurrent.locks.ReentrantLock
    2. based on lock ordering & standard synchronized mechanism
- **translation involves**
  - create lock field `$lock_S` in any class C such that atomic set S is present
  - for each class with a lock field `$lock_S`, generate methods `takeLockForS()`, `tryLockForS()`, and `releaseLockForS()`
  - transform object allocations to set locks
  - transform units of work to acquire all needed locks
- **doing a build invokes the translator on the current project**
  - translated sources saved in a separate project in the workspace

AbsList Translated

```
abstract class AbsList implements atomicsets.Atomic {
  protected Lock $lock_L;

  public final void takeLockForL() {
    if ($lock_L != null)
      $lock_L.lock();
  }
  public final boolean tryLockForL() {
    return $lock_L == null || $lock_L.tryLock();
  }
  public final void releaseLockForL() {
    if ($lock_L != null)
      $lock_L.unlock();
  }
  public final Lock getLockForL() {
    return $lock_L;
  }
  public AbsList setLockForL(Lock l) {
    $lock_L = l; return this;
  }
  /*atomic(L)*/ int size = 0;


  public int size(){
    try {
      this.takeLockForL();
      { return size; }
    } finally {
      this.releaseLockForL();
    }
  }
}
```

generated lock field for atomic set L

generated methods for manipulating the lock associated with atomic set L

translated size() method

LinkedList.iterator() Translated

```
class LinkedList extends AbsList {
  ...
  public ListIterator iterator() {
    try {
      this.takeLockForL();
      {
        return (ListIterator)
            new ListItr(this, this.header,0).setLockForI(this.getLockForL());
      }
    } finally {
      this.releaseLockForL();
    }
  }
}
```

translated allocation (aliased)

LinkedList.addAll() Translated

```
  public boolean addAll(/*unitfor(L)*/ AbsList c) {
    boolean $repeat = true;
    do {
      try {
        this.takeLockForL();
        if (c.tryLockForL())
          try {
            $repeat = false;
            {
              boolean modified = false;
              ListIterator e = c.iterator();
              while (e.hasNext()) {
                add(e.next());
                modified = true;
              }
              return modified;
            }
          } finally {
            c.releaseLockForL();
          }
      } finally {
        this.releaseLockForL();
      }
    } while ($repeat);
    throw new Error();
  }
}
```

atomically obtain locks for multiple atomic sets

Client Translated

```
public class Client {
  public static void main(String[] args) throws Throwable {
    final AbsList x = new LinkedList().setLockForL(new ReentrantLock());
    final AbsList y = new LinkedList().setLockForL(new ReentrantLock());
    y.add("a"); y.add("a");
    final AbsList z = new LinkedList().setLockForL(new ReentrantLock());
    z.add("b"); z.add("b");
    Thread t1 = new Thread(){
      public void run(){ x.addAll(y); }
    };
    Thread t2 = new Thread(){
      public void run(){ x.addAll(z); }
    };
    t1.start(); t2.start();
    t1.join(); t2.join();
    ListIterator it;
    for (it = x.iterator(); it.hasNext();){
      Object o = it.next();
      if (o.equals("b")) it.set("c");
    }
    // can print aacc or ccaa, but not acac, caca, caac, acca
    for ( ; it.hasPrev();)
      System.err.println(it.prev());
  }
}
```

object allocation, not aliased

Experiment #1: Java Collections Framework

- **selected several classes from Java Collections Framework**
  - ArrayList, LinkedList, HashMap, HashSet, TreeMap, LinkedHashMap, LinkedHashSet
  - along with any types in java.util on which they depend
  - 63 types, 10,860 LOC
- **manually refactored these classes to eliminate generics and nested classes**
  - not yet supported by our implementation
- **then introduced atomic sets**
  - one atomic set for each of 5 subhierarchies, which includes all instance fields
  - unitfor annotations on “bulk” methods such as addAll()
  - alias annotations to relate entries, iterators, views, etc. to their “owner”
  - only one class could be made internal (LinkedList.Entry)

Experiment #1: Java Collections Framework

- **selected several classes from Java Collections Framework**
  - ArrayList, LinkedList, HashMap, HashSet, TreeMap, LinkedHashMap, LinkedHashSet
  - along with any types in java.util on which they depend
  - 63 types, 10,860 LOC
- **manually refactored these classes to eliminate generics and nested classes**
  - not yet supported by our implementation
- **then introduced atomic sets**
  - one atomic set for each of 5 subhierarchies, which includes all instance fields
  - unitfor annotations on “bulk” methods such as addAll()
  - alias annotations to relate entries, iterators, views, etc. to their “owner”
  - only one class could be made internal (LinkedList.Entry)

Annotation Overhead for classes from JCF

annotation type	number of annotations
atomicset	0
atomic class	5
atomic	0
unitfor	55
alias	330
array object	24
array element	16
TOTAL	430

- **~1 annotation per 25 lines of code**

Experiment #2: SPECjbb2005

- **widely used multi-threaded performance benchmark**
- **simulates companies, warehouses, customers, orders, etc.; performs a series of 4-minute runs with increasing number of warehouses**
- **has a built-in performance mechanism for measuring throughput (transactions/second)**
- **uses both `synchronized` and `wait()/notify()`**
- **7,891 LOC**

Refactoring SPECjbb2005: Some Observations

- **We analyzed the shared state & synchronization in SPECjbb, and introduced atomic sets in a way that ensures that all access to shared state is properly synchronized.**

- **synchronization seems inconsistent in several places**
  - some shared fields, e.g.,Customer.creditLimit have synchronized accessors, but others, e.g., Customer.address have unsynchronized accessors
  - some methods, e.g. TreeMapDataStorage.deleteFirstEntities() should logically be executed atomically, but there is no synchronization to enforce this
- **redundant synchronization**
  - e.g., of fields that are written only once during execution of the constructor and that could be made final
- **wait() and notify() used to implement barriers that coordindate the threads of the multiple warehouses.**
  - wait()/notify() must be used with care when atomic sets are introduced.
- **ownership issues: some classes in SPECjbb rely on collections to store data**
  - e.g., TreeMapDataStorage relies on a TreeMap to store its data
  - achieved by aliasing their atomic sets

Annotation Overhead for SPECjbb2005

annotation type	number of annotations
atomicset	1
atomic class	14
atomic	25
unitfor	0
alias	8
array object	0
array element	1
TOTAL	49

- **these annotations replace 125 occurrences of `synchronized` in the original code**
- **25 occurrences of `synchronized` remain that are related to the use of wait()/notify()**
- **~1 annotation per 160 lines of code**

SPECjbb2005: Performance Experiments

	1	2	3	4	5	6	7	8
original	9654	13341	13566	12854	12914	12875	13951	14063
naive	5588	8068	8074	8030	8106	8205	8159	8248
tuned	6295	8394	8498	8767	8945	9380	9277	9403

- **tuned version: refactored code to make fields final, so our compiler can avoid inserting synchronization**
- **bottom line:**
  - naive implementation achieves approx. 60% of performance of original
  - tuned implementation achieves approx. 67% of performance of original

Related Work (not a complete list)

- **previous work on atomic sets**
  - language proposal, correctness criterion [Vaziri et al. POPL'06]
  - dynamic detection of a-set serializability violations [Hammer et al. ICSE'08]
  - static detection of a-set serializability violations [Kidd et al. VMCAI'09]
- **data groups: abstract representation of groups of fields for modular reasoning**
  - Leino et al. [OOPSLA'98, PLDI'02]
- **type systems for ensuring race-freedom and atomicity**
  - Flanagan, Freund et al. [ESOP'99,PLDI'00,PLDI'03,TOPLAS'08,...]
- **ownership type systems**
  - Noble, Vitek et al. [ECOOP'98, OOPSLA'98, OOPSLA'99], Banerjee, Naumann [POPL'02], Clarke, Drossopoulou [OOPSLA'02], ...
  - unlike traditional owner-as-dominator type systems, there is no single access point
- **lock inference for atomic sections**
  - Cherem et al. [PLDI'08], McCloskey et al. [POPL'06]

Conclusions

- **type-based data-centric approach to synchronization**
  (our original approach required whole-program analysis)
  – correctness property: a type-correct program is atomic-set serializable
  – enables separate compilation
  – handles linked data structures naturally; expressive enough for real code

- **annotation overhead similar to that of previous type systems that guarantee race-freedom or atomicity**
  – but those previous type systems were in addition to existing synchronization

- **preliminary performance experiments indicate reasonable performance**
  – tuned version SPECjbb achieves 67% of throughput of the original code
  – many opportunities for optimization

Future Work

- **improve implementation**
  - improve performance of generated code (e.g., use ReadWriteLocks)
  - automatically infer annotations of most internal types
  - investigate lock-free implementation
- **design analysis to detect possible deadlock**
- **extend type system with effects on methods for better code generation**
  - e.g., “reads this.L”, “writes this.L”

BACKUP

Example: Low-Level Data Race

```
public class Example extends Thread {
   private static int cnt = 0; // shared state
   public void run(){
     int y = cnt;
     cnt = y + 1;
   }

   public static void main(String[] args){
     Thread t1 = new Example();
     Thread t2 = new Example();
     t1.start();
     t2.start();
   }
}
```

Example: Low-Level Data Race

```
private static int cnt = 1; // shared state
```

thread #1

```
public void run(){
      int y = cnt;      y=0
      cnt = y + 1;
}
```

thread #2

```
public void run(){
      int y = cnt;      y=0
      cnt = y + 1;
}
```

Example: High-Level Data Race in java.util.Vector

```
public class Vector<E> {
  protected Object[] elementData;      ⟶ 3
  protected int elementCount;          ⟶ [null,null,null]

  public Vector(Collection<? extends E> c) {
    elementCount = c.size();
    elementData = new Object[(int)Math.min((elementCount*110L)/100,
                                            Integer.MAX_VALUE)];
    c.toArray(elementData);
  }
}
```

```
                    c ⟶ []
```

thread #1

```
elemCount = c.size();
elemData = new Object[ …elemCount…];
c.toArray(elemData);
```

thread #2

```
c.clear();
```

Type System

- **Subtyping:**

$$\frac{C <: D}{C|a=this.b| <: D|a=this.b|}$$

- **Rules for object creation**
  - An internal object must be created with alias information
  - 'C has a' means class C defines or inherits atomic set a

(T-New-Raw)

$$\frac{E(x)=C, \quad C \text{ not internal}}{E \vdash x = \text{new } C()}$$

(T-New-Aset)

$$\frac{E(x)= C|a=this.b|,\ C \text{ has } a, \quad E(this) \text{ has } b}{E \vdash x = \text{new } C|a=this.b|()}$$

Type System (2)

- **Rules for casts**
  - Casts are explicit in the formal language
  - Alias information of non-internal objects can be erased

(T-Cast-Plain)

$$\frac{E(x)=D, \quad E(y)=C, \quad D <: C}{E \; d \; y = (C)\; x}$$

(T-Cast-Aset)

$$\frac{\begin{array}{c} E(x)=D|a=this.b|, \quad E(y)=C|a=this.b| \\ C \text{ has } a, \;\; E(this) \text{ has } b, \;\; D <: C \end{array}}{E \; d \; y = (C|a=this.b|)\; x}$$

(T-Cast-Off)

$$\frac{E(x)=C|a=this.b|, \quad C \text{ not internal}, \quad E(y)=C}{E \; d \; y = (C)\; x}$$

Type System (3)

- **Rule for method call**
  - adapt is a predicate used to unify alias annotations, to “adapt” an alias annotation to the context of the receiver
  - When adapt is undefined, type rules fail, preventing internal objects from being used in a context in which their alias annotation is not well-defined.

$$\text{adapt}(C,t) = C$$

$$\text{adapt}(C|a=\text{this}.b|,\ D|b=\text{this}.c|) = C|a=\text{this}.c|$$

(T-Call)

$$\frac{E(y)=t_y,\ \text{typeof}(t_y.m)=\bar{t} \rightarrow t,\ E(\bar{z})=\bar{t}_z \quad \bar{t}_z <: \text{adapt}(\bar{t},\ t_y),\ t' = \text{adapt}(t,\ t_y),\ E(x)= t'}{E\ d\ x = y.m(\bar{z})}$$

Type System (4)

- **Rules for field selection & update**

(T-Select)

$$\frac{E(x)=t_x,\ E(y)=t_y,\quad \mathrm{typeof}(t_y.f)=t \qquad t_x <: \mathrm{adapt}(t,\ t_y),\quad (t_y.f \text{ is atomic} \Rightarrow y = \mathrm{this})}{E \text{ d } x = y.f}$$

(T-Update)

$$\frac{E(x)=t_x,\ E(y)=t_y,\quad \mathrm{typeof}(t_x.f)=t \qquad t_y <: \mathrm{adapt}(t,\ t_x),\quad (t_x.f \text{ is atomic} \Rightarrow x = \mathrm{this})}{E \text{ d } x.f = y}$$

Example: Aliasing

```
abstract class AbsList {
  atomicset L;
  atomic(L) int size = 0;
  public int size(){ return size; }
  public abstract ListIterator iterator();
  public abstract void add(Object o);
  public abstract boolean addAll(unitfor(L) AbsList c);
  public abstract Object get(int i);
}
class LinkedList extends AbsList {
  atomic(L) private Entry header|E=this.L|;
  public LinkedList(){
    header = new Entry|E=this.L|(null, null, null);
    header.next = header.prev = header;
  }
  public void add(Object o){
    Entry newEntry|E=this.L| = new Entry|E=this.L|(o, header, header.prev);
    newEntry.prev.next = newEntry;
    newEntry.next.prev = newEntry;
    size++;
  }
}
internal class Entry {
  atomicset E;
  atomic(E) Object elem;
  atomic(E) Entry next|E=this.E|;
  atomic(E) Entry prev|E=this.E|;
}
```

the atomic set E in the object pointed to by header is merged with atomic set L in the object pointed to by this

internal type: must be aliased, some restrictions, efficient implementation

the atomic sets E in the objects pointed to by next/prev are merged with atomic set E in the object pointed to by this