

UMassAmherst

*Provably Correct  
Abstract Concurrency Control,  
and More*

Eliot Moss and Trek Palmer

*Architecture and Language Implementation (ALI) Lab*  
*{moss,trekp}@cs.umass.edu*



# Motivation: Open nesting

Open nesting can increase concurrency but ...

- It requires abstract concurrency control
  - Some people fear this is too hard to program
- It requires proper compensating actions
  - Less fear, perhaps, but still some concern



# Proposal: Use Specs & Tools

- Describe abstractions and op conflicts
  - We propose a suitable specification language
- Prove correct the abstract conflict specs
  - Automated proof that non-conflicting operations commute
- Possibly: derive conflict predicates from specification via heuristic search
  
- *Note:* This is not proof of implementation!



# *Further Proposal: Undos Likewise*

- Describe designated compensations for ops
- Prove them correct
  - Very similar to ACC proofs of commutativity
- Possibly: derive undos specification via heuristic search over expressions built using the abstract API
- Will not describe further in this talk



# Specification Language

- Looks a lot like Java (since that's the target)
- Describes *abstract models*
- *Finite* spaces of values and objects
  - Like real languages, but also for tractability
- No unbounded loops, no recursion
  - But other features reduce need
- Heap modeled with structured arrays
  - Formally, uninterpreted functions
- Interpretations for easily computed ops
  - Think: simple boolean logic circuits



# Specification Language 2

- Uninterpreted functions for complex ops
  - $x == y \rightarrow f(x) == f(y)$
  - Can add *simple* axioms, such as commutativity, properties of special values ( $x * 1 == x$ )
- Bulk conditional update
  - forall (T x; pred(...x...)) { a[x] = expr(...x...); }
- If-then-else and bounded for loops
- Return, throw, and catch statements
- Reductions over Abelian groups
  - Won't discuss further here



# Simple example: Set<T>

```
Model Set<T> {
```

```
    boolean in[T]; // in[x] is true iff x is in the Set
```

```
    // the language allows these large (but finite) arrays
```

```
    Set<T>() { forall (Object o) { in[o] = false; } }
```

```
    boolean add (T x) {
```

```
        boolean result = !in[x];  in[x] = true;  return result; }
```

```
    boolean remove (T y) {
```

```
        boolean result = in[y];  in[y] = false;  return result; }
```

```
    boolean contains (T z) { return in[z]; }
```

```
}
```

Extends Java in that T may be a primitive type

The arrays indexed by objects are a very useful extension



# Simple example: Set<T>

```
Model Set<T> {  
    boolean in[T]; // in[x] is true iff x is in the Set  
    // the language allows these large (but finite) arrays  
    Set<T>() { forall (Object o) { in[o] = false; } }  
    boolean add (T x) {  
        boolean result = !in[x]; in[x] = true; return result; }  
    boolean remove (T y) {  
        boolean result = in[y]; in[y] = false; return result; }  
    boolean contains (T z) { return in[z]; }  
}
```

Extends Java in that T may be a primitive type

The arrays indexed by objects are a very useful extension



# Simple example: Set<T>

```
Model Set<T> {  
    boolean in[T]; // in[x] is true iff x is in the Set  
    // the language allows these large (but finite) arrays  
    Set<T>() { forall (Object o) { in[o] = false; } }  
    boolean add (T x) {  
        boolean result = !in[x]; in[x] = true; return result; }  
    boolean remove (T y) {  
        boolean result = in[y]; in[y] = false; return result; }  
    boolean contains (T z) { return in[z]; }  
}
```

Extends Java in that T may be a primitive type

The arrays indexed by objects are a very useful extension



# Simple example: Set<T>

```
Model Set<T> {  
    boolean in[T]; // in[x] is true iff x is in the Set  
    // the language allows these large (but finite) arrays  
    Set<T>() { forall (Object o) { in[o] = false; } }  
    boolean add (T x) {  
        boolean result = !in[x]; in[x] = true; return result; }  
    boolean remove (T y) {  
        boolean result = in[y]; in[y] = false; return result; }   
    boolean contains (T z) { return in[z]; }  
}
```

Extends Java in that T may be a primitive type

The arrays indexed by objects are a very useful extension



# Simple example: Set<T>

```
Model Set<T> {  
    boolean in[T]; // in[x] is true iff x is in the Set  
    // the language allows these large (but finite) arrays  
    Set<T>() { forall (Object o) { in[o] = false; } }  
    boolean add (T x) {  
        boolean result = !in[x]; in[x] = true; return result; }  
    boolean remove (T y) {  
        boolean result = in[y]; in[y] = false; return result; }  
    boolean contains (T z) { return in[z]; }  
}
```

Extends Java in that T may be a primitive type

The arrays indexed by objects are a very useful extension



# Sample proof condition

- Designated conflict predicate for  $[\text{add}(x); \text{remove}(y)]$  is  $[x == y]$
- $\sigma_{ar} = S[\text{add}(x); \text{remove}(y)] \sigma_{init}$
- $\sigma_{ra} = S[\text{remove}(y); \text{add}(x)] \sigma_{init}$
- $S : \text{Stmt} \rightarrow \Sigma \rightarrow \Sigma, \quad \Sigma : \text{Var} \rightarrow \text{Expr}$

## Proof condition (sketch):

- $! \exists \sigma_{init}, x, y: (\sigma_{ar} \neq \sigma_{ra}) \ \& \ !(x == y)$ 
  - i.e., results differ, but supposedly no conflict
  - Note:  $\sigma_{ar}$  and  $\sigma_{ra}$  include results and designated undos (not shown)



# *Automating this proof*

- The precise correctness condition
- Developing expressions in terms of the initial state: eliminates state update to give a pure boolean expression
- Replacing arrays with scalars: manageable size
- Convert boolean expression to conjunctive normal form (CNF) and apply satisfiability (SAT) solver



# Details of the proof condition

- For each variable and array, develop an *expression* for it in  $\sigma_{ar}$  in terms of  $x, y$ , and all the values in  $\sigma_{init}$ ; likewise for  $\sigma_{ra}$

## In particular:

- For each variable  $v$  develop the expression for  $\sigma_{ar}[[v]] \neq \sigma_{ra}[[v]]$
- For each  $k$ -ary array  $A$  develop expr for  $(\exists i_1, \dots, i_k)(\sigma_{ar}[[A[i_1, \dots, i_k]]] \neq \sigma_{ra}[[A[i_1, \dots, i_k]]])$
- Assemble a big expression  $E$ , the *or* of these
- The proof condition is  $\neg E$



# *Proof condition for the example*

$$\neg (\exists i) (\sigma_{ar}[[in[i]]] \neq \sigma_{ra}[[in[i]]])$$

Would have additional terms to handle results of add/remove and designated undos



# Exprs in terms of initial state

- Assignments to scalars:
  - $( S [[ v = e ] ] \sigma ) [[ v ] ] = E [[ e ] ] \sigma$  (value of  $e$  in  $\sigma$ )
- Assignments to arrays produce a conditional expression:  
 $( S [[ A[e_1] = e_2 ] ] \sigma ) [[ A ] ] =$   
 $\lambda i. ( i == E [[ e_1 ] ] \sigma \rightarrow E [[ e_2 ] ] \sigma, \sigma [[ A ] ] (i) )$
- If-then-else and forall statements produce similar conditional expressions
- Unroll bounded loops to straight-line code
- Return, throw, etc., use a control “token”



# For the example

$$\begin{aligned}\sigma_{ar}[[\text{in}[i]]] &= \\ &(\mathcal{S} [[\text{in}[x] = \text{true}; \text{in}[y] = \text{false}]] \sigma_{init})([\text{in}])(i) \\ &= (i == y_{init} \rightarrow \text{false}, \\ &\quad (i == x_{init} \rightarrow \text{true}, \text{in}_{init}(i)))\end{aligned}$$

Omitting some work, and using  $\text{init}$  subscript for values in  $\sigma_{init}$

Similarly for  $\sigma_{ra}$



# Turning array refs into scalars

- Our big expression !E is a large boolean circuit with  $x, y$ , and vars/arrays of  $\sigma_{init}$  as inputs, plus the existentially quantified indices  $i_1, \dots, i_k$  (multiple groups of them)
- It may have many conditionals because of array assignments and if-them-else stmts
- Arrays are very large ***but*** we care only about elements of arrays that are *mentioned in the expression*
- Working bottom up, we rewrite uses of arrays to scalars



# Scalarization details

- First use of  $A$ ,  $A[e_1]$ , we rewrite to scalar  $A_1$
- Second use,  $A[e_2]$ , we rewrite to:  
 $\lambda i. (i == e_1 \rightarrow A_1, A_2)$
- Third use,  $A[e_3]$ , we rewrite to:  
 $\lambda i. (i == e_1 \rightarrow A_1,$   
 $(i == e_2 \rightarrow A_2, A_3))$
- And so on ...

We call this process scalarization.



# Scalarizing the example

$((i == y_{init} \rightarrow \text{false}, (i == x_{init} \rightarrow \text{true}, \text{in}_{init}(i))) \neq$   
 $(i == x_{init} \rightarrow \text{true}, (i == y_{init} \rightarrow \text{false}, \text{in}_{init}(i)))) \&$   
 $x_{init} \neq y_{init}$

First use changed to  $\text{in}_1$ , second to

$(i == i) \rightarrow \text{in}_1, \text{in}_2$ , which is equivalent to  $\text{in}_1$

$\text{in}_1$  and  $\text{in}_2$  can take any value of their type



# Applying a SAT solver

- The final scalarized expression is a boolean expression in terms of *scalar variables*
- The proof condition has the form  $\exists(x_1, \dots)(\text{boolean expr})$
- We write  $\exists(x_1, \dots)(\text{boolean expr})$  in CNF (conjunctive normal form) and pass it to a standard satisfiability (SAT) solver
- The answer should be UNSAT
- If SAT, then satisfying assignment gives a concrete counterexample



# Preliminary Results

	Clauses	Literals	Time(ms)
Good (add/remove)	3962	1460	377
Bad (conflicts = "false")	3174	1208	18
Tuned (made array indices small)	2382	891	87

Time is the estimated processor time as reported by sat4j



# Deriving conflict conditions

- Can start with conflict predicates of *false*
- Collect a number of distinct SAT counterexamples
- Apply *heuristic predicate induction* to obtain a possible predicate
- If too weak, continue; if strong enough done
- Can also try to weaken overly strong predicates - slightly different proof condition: conflict asserted *but* operations commute



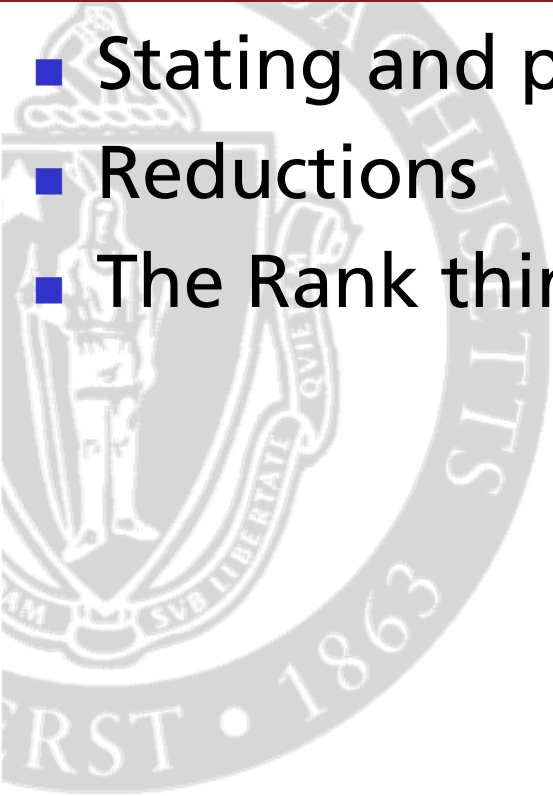
# *Some comments on complexity*

- Our method code is short and straight-line
- It *can* expand exponentially as we build expressions describing conditionals and updates
- The level of abstraction should keep things shorter and tidier:  $2^k$  is not bad if  $k$  is small
- Writing these specs can serve as clear documentation of what library classes do!



# *Things I can talk about offline*

- Stating and proving abstract invariants
- Reductions
- The Rank thing, hash codes, and the like



# Conclusion

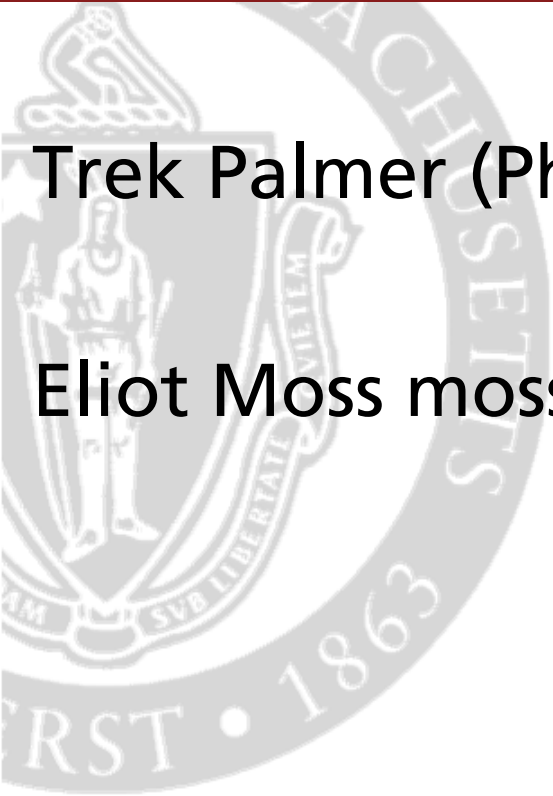
- We have overcome one of the primary voiced objections to open nesting
- We have proved some real examples with an automated technique ... and it is fast



# Contact information again

Trek Palmer (PhD student) [trekp@cs.umass.edu](mailto:trekp@cs.umass.edu)

Eliot Moss [moss@cs.umass.edu](mailto:moss@cs.umass.edu)



# Harder Example: OrderedSet<T>

Model OrderedSet<T> {

**boolean in[T]; // in[x] is true iff x is in the OS**

T prev[T]; // next lower T in the OS; has value for all instance of T

T next[T]; // next higher T in the OS; has value for all instances of T

OS<T>() { forall (T x) { in[x] = false; prev[x] = null; next[x] = null; } }

boolean add (T x) {

if (in[x]) return false;

forall (T z; (Rank[prev[x]] <= Rank[z]) & (Rank[z] < Rank[x])) {

next[z] = x; }

forall (T z; (Rank[x] < Rank[z] & (Rank[z] <= Rank[next[x]])) {

prev[z] = x;

return result; }

T next (T x) { return next[x]; }

boolean contains (T z) { return in[z]; } }

Conflict for add(x);z=next(y) is Rank[y] < Rank[x] & Rank[x] < Rank[z]

(roughly; need to handle nulls, etc.)



# Harder Example: OrderedSet<T>

```
Model OrderedSet<T> {
  boolean in[T]; // in[x] is true iff x is in the OS
  T prev[T]; // next lower T in the OS; has value for all instance of T
  T next[T]; // next higher T in the OS; has value for all instances of T
  OS<T>() { forall (T x) { in[x] = false; prev[x] = null; next[x] = null; } }
  boolean add (T x) {
    if (in[x]) return false;
    forall (T z; (Rank[prev[x]] <= Rank[z]) & (Rank[z] < Rank[x])) {
      next[z] = x; }
    forall (T z; (Rank[x] < Rank[z] & (Rank[z] <= Rank[next[x]])) {
      prev[z] = x;
    }
    return result; }
  T next (T x) { return next[x]; }
  boolean contains (T z) { return in[z]; } }
Conflict for add(x);z=next(y) is Rank[y] < Rank[x] & Rank[x] < Rank[z]
(roughly; need to handle nulls, etc.)
```



# Harder Example: OrderedSet<T>

```
Model OrderedSet<T> {
  boolean in[T]; // in[x] is true iff x is in the OS
  T prev[T]; // next lower T in the OS; has value for all instance of T
  T next[T]; // next higher T in the OS; has value for all instances of T
  OS<T>() { forall (T x) { in[x] = false; prev[x] = null; next[x] = null; } }
  boolean add (T x) {
    if (in[x]) return false;
    forall (T z; (Rank[prev[x]] <= Rank[z]) & (Rank[z] < Rank[x])) {
      next[z] = x; }
    forall (T z; (Rank[x] < Rank[z] & (Rank[z] <= Rank[next[x]])) {
      prev[z] = x;
    }
    return result; }
  T next (T x) { return next[x]; }
  boolean contains (T z) { return in[z]; } }
Conflict for add(x);z=next(y) is Rank[y] < Rank[x] & Rank[x] < Rank[z]
(roughly; need to handle nulls, etc.)
```



# Harder Example: OrderedSet<T>

```
Model OrderedSet<T> {
  boolean in[T]; // in[x] is true iff x is in the OS
  T prev[T]; // next lower T in the OS; has value for all instance of T
  T next[T]; // next higher T in the OS; has value for all instances of T
  OS<T>() { forall (T x) { in[x] = false; prev[x] = null; next[x] = null; } }
  boolean add (T x) {
    if (in[x]) return false;
    forall (T z; (Rank[prev[x]] <= Rank[z]) & (Rank[z] < Rank[x])) {
      next[z] = x; }
    forall (T z; (Rank[x] < Rank[z] & (Rank[z] <= Rank[next[x]])) {
      prev[z] = x;
    return result; }
  T next (T x) { return next[x]; }
  boolean contains (T z) { return in[z]; } }
Conflict for add(x);z=next(y) is Rank[y] < Rank[x] & Rank[x] < Rank[z]
(roughly; need to handle nulls, etc.)
```



# Harder Example: OrderedSet<T>

```
Model OrderedSet<T> {
  boolean in[T]; // in[x] is true iff x is in the OS
  T prev[T]; // next lower T in the OS; has value for all instance of T
  T next[T]; // next higher T in the OS; has value for all instances of T
  OS<T>() { forall (T x) { in[x] = false; prev[x] = null; next[x] = null; } }
  boolean add (T x) {
    if (in[x]) return false;
    forall (T z; (Rank[prev[x]] <= Rank[z]) & (Rank[z] < Rank[x])) {
      next[z] = x; }
    forall (T z; (Rank[x] < Rank[z] & (Rank[z] <= Rank[next[x]])) {
      prev[z] = x;
    }
    return result; }
  T next (T x) { return next[x]; }
  boolean contains (T z) { return in[z]; } }
Conflict for add(x);z=next(y) is Rank[y] < Rank[x] & Rank[x] < Rank[z]
(roughly; need to handle nulls, etc.)
```



# Harder Example: OrderedSet<T>

```
Model OrderedSet<T> {
  boolean in[T]; // in[x] is true iff x is in the OS
  T prev[T]; // next lower T in the OS; has value for all instance of T
  T next[T]; // next higher T in the OS; has value for all instances of T
  OS<T>() { forall (T x) { in[x] = false; prev[x] = null; next[x] = null; } }
  boolean add (T x) {
    if (in[x]) return false;
    forall (T z; (Rank[prev[x]] <= Rank[z]) & (Rank[z] < Rank[x])) {
      next[z] = x; }
    forall (T z; (Rank[x] < Rank[z] & (Rank[z] <= Rank[next[x]])) {
      prev[z] = x;
    }
    return result; }
  T next (T x) { return next[x]; }
  boolean contains (T z) { return in[z]; } }
  Conflict for add(x);z=next(y) is Rank[y] < Rank[x] & Rank[x] < Rank[z]
  (roughly; need to handle nulls, etc.)
```



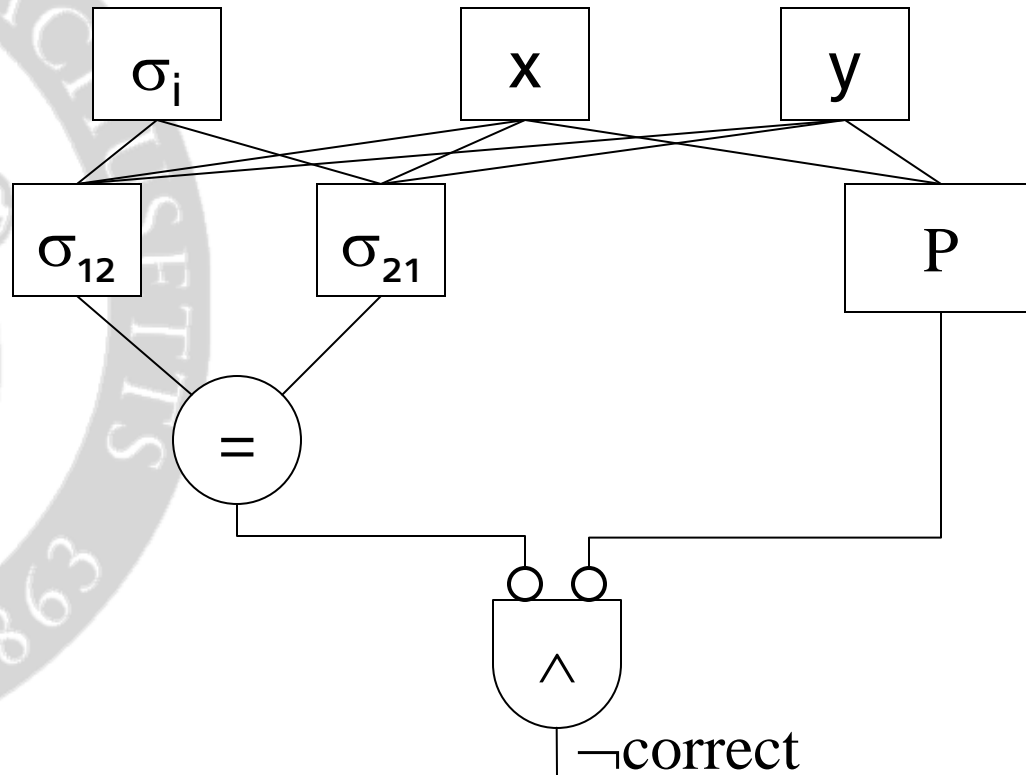
# Harder Example: OrderedSet<T>

```
Model OrderedSet<T> {
  boolean in[T]; // in[x] is true iff x is in the OS
  T prev[T]; // next lower T in the OS; has value for all instance of T
  T next[T]; // next higher T in the OS; has value for all instances of T
  OS<T>() { forall (T x) { in[x] = false; prev[x] = null; next[x] = null; } }
  boolean add (T x) {
    if (in[x]) return false;
    forall (T z; (Rank[prev[x]] <= Rank[z]) & (Rank[z] < Rank[x])) {
      next[z] = x; }
    forall (T z; (Rank[x] < Rank[z] & (Rank[z] <= Rank[next[x]])) {
      prev[z] = x;
    }
    return result; }
  T next (T x) { return next[x]; }
  boolean contains (T z) { return in[z]; } }
```

**Conflict for add(x);z=next(y) is Rank[y] < Rank[x] & Rank[x] < Rank[z]**  
(roughly; need to handle nulls, etc.)



# Circuit Form



- Now the question is: how to build circuits for  $\sigma_{12}$  and  $\sigma_{21}$

