

## Local Reasoning about Storable Locks and Threads

Josh Berdine Microsoft Research

Joint work with Alexey Gotsman (Cambridge), Byron Cook (MSR), Noam Rinetzky and Mooly Sagiv (Tel Aviv)

### Shared variable concurrent programs

- Dijkstra
  - Programs should be insensitive to relative execution speeds
- Brinch Hansen / Hoare
  - Shared variables should be encapsulated and their access controlled
  - Monitors
    - Compiler could check if encapsulation violated for variables
    - Solo operating system written almost entirely with safe primitives
    - But what about the heap? Needed for multi-user OSes
- Owicki & Gries / Jones
  - Limit interference through shared state with predicates / relations
- O'Hearn
  - Concurrent separation logic: encapsulation checking for the heap
    - "Size" of shared state can change
    - "Topology" of access control still fixed

Microsoft<sup>®</sup>

### Locks in the heap — Why?

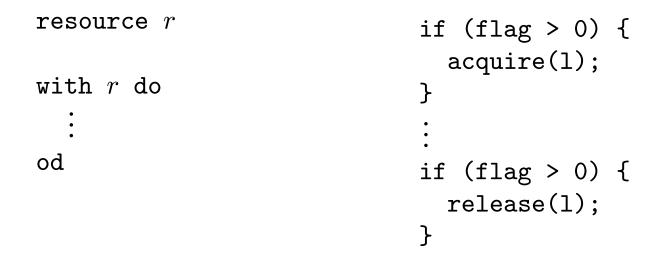
#### typedef struct NODE { typedef struct NODE { int Val; LOCK Lock; struct NODE\* Next; int Val: } NODE; struct NODE\* Next; } NODE; non-empty, last LOCK lock; value is $+\infty$ NODE\* head; NODE\* head; locate coarse(int e) { locate hand over hand(int e) { NODE \*prev, \*curr; NODE \*prev, \*curr; acquire(lock); prev = head; prev = head; acquire(prev); curr = prev->Next; curr = prev->Next; while (curr->Val < e) {</pre> acquire(curr); while (curr->Val < e) {</pre> prev = curr; curr = prev->Next; release(prev); } prev = curr; return (prev, curr); curr = prev->Next; } acquire(curr); } return (prev, curr); }

Microsoft<sup>®</sup>



Optimistic / Idealistic

(More) Realistic



- syntactically determined critical
   regions
  - semantically determined critical regions

# Locks on the stack vs locks in the heap



#### Optimistic / Idealistic

resource rwith r do : od (More) Realistic

- l = new LOCK;init(1); acquire(1); release(1); finalize(l); delete 1;
- bounded numbers of resources
   unbounded numbers of locks

Parallel composition vs of the dynamic thread creation

Optimistic / Idealistic

(while b do  $(P_1 \parallel P_2)) \parallel P_3$ 

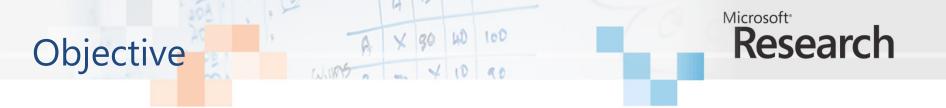
Research

(More) Realistic

```
for (i = 0; i < n; i++) {
   t[i] = fork(proc, i);
}

for (i = 0; i < n; i++) {
   join(t[i]);
}</pre>
```

- bounded numbers of processes
- unbounded numbers of threads



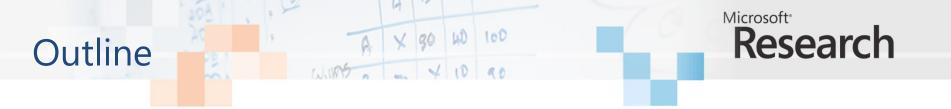
- Program logics for analysis and verification of multithreaded heapmanipulating programs
- Goal: ease static access control
  - Allow unboundedly-many locks and threads
  - That live in the heap (to exploit indirection)
- but also aim to:
  - Retain local reasoning
  - Enable automation in program analysis
  - Treat more realistic programming language constructs



- Logic for storable locks and threads
  - Local reasoning preserved
  - Storable locks as resources
- Not only technical difficulties:
  - Storable locks "make theoreticians wince" (Richard Bornat)
  - Russell's paradox is lurking nearby:

heaps  $\rightarrow$  locks  $\rightarrow$  resource invariants  $\rightarrow$  heaps

- Analogous to stored procedures: Landin's "knots in the store"



- First one top-level parallel composition:  $C_1 \parallel \cdots \parallel C_n$
- Then dynamic thread creation
- Simplification: no shared mutable variables
  - shared mutable heap
  - global pre-initialized constants
  - local variables of threads
- General cases and details: Technical report MSR-TR-2007-39

Concurrent separation logic [O'Hearn04]

- A Floyd/Hoare-style program logic
- Assertion language: \* splits the state into disjoint parts
- Proof system:

$$\frac{\{P\} \ C \ \{Q\}}{\{P*R\} \ C \ \{Q*R\}} \qquad \frac{\{P_1\} \ C_1 \ \{Q_1\} \ \{P_2\} \ C_2 \ \{Q_2\}}{\{P_1*P_2\} \ C_1 \| C_2 \ \{Q_1*Q_2\}}$$

- Allows for local reasoning
- Processes access shared resources
- Synchronization via conditional critical regions:

with r when b do C

Microsoft<sup>®</sup>





- Program state partitioned into (disjoint) substates owned by the different processes and locks
- Processes may access only parts of the state that they own
- Process interaction mediated using resource invariants
- Key in achieving local reasoning:
  - reasoning about each process in isolation
  - using the sequential semantics

#### Reasoning informally

# Research

```
locate coarse(int e) {
  NODE *prev, *curr;
  acquire(lock);
  "have (exclusive access to) head list"
  prev = head;
  "head has a Next"
  curr = prev->Next;
  "curr has a Val"
  while (curr->Val < e) {</pre>
    prev = curr;
    "curr has a Next"
    curr = prev->Next;
  }
  return (prev, curr);
}
```

Need to know this even without owning curr node: So ownership of a node comes with knowledge that the Next node has a Lock

```
locate_hand_over_hand(int e) {
  NODE *prev, *curr;
  prev = head;
  acquire(prev);
  "have (exclusive access to) prev node"
  curr = prev->Next;
  "curr has a Lock"
  acquire(curr);
  "have curr node"
  while (curr->Val < e) {</pre>
    "prev is locked by this thread"
    release(prev);
    "don't have prev node any more"
    prev = curr;
    curr = prev->Next;
    "curr has a Lock"
    acquire(curr);
    "have curr node"
  return (prev, curr);
}
```

X 30 WD

100

# Approach

- Lock  $\rightarrow$  resource invariant
  - lock  $\rightarrow$  sort  $A(\cdot, \cdot)$
  - sort  $A(\cdot, \cdot) \rightarrow$  resource invariant  $I_A(\cdot, \cdot)$
  - first parameter address of the lock
- Example:

struct R {
 LOCK Lock;
 int Data;
};

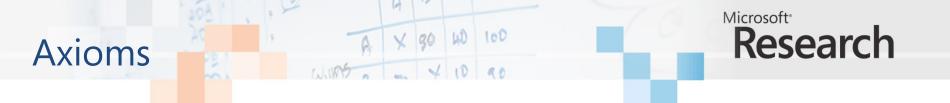
$$I_R(l,v) \stackrel{\Delta}{=} l: Data \mapsto v$$

• Knots in the store cut by indirection through  $A(\cdot, \cdot)$ 

# Assertion language

- Handles:  $A(E, \vec{F})$ 
  - ensures that the lock at the address E exists and has the sort A and parameters  $\vec{F}$
  - gives permission to acquire the lock
  - can be split among threads:
    - $1A(E, \vec{F}) = \frac{1}{2}A(E, \vec{F}) * \frac{1}{2}A(E, \vec{F})$
    - < 1 can acquire the lock
    - = 1 can finalize the lock
- Locked-facts: Locked<sub>A</sub> $(E, \vec{F})$ 
  - lock E is held by the thread owning  $Locked_A(E, \vec{F})$
  - ensures the existence of the lock

Microsoft<sup>®</sup>



#### ${E\mapsto}_{A,\vec{F}}$ (E) ${A(E,\vec{F}) * \mathsf{Locked}_A(E,\vec{F})}$

#### $\{A(E, \vec{F}) * \mathsf{Locked}_A(E, \vec{F})\} \text{ finalize}(E) \{E \mapsto \}$

#### $\{ \mathsf{Locked}_A(E, \vec{F}) * I_A(E, \vec{F}) \} \text{ release}(E) \{ \mathsf{emp}_h \}$

 $\{\pi A(E,\vec{F})\}$  acquire(E)  $\{\pi A(E,\vec{F}) * \text{Locked}_A(E,\vec{F}) * I_A(E,\vec{F})\}$ 

# A simple example A × 90 100

struct R { LOCK Lock; int Data; } \*x; //  $I_R(l) \stackrel{\Delta}{=} l: Data \mapsto \_$ 

```
initialize() {
   \{emp_h\}
   x = new R;
   \{x \mapsto x: Data \mapsto \}
   \operatorname{init}_{R}(\mathbf{x});
   \{x: Data \mapsto R(x)\}
     * Locked<sub>R</sub>(x)}
   x \rightarrow Data = 0;
   {x:Data \mapsto 0 * R(x)}
     * Locked<sub>R</sub>(x)}
   release(x);
   \{R(x)\}
```

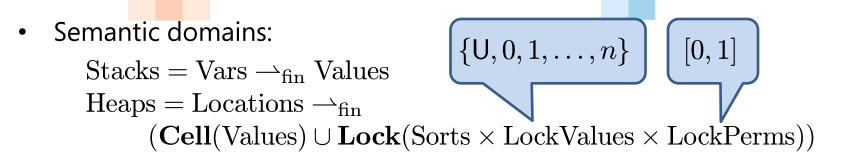
}

```
thread() {
    {\frac{1}{2}R(x)}
    acquire(x);
    {x:Data\mapsto_-*\frac{1}{2}R(x)
    * Locked<sub>R</sub>(x)}
    x->Data++;
    {x:Data\mapsto_-*\frac{1}{2}R(x)
    * Locked<sub>R</sub>(x)}
    release(x);
    {\frac{1}{2}R(x)}
}
```

cleanup() {
 {R(x)}
 acquire(x);
 { $x:Data\mapsto\_*R(x)$  \* Locked<sub>R</sub>(x)}
 finalize(x);
 { $x\mapsto\_*x:Data\mapsto\_$ }
 delete x;
 {emp<sub>h</sub>}
}

Microsoft<sup>®</sup>

### Assertion language model



- each program proof associates each sort with an invariant:  $I_A(\vec{E}) : \text{Sorts} \to \text{Values}^+ \to \mathcal{P}(\text{Stacks} \times \text{Heaps})$
- Satisfaction relation :  $(s, h) \vDash_k \Phi$

$$\begin{array}{ll} (s,h) \models_k E \mapsto F & \Leftrightarrow & h = [\llbracket E \rrbracket_s : \mathbf{Cell}(\llbracket F \rrbracket_s)] \\ (s,h) \models_k \pi A(E) & \Leftrightarrow & h = [\llbracket E \rrbracket_s : \mathbf{Lock}(A, \mathsf{U}, \llbracket \pi \rrbracket_s)] \land \llbracket \pi \rrbracket_s > 0 \\ (s,h) \models_k \mathbf{Locked}_A(E) & \Leftrightarrow & h = [\llbracket E \rrbracket_s : \mathbf{Lock}(A,k,0)] \end{array}$$

\* adds up permissions for locks and their values:

$$U * k = k$$
,  $U * U = U$ ,  $k * j$  undefined

Microsoft<sup>®</sup>

### Semantics of programs

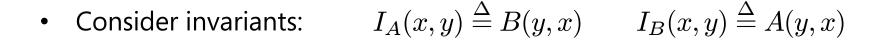
- $\mathsf{pc} \in \{1, ..., n\} \rightarrow \mathsf{ProgPoint}$
- $F \subseteq \operatorname{ProgPoint} \times \operatorname{Command} \times \operatorname{ProgPoint}$
- $\rightarrow_{\mathcal{S}}$  is the least relation satisfying:

$$\frac{(v, C, v') \in F \quad k \in \{1, \dots, n\} \quad C, (s, h) \rightsquigarrow_k q}{\mathsf{pc}[k:v], (s, h) \rightarrow_S \mathsf{pc}[k:v'], q}$$

100

Microsoft<sup>®</sup>

### Flies in the ointment × 90 100



• with code:  $\{x \mapsto \_ * y \mapsto \_\}$ init<sub>A,y</sub>(x);
init<sub>B,x</sub>(y);  $\{A(x, y) * \mathsf{Locked}_A(x, y) * B(y, x) * \mathsf{Locked}_B(y, x)\}$ release(x);  $\{A(x, y) * \mathsf{Locked}_B(y, x)\}$ release(y);  $\{\mathsf{emp}_h\}$ 

- Postcondition has forgotten that locks x and y exist!
- Logic may not detect a memory leak
- Formulating soundness becomes non-trivial

Microsoft<sup>®</sup>

### Soundness (cheating version)

- Usual interleaving-based operational semantics
- Program  $C_1 \parallel \cdots \parallel C_n$
- $\vdash \{P_k\} C_k \{Q_k\}$
- Resource invariants are precise
  - Unambiguously pick out an area of the heap

Theorem:  
If 
$$\sigma_0 \in \begin{pmatrix} n \\ \circledast \\ k=1 \end{pmatrix}^k P_k \end{pmatrix}^k$$
 ( $\circledast$ {invariants for free locks in  $\sigma_0$ }),  
then the program is "safe"  
and  $\sigma_{\mathbf{f}} \in \begin{pmatrix} n \\ \circledast \\ k=1 \end{pmatrix}^k \otimes (\circledast$ {invariants for free locks in  $\sigma_{\mathbf{f}}$ })

• Cheat: statement about  $\sigma_0/\sigma_f$  uses information about free locks in  $\sigma_0/\sigma_f$ 

Microsoft<sup>®</sup>

#### Closure

• How can we find all free locks allocated in a state from a set p?

× 20

- Take  $\sigma \in p$
- Conjoin to  $\sigma$  resource invariants for all locks with value U in  $\sigma$
- and set the value of these locks to 0
- Do the same for every state obtained in this way...
- Definition:

The resulting states without locks with value U form the closure of p:  $\langle p \rangle$ 

- Example:  $\langle R(x) \rangle$  where  $I_R(l) = (l: Data \rightarrow -)$
- Example:  $\langle B(y,x) \rangle$  where  $I_B(x,y) = A(y,x)$  and  $I_A(x,y) = B(y,x)$
- Are we guaranteed to add invariants for all free locks in this way?
- No! Due to self-contained sets of locks

Microsoft<sup>®</sup>

### Admissibility of resource invariants



- Admissibility disallows self-contained sets of locks
- If resource invariants are admissible, closure finds all free locks
- Definition:

Resource invariants for lock sorts  $\mathcal{L}$  re admissible if there do not exist:

- a non-empty set L of lock sorts from  $\mathcal{L}$  with parameters
- a state  $\sigma \in \circledast$ {invariants for all locks in *L*}

such that the permission associated with the every lock from L in  $\sigma$  is 1

• Examples:

$$- \{I_R(l) \stackrel{\Delta}{=} l: Data \mapsto_{-}\} \text{ is admissible}$$
$$- \{I_A(x, y) \stackrel{\Delta}{=} B(y, x), I_B(x, y) \stackrel{\Delta}{=} A(y, x)\} \text{ is not}$$

#### Soundness



- Program  $C_1 \parallel \cdots \parallel C_n$
- $\vdash \{P_k\} C_k \{Q_k\}$
- Resource invariants are precise
- Theorem:

Suppose that

- either resource invariants are admissible
- or one of  $Q_k$  is intuitionistic (does not notice heap extension)

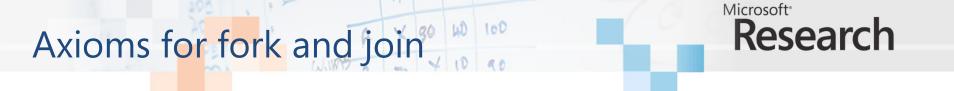
If 
$$\sigma_0 \in \left\langle \stackrel{n}{\underset{k=1}{\circledast}} \llbracket P_k \rrbracket^k \right\rangle$$
, then the program is "safe"  
and  $\sigma_{\mathbf{f}} \in \left\langle \stackrel{n}{\underset{k=1}{\circledast}} \llbracket Q_k \rrbracket^k \right\rangle$ 

Microsoft<sup>®</sup>

# Dynamic thread creation



- Programs: let  $f_1() = C_1, \ldots, f_n() = C_n$  in C
- Two new commands: x = fork(f) and join(E)
- Assertion language: thread handles  $tid_f(E)$ 
  - thread running f with identifier E exists
  - gives permission to join it
  - only one thread can join any given thread
- Satisfaction relation:  $(s, h, t) \vDash_k \Phi$ 
  - t-thread pool



• Need to give up the precondition of the thread at fork:

 $\Gamma, \{P\} f() \{Q\} \vdash \{P\} \mathtt{x} = \texttt{fork}(f) \{\texttt{emp}_{h} \land \mathsf{tid}_{f}(x)\}$ 

• and receive the postcondition at join:

 $\Gamma, \{P\} f() \{Q\} \vdash \{\mathsf{emp}_{\mathbf{h}} \land \mathsf{tid}_{f}(E)\} \texttt{join}(E) \{Q\}$ 

where  $fv(\{P,Q\}) \subseteq GlobalConsts$ 

• Other axioms adjusted accordingly

# Soundness A x 80 100 Microsoft Research

- Proof of the program let  $f_1() = C_1, \ldots, f_n() = C_n$  in C :
  - $$\begin{split} \Gamma &\vdash \{P_1\} \ C_1 \ \{Q_1\} \\ \vdots & & \text{where} \\ \Gamma &\vdash \{P_n\} \ C_n \ \{Q_n\} & & \Gamma &= \{P_1\} \ f() \ \{Q_1\}, \dots, \{P_n\} \ f() \ \{Q_n\} \\ \Gamma &\vdash \{P\} \ C \ \{Q\} \end{split}$$
- Technical issues:
  - Soundness conditions:
    - $P_k$  are precise
    - $P_k$  and  $Q_k$  have an empty lockset (no lock in a state satisfying them has a value other than U)
  - Same circularity problem as with locks:  $\operatorname{tid}_f \to Q_f \to \operatorname{tid}_f$
  - Admissibility, closure, and soundness can be generalized





- Original concurrent separation logic can reason about storable locks:
  - represent them as cells storing the identifier of the thread owning the lock
  - build a global invariant of memory as a whole
- Drawbacks:
  - lots of auxiliary state  $\Rightarrow$  horrible proofs
  - reasoning is not modular
  - automation is infeasible

### Compared to RGSep [Vafeiadis+07]



- RGSep Combination of Jones' rely-guarantee and separation logic
  - Locks not treated natively
  - Uses rely-guarantee to simplify reasoning about the global invariant
  - (+) Reasoning about complex fined-grained concurrency algorithms
  - (–) Awkward reasoning about programs that allocate and deallocate many simple data structures
- One fancy pre-allocated data structure vs many dynamically allocated simpler ones
- We'd like both at once

#### Summary



- Proposed a Floyd/Hoare-style program logic for
  - concurrent, heap-manipulating programs that:
  - allows local reasoning about unboundedly-many storable locks and threads
    - i.e., more realistic concurrent programming primitives

× 20

- is strong enough to prove some examples published as challenges
  - piece of multicasting code
  - lock-coupling list operations
- is set up to found a program analysis
  - thread-local fixed-point semantics is an analysis scheme
- is sound via a reasonably lightweight mechanism for cutting recursive knots in the heap
  - using only a simple semantics
- Want a semantic analysis of admissibility of resource invariants