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

Microsoft Research

- 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

Locks in the heap — Why?

```
typedef struct NODE {
  int Val;
  struct NODE* Next;
} NODE;

LOCK lock;
NODE* head;

locate_coarse(int e) {
  NODE *prev, *curr;
  acquire(lock);
  prev = head;
  curr = prev->Next;
  while (curr->Val < e) {
    prev = curr;
    curr = prev->Next;
  }
  return (prev, curr);
}
```

non-empty, last value is $+\infty$

```
typedef struct NODE {
  LOCK Lock;
  int Val;
  struct NODE* Next;
} NODE;

NODE* head;

locate_hand_over_hand(int e) {
  NODE *prev, *curr;
  prev = head;
  acquire(prev);
  curr = prev->Next;
  acquire(curr);
  while (curr->Val < e) {
    release(prev);
    prev = curr;
    curr = prev->Next;
    acquire(curr);
  }
  return (prev, curr);
}
```

Resources & CCRs vs locks

Optimistic / Idealistic

```
resource r

with r do
   ⋮
od
```

- syntactically determined critical regions

(More) Realistic

```
if (flag > 0) {
  acquire(l);
}
⋮
if (flag > 0) {
  release(l);
}
```

- semantically determined critical regions

Locks on the stack vs locks in the heap

Microsoft Research

Optimistic / Idealistic

```
resource r

with r do
   ⋮
od
```

- bounded numbers of resources

(More) Realistic

```
l = new LOCK;
⋮
init(l);
⋮
acquire(l);
⋮
release(l);
⋮
finalize(l);
⋮
delete l;
```

- unbounded numbers of locks

Parallel composition vs dynamic thread creation

Optimistic / Idealistic

$$(\texttt{while}\ b\ \texttt{do}\ (P_1 \parallel P_2)) \parallel P_3$$

- bounded numbers of processes

(More) Realistic

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

- unbounded numbers of threads

Objective

Microsoft Research

- Program logics for analysis and verification of multithreaded heap-manipulating 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

This talk

Microsoft Research

- 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 → locks → resource invariants → heaps
  - Analogous to stored procedures: Landin’s “knots in the store”

Outline

- 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]

Microsoft Research

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

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

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

$$\texttt{with}\ r\ \texttt{when}\ b\ \texttt{do}\ C$$

to be replaced

Concurrent separation logic

- 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

Microsoft 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) {
    prev = curr;
    "curr has a Next"
    curr = prev->Next;
  }
  return (prev, curr);
}
```

```
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) {
    "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);
}
```

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

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;
};
```

$$I_R(l, v) \triangleq l{:}Data \mapsto v$$

- 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: $\text{Locked}_A(E, \vec{F})$
  - lock E is held by the thread owning $\text{Locked}_A(E, \vec{F})$
  - ensures the existence of the lock

Axioms

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

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

$$\{\mathsf{Locked}_A(E,\vec{F}) * I_A(E,\vec{F})\}\ \texttt{release(}E\texttt{)}\ \{\mathsf{emp}_{\mathbf{h}}\}$$

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

A simple example

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

```
initialize() {
```
$\{\mathsf{emp}_\mathbf{h}\}$
```
  x = new R;
```
$\{x \mapsto \_ * x{:}Data \mapsto \_\}$
```
  init_R(x);
```
$\{x{:}Data \mapsto \_ * R(x) * \mathsf{Locked}_R(x)\}$
```
  x->Data = 0;
```
$\{x{:}Data \mapsto 0 * R(x) * \mathsf{Locked}_R(x)\}$
```
  release(x);
```
$\{R(x)\}$
```
}
```

```
thread() {
```
$\{\frac{1}{2}R(x)\}$
```
  acquire(x);
```
$\{x{:}Data \mapsto \_ * \frac{1}{2}R(x) * \mathsf{Locked}_R(x)\}$
```
  x->Data++;
```
$\{x{:}Data \mapsto \_ * \frac{1}{2}R(x) * \mathsf{Locked}_R(x)\}$
```
  release(x);
```
$\{\frac{1}{2}R(x)\}$
```
}
```

```
cleanup() {
```
$\{R(x)\}$
```
  acquire(x);
```
$\{x{:}Data \mapsto \_ * R(x) * \mathsf{Locked}_R(x)\}$
```
  finalize(x);
```
$\{x \mapsto \_ * x{:}Data \mapsto \_\}$
```
  delete x;
```
$\{\mathsf{emp}_\mathbf{h}\}$
```
}
```

Assertion language model

Microsoft Research

- Semantic domains:

$$\text{Stacks} = \text{Vars} \rightharpoonup_{\text{fin}} \text{Values}$$
$$\text{Heaps} = \text{Locations} \rightharpoonup_{\text{fin}} (\mathbf{Cell}(\text{Values}) \cup \mathbf{Lock}(\text{Sorts} \times \text{LockValues} \times \text{LockPerms}))$$

$\{\mathsf{U}, 0, 1, \ldots, n\}$ (LockValues)

$[0, 1]$ (LockPerms)

  – 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 : $(s, h) \vDash_k \Phi$

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

$*$ adds up permissions for locks and their values:

$$\mathsf{U} * k = k, \ \ \mathsf{U} * \mathsf{U} = \mathsf{U}, \ \ k * j \text{ undefined}$$

Semantics of programs

- pc $\in \{1,\ldots,n\} \to \mathrm{ProgPoint}$
- $F \subseteq \mathrm{ProgPoint} \times \mathrm{Command} \times \mathrm{ProgPoint}$
- $\to_S$ is the least relation satisfying:

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

$$\begin{array}{lll}
x = E, (s[x:(u,1)], h) & \rightsquigarrow_k & (s[x:([\![E]\!]_{s[x:(u,1)]}, 1)], h) \\
x = [E], (s[x:(u,1)], h[e:\mathsf{Cell}(u)]) & \rightsquigarrow_k & (s[x:(u,1)], h[e:\mathsf{Cell}(u)]), e = [\![E]\!]_{s[x:(u,1)]} \\
[E] = F, (s, h[[\![E]\!]_s : \mathsf{Cell}(u)]) & \rightsquigarrow_k & (s, h[[\![E]\!]_s : \mathsf{Cell}([\![F]\!]_s)]) \\
x = \mathbf{new}, (s[x:(u,1)], h) & \rightsquigarrow_k & (s[x:(v,1)], h[v:\mathsf{Cell}(w)]),\ \text{if } h(v)\uparrow \\
\mathbf{delete}\ E, (s, h[[\![E]\!]_s : \mathsf{Cell}(u)]) & \rightsquigarrow_k & (s,h) \\
\mathbf{init}_A(E), (s, h[[\![E]\!]_s : \mathsf{Cell}(u)]) & \rightsquigarrow_k & (s, h[[\![E]\!]_s : \mathsf{Lock}(A,k,1)]) \\
\mathbf{finalize}(E), (s, h[[\![E]\!]_s : \mathsf{Lock}(A,k,1)]) & \rightsquigarrow_k & (s, h[[\![E]\!]_s : \mathsf{Cell}(u)]) \\
\mathbf{assume}(G), (s,h) & \rightsquigarrow_k & (s,h),\ \text{if } [\![G]\!]_s = \mathsf{true} \\
\mathbf{assume}(G), (s,h) & \not\rightsquigarrow_k & \qquad \text{if } [\![G]\!]_s = \mathsf{false} \\
\mathbf{acquire}(E), (s, h[[\![E]\!]_s : \mathsf{Lock}(A,0,\pi)]) & \rightsquigarrow_k & (s, h[[\![E]\!]_s : \mathsf{Lock}(A,k,\pi)]) \\
\mathbf{acquire}(E), (s, h[[\![E]\!]_s : \mathsf{Lock}(A,j,\pi)]) & \not\rightsquigarrow_k & \qquad \text{if } j > 0 \\
\mathbf{release}(E), (s, h[[\![E]\!]_s : \mathsf{Lock}(A,k,\pi)]) & \rightsquigarrow_k & (s, h[[\![E]\!]_s : \mathsf{Lock}(A,0,\pi)])
\end{array}$$

Flies in the ointment

Microsoft Research

- Consider invariants: $I_A(x,y) \triangleq B(y,x)$ $I_B(x,y) \triangleq A(y,x)$

- with code:

$\{x \mapsto \_ * y \mapsto \_\}$

```
init_{A,y}(x);
init_{B,x}(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);
```

$\{\mathsf{emp}_{\mathbf{h}}\}$

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

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

$$[\![\Phi]\!]^k = \{(s,h) : (s,h) \models_k \Phi\}$$

- Theorem:

$$\text{If} \quad \sigma_0 \in \left(\mathop{\circledast}_{k=1}^{n} [\![P_k]\!]^k \right) * (\circledast\{\text{invariants for free locks in } \sigma_0\}),$$
$$\text{then the program is "safe"}$$
$$\text{and} \quad \sigma_{\mathbf{f}} \in \left(\mathop{\circledast}_{k=1}^{n} [\![Q_k]\!]^k \right) * (\circledast\{\text{invariants for free locks in } \sigma_{\mathbf{f}}\})$$

- Cheat: statement about σ_0/σ_f uses information about free locks in σ_0/σ_f

Closure

Microsoft Research

- How can we find all free locks allocated in a state from a set p?
  - Take $\sigma \in p$
  - Conjoin to σ resource invariants for all locks with value U in σ
  - 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 $I_R(l) = (l\text{:Data}\mapsto\text{-})$
- 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

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 $\mathcal{L}$ are 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 σ is 1

- Examples:
  - $\{I_R(l) \triangleq l{:}Data \mapsto \_\}$ is admissible
  - $\{I_A(x,y) \triangleq B(y,x),\ I_B(x,y) \triangleq A(y,x)\}$ is not

Soundness

- Usual interleaving-based operational semantics
- 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 $\sigma_0 \in \left\langle \bigcircledast_{k=1}^{n} [\![P_k]\!]^k \right\rangle$, then the program is "safe"

  and $\sigma_{\mathbf{f}} \in \left\langle \bigcircledast_{k=1}^{n} [\![Q_k]\!]^k \right\rangle$

Dynamic thread creation

- Programs: $\texttt{let}\ f_1() = C_1, \ldots, f_n() = C_n\ \texttt{in}\ C$
- Two new commands: $\texttt{x = fork(}f\texttt{)}$ and $\texttt{join(}E\texttt{)}$
- Assertion language: thread handles $\mathsf{tid}_f(E)$
  - 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

Axioms for fork and join

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

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

- and receive the postcondition at join:

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

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

- Other axioms adjusted accordingly

Soundness

- Proof of the program `let` $f_1() = C_1, \ldots, f_n() = C_n$ `in` C :

$$\Gamma \vdash \{P_1\}\ C_1\ \{Q_1\}$$
$$\vdots$$
$$\Gamma \vdash \{P_n\}\ C_n\ \{Q_n\}$$

where

$$\Gamma = \{P_1\}\ f()\ \{Q_1\}, \ldots, \{P_n\}\ f()\ \{Q_n\}$$

$$\Gamma \vdash \{P\}\ C\ \{Q\}$$

- 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: $\mathsf{tid}_f \to Q_f \to \mathsf{tid}_f$
  - Admissibility, closure, and soundness can be generalized

Compared to concurrent separation logic

- 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 ⇒ 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

Microsoft Research

- 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
  - 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