Separation Logic for a Java-like Language with Reentrant Locks and Fork/Join

Christian Haack*   Marieke Huisman‡   Clément Hurlin‡

*Radboud Universiteit Nijmegen, The Netherlands
‡ INRIA Sophia Antipolis - Méditerranée, France

June 9, 2008
Motivation

Synchronization in Java can be done in 2 ways:

- `synchronized(x){ ... }`
- `x.lock()/x.unlock()` (generalize synchronized blocks).
Motivation

Synchronization in Java can be done in 2 ways:

- `synchronized(x){ ... }`
- `x.lock()/x.unlock()` (generalize synchronized blocks).

Java locks are *reentrant*:

- Acquiring a lock twice is possible.

Contrary to C-threads:

- Acquiring a lock twice results in blocking.
Separation Logic

\( x.f \xrightarrow{\pi} v \) (called “points-to predicate”) has a dual meaning:
- \( x.f \) contains value \( v \).
- Permission \( \pi \) to access field \( x.f \).

\( \pi \) is a fraction in \((0, 1] \):
- 1 is the permission to write access a location.
- Any \( 0 < \pi < 1 \) is the permission to read-only access a location.
Separation Logic

$x.f \xrightarrow{\pi} v$ (called “points-to predicate”) has a dual meaning:
- $x.f$ contains value $v$.
- Permission $\pi$ to access field $x.f$.

$\pi$ is a fraction in $(0, 1]$:
- $1$ is the permission to write access a location.
- Any $0 < \pi < 1$ is the permission to read-only access a location.

Global invariant:
- For each location $x.f$ the sum of permissions to $x.f$ is $\leq 1$.

- Prevents read-write and write-write conflicts (data races).
- Allows concurrent reads.
Separation Logic

\( F \ast G \) is the *separating conjunction*:

- \( F \ast F \) implies \( F \) (weakening).
- But \( F \) does not imply \( F \ast F \).

\( F \rightarrow^* G \) is the *linear implication* (or “*baguette magique*”):

- Reads “consume \( F \) yielding \( G \)” or “trade \( F \) and receive \( G \)”
- \( F \ast (F \rightarrow^* G) \) implies \( G \)
Hoare Rules for Non-Reentrant Locks

\[ \Gamma \vdash \{F\} \ c \ \{G\} \]

- A thread can safely execute command \( c \) with initial resource (permissions) \( F \) and end with resource \( G \).
- Threads own resources and use resources to read/write to the heap.
Hoare Rules for Non-Reentrant Locks

\[ \Gamma \vdash \{F\} \ c \ \{G\} \]

- A thread can safely execute command \( c \) with initial resource (permissions) \( F \) and end with resource \( G \).
- Threads \textit{own} resources and use resources to read/write to the heap.

To deal with non-reentrant locks, O’Hearn proposed to attach \textit{resource invariants} to locks:

\[ I \text{ is } x\text{’s resource invariant} \]

\[ \Gamma \vdash \{F\} \ x\text{.lock()} \ \{F*I\} \]

\(\Downarrow\) Locks also own resources.

\(\Downarrow\) When a lock is acquired, it \textit{lends} its resource invariant to the acquiring thread.
Reentrant Locks for Object Oriented Programs

Resource invariants are described with *abstract predicates* (Parkinson’05):

```java
class C{
    int f;
    pred inv = f \rightarrow _;
}
class D extends C{
    int g;
    extends pred inv by g \rightarrow _;
}
```

Intuition:
- If x’s dynamic type is C, then x.inv is x.f \rightarrow _
- If x’s dynamic type is D, then x.inv is (x.f \rightarrow _ \ast x.g \rightarrow _)
- Resource invariants get stronger in subclasses.
Reentrant Locks for Object Oriented Programs

Resource invariants are described with *abstract predicates* (Parkinson’05):

```java
class C{
    int f;
    pred inv = f \uparrow_1 _;
}

class D extends C{
    int g;
    extends pred inv by g \uparrow_1 _;
}
```

Intuition:

- If x’s dynamic type is C, then x.inv is x.f \uparrow _
- If x’s dynamic type is D, then x.inv is (x.f \uparrow _* x.g \uparrow _)
- Resource invariants get **stronger** in subclasses.

Resource invariants are represented by the distinguished predicate inv:

```java
class Object{
    pred inv = true;
}
```
O’Hearn’s rule does not support reentrant locks:

{true}
x.lock();
{I} (I is x’s resource invariant)
x.lock();
{I ∗ I} ← wrong!
...

Separation Logic for Reentrant Locks

4 formulas to speak about locks (where $S$ is a *multiset*):

$$ F ::= \ldots \mid \text{lockset}(S) \mid S \text{ contains } x \mid x.\text{Initialized} \mid x.\text{Fresh} \mid \ldots $$

- **$F$ linear**: $F$ does not imply $F * F$.
- **$F$ copyable**: $F$ implies $F * F$.

For each thread, we track the set of currently held locks:

- **lockset** $(S)$: $S$ is the multiset of currently held locks. (linear)
- **$S$ contains $x$**: lockset $S$ contains lock $x$. (copyable)
Separation Logic for Reentrant Locks

4 formulas to speak about locks (where $S$ is a multiset):

$$F ::= \ldots \mid \text{lockset}(S) \mid S \text{ contains } x \mid x.\text{Initialized} \mid x.\text{Fresh} \mid \ldots$$

- **$F$ linear**: $F$ does not imply $F \ast F$.
- **$F$ copyable**: $F$ implies $F \ast F$.

For each thread, we track the set of currently held locks:

- **lockset($S$)**: $S$ is the multiset of currently held locks. (linear)
- **$S$ contains $x$**: lockset $S$ contains lock $x$. (copyable)

For each lock, we track its abstract lock state:

- **$x.\text{Fresh}$**: ticket to initialize $x$’s resource invariant. (linear)
- **$x.\text{Initialized}$**: $x$’s resource invariant is initialized. (copyable)
Initializing Locks

\[
C<\pi> <: \Gamma(x) \\
\{\text{true}\} \\
\Gamma \vdash x = \text{new } C<\pi> \\
\{x.\text{init} * C \text{ isclassof } x * \ominus_{\Gamma(u)} <:\text{Object} \land x \neq u * x.\text{Fresh}\}
\]

\[\downarrow\] After creation a lock cannot be acquired: \(x.\text{Initialized}\) misses to match (Lock)’s precondition.

\[
\{\text{lockset}(S) * x.\text{inv} * x.\text{Fresh}\} \\
\Gamma \vdash x.\text{commit} \\
\{\text{lockset}(S) * !(S \text{ contains } x) * x.\text{Initialized}\}
\]

\[\downarrow\] \(x.\text{commit}\) is a no-op.

\[\downarrow\] After being committed a lock can be acquired: (Commit)’s postcondition matches (Lock)’s precondition.
Acquiring Locks

\[
\begin{align*}
\Gamma \vdash \text{lockset}(S) \neq \text{lockset}(S) & \iff S \text{ contains } x \\
\{ \text{lockset}(S) \neq \text{lockset}(S) \} \land \text{lock}(x) & \implies \{ \text{lockset}(x \cdot S) \neq \text{lockset}(x \cdot S) \}
\end{align*}
\]

- Threads obtain resource invariants only when initially acquiring a lock (precondition \((S \text{ contains } x)\)).

- Nothing special to handle subclassing.

\[
\begin{align*}
\Gamma \vdash \text{lockset}(x \cdot S) & \neq \text{lockset}(x \cdot S) \\
\{ \text{lockset}(x \cdot S) \neq \text{lockset}(x \cdot S) \} \land \text{lock}(x) & \implies \{ \text{lockset}(x \cdot S) \neq \text{lockset}(x \cdot S) \}
\end{align*}
\]

- Reentrant acquirement (precondition \(\text{lockset}(x \cdot S)\)): \(x\)'s resource invariant is not obtained.
Releasing Locks

\[
\Gamma \vdash \{\text{lockset}(x \cdot x \cdot S)\}x.\text{unlock()}\{\text{lockset}(x \cdot S)\}
\]  \hspace{1cm} \text{(Re-Unlock)}

⇒ Releasing \( x \) but \( x \)'s reentrancy level \( > 1 \) (precondition \( \text{lockset}(x \cdot x \cdot S) \)): invariant not abandoned.

\[
\Gamma \vdash \{\text{lockset}(x \cdot S) \cdot x.\text{inv}\}x.\text{unlock()}\{\text{lockset}(S)\}
\]  \hspace{1cm} \text{(Unlock)}

⇒ If \( x \)'s reentrancy level is not known to be \( > 1 \), \( x \)'s resource invariant is abandoned.
Reasoning about the Absence of Aliasing

Problem: (Lock)’s precondition requires that $x$ is not in the current lockset.

To establish this precondition, one has to show that $x$ does not alias any member of the current lockset.

Separation logic tries to avoid the need to reason about the absence of aliasing! We sneak some of this back into the program logic.

Our next example shows that we can still deal with fine-grained concurrency in spite of this.
Reasoning about the Absence of Aliasing

Problem: (Lock)’s precondition requires that $x$ is not in the current lockset.

To establish this precondition, one has to show that $x$ does not alias any member of the current lockset.

Separation logic tries to avoid the need to reason about the absence of aliasing! We sneak some of this back into the program logic.

Our next example shows that we can still deal with fine-grained concurrency in spite of this.

- We have classes parametrized by specification values (≡ Java + permissions + locksets).
- We can encode ownership type-system to help us deal with aliasing.
Lock coupling list algorithm:

- Standard test case for fine-grained concurrency (Gotsman et al’07, Parkinson et al’07).
- **No single lock** that guards the entire list.
- List-traversing methods acquire each of the node locks right before accessing the node, and release it again after moving to the next node.

The precondition of traversal methods requires that none of the list nodes is in the current lockset:

```java
class List{
    pred nodes_unlocked<Lockset S> = ???;
    requires nodes_unlocked<S>;
    ensures nodes_unlocked<S>;
    void traverse(){...}
}
```
Lock Coupling: A Solution with Type-based Ownership

```java
class Node<Object owner>{
    public int val;
    public Node<owner> next;

    extends pred inv by (∃ Node<owner> x)(val ⊸ _ * next ⊸ x * x.Initialized)
}

class List{
    Node<this> header;
    extends pred inv by (∃ Node<this> x)(header ⊸ x * x.initialized);
    pred nodes_unlocked<Lockset s> = lockset(S) * (∀ Object owner, Node<owner> x)
    (S contains x -* owner != this);

    requires nodes_unlocked<S>;
    ensures nodes_unlocked<S>;
    void traverse(){...}
}
```

- We universally quantify over a type parameter: we exploit that in our language type system and program logic are inter-dependent.
Multithreading in Java:
- Threads should define the `run()` method.
- When `start()` is called on a thread, the new thread is forked and its `run()` method executes in parallel with the rest of the program.
- A thread `t` is finished when `t.join()` returns.

- We use `run()`’s precondition as `start()`’s precondition.
- We use `run()`’s postcondition as `join()`’s postcondition.
Start/Join

- Parent threads pass resources to newly created threads.
- Alive threads take back resources of terminated threads.

\[ F ::= \ldots \mid \text{Join}(x, \pi) \mid \ldots \]

\[ \Rightarrow \] Ticket to take back fraction \( \pi \) of \( x \)'s resource when \( x \) terminates.

\( \pi \cdot F \)

\[ \Rightarrow \] Formula \( F \) scaled by \( \pi \) (defined as a derived form)

\[
\begin{align*}
\Gamma \vdash x : C & \quad F \text{ is run's postcondition in class } C \\
\{ x \neq \text{null} \ast \text{Join}(x, \pi) \} & \quad x.\text{join()} \{ \pi \cdot F \}
\end{align*}
\] (Join)
Start/Join: Two Examples

Thread $t_1$ takes back $t_0$’s resource.

Threads $t_1$ and $t_2$ both take back half of $t_0$’s resource.
Semantics of Formulas

\( \mathcal{R}; s \models F \)

Resource \( \mathcal{R} \) satisfies \( F \).

Resources are quintets \((h, \mathcal{P}, \mathcal{L}, \mathcal{F}, \mathcal{I})\). Resources are owned by threads.

\( h \) is the part of the heap owned by the thread considered.

\( \mathcal{P} \) is a permission table: it contains permissions to access \( h \) and join threads.

\( \mathcal{L} \) is an abstract lock table: it keeps track of the lockset of the thread considered.

\( \mathcal{F} \) is a fresh set: it is the set of objects that can be initialized by the thread considered.

\( \mathcal{I} \) is an initialized set: it is the global set of initialized objects.
Semantics of Formulas

Semantics of locksets:

\[
[[\text{nil}]]_h^S \triangleq \lambda x.0
\]

\[
[[S \cdot S']]_h^S \triangleq \lambda x. [[S]]_h^S(x) + [[S']]_h^S(x)
\]

\[
(h, P, L, F, I); s \models x.f \xrightarrow{\pi} v \quad \text{iff} \quad [[x.f]]_h^S = v \text{ and } [[\pi]] \leq P([[x]]_h^S, f)
\]

\[
(h, P, L, F, I); s \models \text{lockset}(S) \quad \text{iff} \quad L(t) = [[S]]_h^S \text{ for some } t
\]

\[
(h, P, L, F, I); s \models S \text{ contains } x \quad \text{iff} \quad [[S]]_h^S([[x]]_h^S) > 0
\]

\[
(h, P, L, F, I); s \models x.\text{Initialized} \quad \text{iff} \quad [[x]]_h^S \in I
\]

\[
(h, P, L, F, I); s \models x.\text{Fresh} \quad \text{iff} \quad [[x]]_h^S \in F
\]

\[
(h, P, L, F, I); s \models \text{Join}(x, \pi) \quad \text{iff} \quad [[\pi]] \leq P([[x]]_h^S, \text{join})
\]
Preservation

\[
R = (h, P, abs(l), F, I) \quad R' \cup \emptyset \models \otimes_{o \in ready(R)} o.inv \\
\frac{R \vdash (t_0 \mid \ldots \mid t_n) : \Diamond}{R \# R'}
\]

(State)

\[\downarrow t_0 \mid \ldots \mid t_n \text{ is the program’s thread pool.}\]
\[\quad \bullet \text{The thread pool is verified w.r.t. } R.\]

\[\downarrow h \text{ is the program’s heap.}\]
\[\quad \bullet \text{The program is verified w.r.t. to } h: R = (h, \ldots)\]

\[\downarrow l \text{ is the program’s } \text{concrete lock table}.\]
\[\quad \bullet \text{The program is verified w.r.t. to an abstraction of } l: R = (\ldots, abs(l), \ldots)\]

\[\downarrow R' \text{ is a resource that satisfy the resource invariants of unheld locks (looked up by} \text{ready}(R))\]

\[\downarrow R \# R': R \text{ and } R' \text{ are compatible.}\]
Preservation

\[
\begin{align*}
\mathcal{R} &\vdash \emptyset : \diamond & \text{(Empty Pool)} \\
\mathcal{R} &\vdash t_0 : \diamond & \mathcal{R}' &\vdash (t_1 | \ldots | t_n) : \diamond & \text{(Cons Pool)} \\
\mathcal{R} \cdot \mathcal{R}' &\vdash (t_0 | t_1 | \ldots | t_n) : \diamond \\
\mathcal{R} ; s &\models F & \{F\} c \{G\} & \text{(Thread)} \\
\mathcal{R} &\vdash o \text{ is } (s \text{ in } c) : \diamond
\end{align*}
\]

\( o \text{ is } (s \text{ in } c) \) represents a thread.
  \( o \) : thread identifier
  \( s \) : thread-local stack
  \( c \) : command to execute.
Achievements

- A concurrent separation logic for multithreaded Java.
- Combination of abstract predicates with class axioms and value-parametrized types, to express relations between abstract predicates and dependencies between object interfaces.
- Flexible combination of abstract predicates and fractional permissions, through permission-parametrized predicates.
- Support for read-sharing of `join`’s postcondition (not supported in Gotsman et al.’s work).
- Separation logic rules for re-entrant locks (in progress).
- Soundness proof.
- Challenge examples proven (concurrent iterator, lock coupling algorithm, worker thread).
Future Work

- Generation of proof obligations
- Automatic support for solving proof obligations
- Richer specification language
- Formalization in Coq (some initial work already)?
- Compilation to Concurrent CMinor?
class List{
    public Node<this> header;

    extends pred inv by ( ∃ Node<this> x)(header \xrightarrow{1} x * x.initialized);

    requires lockset(S) * ( ∀ Object owner, Node<owner> x)(S contains x -* owner!=this);
    ensures lockset(S) * ( ∀ Object owner, Node<owner> x)(S contains x -* owner!=this);

    public void delete(int n){
        lock();
        Node<this> current = header;
        current.lock();
        if(current.val==n){
            header=current.next;
            unlock(); current.unlock();
            return;
        }
        unlock();
        Node<this> prev = current;
        current = prev.next;
        current.lock();
        while(current.next != null && current.val != n){
            prev.unlock();
            prev=current;
            current=prev.next;
            current.lock();
        }
        prev.next = current.next;
        prev.unlock();
    }
}