

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



ParSec
Parallelism and Security

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.

← our goal

Contrary to C-threads:

- Acquiring a lock twice results in blocking.

Separation Logic

$x.f \stackrel{\pi}{\mapsto} v$ (called “points-to predicate”) has a dual meaning:

- $x.f$ contains value v .
- Permission π to access field $x.f$.

π 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

$x.f \overset{\pi}{\mapsto} v$ (called “points-to predicate”) has a dual meaning:

- $x.f$ contains value v .
- Permission π to access field $x.f$.

π 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 ≤ 1 .
- ↳ Prevents read-write and write-write conflicts (data races).
- ↳ Allows concurrent reads.

Separation Logic

$F * G$ is the *separating conjunction*:

- $F * F$ implies F (weakening).
- But F **does not** imply $F * F$.

$F - * G$ is the *linear implication* (or “*baguette magique*”):

- Reads “consume F yielding G ” or “trade F and receive G ”
- $F * (F - * G)$ implies G

Hoare Rules for Non-Reentrant Locks

$$\frac{}{\Gamma \vdash \{F\} c \{G\}}$$

- 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

$$\frac{}{\Gamma \vdash \{F\} c \{G\}}$$

- 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.

To deal with non-reentrant locks, O'Hearn proposed to attach *resource invariants* to locks:

$$\frac{I \text{ is } x\text{'s resource invariant}}{\Gamma \vdash \{F\} x.\text{lock}() \{F * I\}}$$

- ↳ Locks also own resources.
- ↳ When a lock is acquired, it *lends* its resource invariant to the acquiring thread.

Reentrant Locks for Object Oriented Programs

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

```
class C{  
  int f;  
  
  pred inv =  $f \mapsto \_;$   
}
```

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

Intuition:

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

Reentrant Locks for Object Oriented Programs

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

```
class C{
    int f;

    pred inv = f  $\mapsto$  ^1 _;
}

class D extends C{
    int g;

    extends pred inv by g  $\mapsto$  ^1 _;
}
```

Intuition:

- If x 's dynamic type is C , then $x.inv$ is $x.f \mapsto ^1 _$
- If x 's dynamic type is D , then $x.inv$ is $(x.f \mapsto ^1 _ * x.g \mapsto ^1 _)$
- Resource invariants get **stronger** in subclasses.

Resource invariants are represented by the distinguished predicate `inv`:

```
class Object{
    pred inv = true;
}
```

Separation Logic for Reentrant Locks

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*):

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

- 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:

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

Separation Logic for Reentrant Locks

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

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

- 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:

- $\text{lockset}(S)$: S is the multiset of currently held locks. (linear)
- $S \text{ contains } x$: 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

$$\frac{C\langle\bar{\pi}\rangle <: \Gamma(x)}{\Gamma \vdash \begin{array}{c} \{\text{true}\} \\ x = \text{new } C\langle\bar{\pi}\rangle \\ \{x.\text{init} * C \text{ isclassof } x * \textcircled{*}_{\Gamma(u)} <: \text{Object } x \neq u * x.\text{Fresh}\} \end{array}} \text{(New)}$$

- ↳ After creation a lock cannot be acquired: $x.\text{Initialized}$ misses to match (Lock)'s precondition.

$$\frac{\Gamma \vdash \begin{array}{c} \{\text{lockset}(S) * x.\text{inv} * x.\text{Fresh}\} \\ x.\text{commit} \\ \{\text{lockset}(S) * !(S \text{ contains } x) * x.\text{Initialized}\} \end{array}}{\text{(Commit)}}$$

- ↳ $x.\text{commit}$ is a **no-op**.
- ↳ After being committed a lock can be acquired: (Commit)'s postcondition matches (Lock)'s precondition.

Acquiring Locks

$$\frac{\Gamma \vdash \{ \text{lockset}(S) * !(S \text{ contains } x) * x.\text{Initialized} \}}{x.\text{lock()} \{ \text{lockset}(x \cdot S) * x.\text{inv} \}} \text{ (Lock)}$$

- ↳ Threads obtain resource invariants only when *initially acquiring* a lock (precondition $!(S \text{ contains } x)$).
- ↳ Nothing special to handle subclassing.

$$\frac{\Gamma \vdash \{ \text{lockset}(x \cdot S) \}}{x.\text{lock()} \{ \text{lockset}(x \cdot x \cdot S) \}} \text{ (Re-Lock)}$$

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

Releasing Locks

$$\frac{}{\Gamma \vdash \{\text{lockset}(x \cdot x \cdot S)\} x.\text{unlock}() \{\text{lockset}(x \cdot S)\}} \text{ (Re-Unlock)}$$

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

$$\frac{}{\Gamma \vdash \{\text{lockset}(x \cdot S) * x.\text{inv}\} x.\text{unlock}() \{\text{lockset}(S)\}} \text{ (Unlock)}$$

- ↳ 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: A Test Case for Fine-grained Concurrency

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:

```
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

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

  extends pred inv by ( $\exists$  Node<owner> x)(val  $\mapsto$  _ * next  $\mapsto$  x * x.Initialized)
}

class List{
  Node<this> header;
  extends pred inv by ( $\exists$  Node<this> x)(header  $\mapsto$  x * x.initialized);
  pred nodes_unlocked<Lockset s> = lockset(S) * ( $\forall$  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.

$F ::= \dots \mid \text{Join}(x, \pi) \mid \dots$

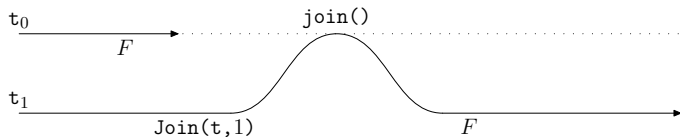
↳ Ticket to take back fraction π of x 's resource when x terminates.

$\pi \cdot F$

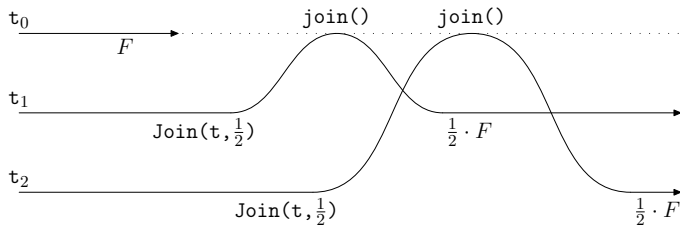
↳ Formula F scaled by π (defined as a derived form)

$$\frac{\Gamma \vdash x : C \quad F \text{ is run's postcondition in class } C}{\{x \neq \text{null} * \text{Join}(x, \pi)\} x.\text{join}() \{\pi \cdot F\}} \text{ (Join)}$$

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:

$$\llbracket \text{nil} \rrbracket_h^s \triangleq \lambda x. 0$$

$$\llbracket S \cdot S' \rrbracket_h^s \triangleq \lambda x. \llbracket S \rrbracket_h^s(x) + \llbracket S' \rrbracket_h^s(x)$$

$(h, \mathcal{P}, \mathcal{L}, \mathcal{F}, \mathcal{I}); s$	$\models x.f \xrightarrow{\pi} v$	iff	$\llbracket x.f \rrbracket_h^s = v$ and $\llbracket \pi \rrbracket \leq \mathcal{P}(\llbracket x \rrbracket_h^s, f)$
$(h, \mathcal{P}, \mathcal{L}, \mathcal{F}, \mathcal{I}); s$	$\models \text{lockset}(S)$	iff	$\mathcal{L}(t) = \llbracket S \rrbracket_h^s$ for some t
$(h, \mathcal{P}, \mathcal{L}, \mathcal{F}, \mathcal{I}); s$	$\models S$ contains x	iff	$\llbracket S \rrbracket_h^s(\llbracket x \rrbracket_h^s) > 0$
$(h, \mathcal{P}, \mathcal{L}, \mathcal{F}, \mathcal{I}); s$	$\models x.\text{Initialized}$	iff	$\llbracket x \rrbracket_h^s \in \mathcal{I}$
$(h, \mathcal{P}, \mathcal{L}, \mathcal{F}, \mathcal{I}); s$	$\models x.\text{Fresh}$	iff	$\llbracket x \rrbracket_h^s \in \mathcal{F}$
$(h, \mathcal{P}, \mathcal{L}, \mathcal{F}, \mathcal{I}); s$	$\models \text{Join}(x, \pi)$	iff	$\llbracket \pi \rrbracket \leq \mathcal{P}(\llbracket x \rrbracket_h^s, \text{join})$

Preservation

$$\frac{\begin{array}{l} \mathcal{R} = (h, \mathcal{P}, \text{abs}(l), \mathcal{F}, \mathcal{I}) \quad \mathcal{R}'; \emptyset \models \bigotimes_{o \in \text{ready}(\mathcal{R})} o.\text{inv} \\ \mathcal{R} \vdash (t_0 \mid \dots \mid t_n) : \diamond \quad \mathcal{R} \# \mathcal{R}' \end{array}}{\langle h, l, t_0 \mid \dots \mid t_n \rangle : \diamond} \text{ (State)}$$

- ↳ $t_0 \mid \dots \mid t_n$ is the program's thread pool.
 - The thread pool is verified w.r.t. \mathcal{R} .
- ↳ h is the program's heap.
 - The program is verified w.r.t. to h : $\mathcal{R} = (h, \dots)$
- ↳ l is the program's *concrete lock table*.
 - The program is verified w.r.t. to an abstraction of l : $\mathcal{R} = (\dots, \text{abs}(l), \dots)$
- ↳ \mathcal{R}' is a resource that satisfy the resource invariants of unheld locks (looked up by $\text{ready}(\mathcal{R})$)
- ↳ $\mathcal{R} \# \mathcal{R}'$: \mathcal{R} and \mathcal{R}' are *compatible*.

Preservation

$$\frac{}{\mathcal{R} \vdash \emptyset : \diamond} \text{ (Empty Pool)} \qquad \frac{\mathcal{R} \vdash t_0 : \diamond \quad \mathcal{R}' \vdash (t_1 \mid \dots \mid t_n) : \diamond}{\mathcal{R} * \mathcal{R}' \vdash (t_0 \mid t_1 \mid \dots \mid t_n) : \diamond} \text{ (Cons Pool)}$$
$$\frac{\mathcal{R}; s \models F \quad \{F\} c \{G\}}{\mathcal{R} \vdash o \text{ is } (s \text{ in } c) : \diamond} \text{ (Thread)}$$

↳ $o \text{ is } (s \text{ in } c)$ 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 ?

Lock Coupling Implementation

```
class List{
  public Node<this> header;

  extends pred inv by ( $\exists$  Node<this> x)(header  $\mapsto$  x * x.initialized);

  requires lockset(S) * ( $\forall$  Object owner, Node<owner> x)(S contains x -* owner!=this);
  ensures lockset(S) * ( $\forall$  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();
  }
}
```