

Separation Logic Contracts for a Java-like Language with Fork/Join

Christian Haack Clément Hurlin

Radboud University of Nijmegen

INRIA Sophia Antipolis

AMAST 2008 in Urbana-Champaign (Illinois) on July 28-31



<http://mobius.inria.fr>

What is this Talk About?

Goals.

- specifying flexible heap access policies for Java programs
- verifying adherence to such policies

Why?

- needed for modular program verification
 - framing
 - controlling thread interference
- to prevent data races
- to prevent errors due to unwanted state modifications

Our technique of choice.

- separation logic

Example: Iterators

```
interface Collection {  
    void add(Object e);  
    Iterator iterator();  
}
```

```
interface Iterator {  
    boolean hasNext();  
    Object next();  
    void remove();  
}
```

Prevent:

- concurrent modifications of collection
- concurrent mutations of collection elements

To this end:

- statically enforce disciplined iterator usage

Outline

- 1 Background on Separation Logic
- 2 Class Axioms
- 3 Abstract Predicates and Subclassing
- 4 Fork/Join
- 5 Conclusion

Background on Separation Logic: Connectives

The points-to predicate.

$$x.f \mapsto v$$

- 1 assertion that $x.f$ contains v
- 2 access ticket for $x.f$

Resource conjunction:

$$F * G$$

- two access tickets
- not idempotent: F does not imply $F * F$
- we allow weakening: $F * F$ implies F
(because Java is garbage-collected)

Background on Separation Logic: Connectives

Resource implication:

$$F \multimap G$$

- ticket to trade ticket F for ticket G
- linear modus ponens: $F * (F \multimap G) \multimap G$
- useful for representing the right to make a state transition:
 - If F and G represent abstract program states, then $F \multimap G$ represents the right to move from F to G .

Background on Separation Logic: Hoare Rules

Writing.

$$\{ x.f \mapsto _ * F \} x.f = v \{ x.f \mapsto v * F \}$$

Reading.

$$\{ x.f \mapsto v * F \} y = x.f \{ x.f \mapsto v * y == v * F \}$$

Concurrent threads.

$$\frac{\{ F \} c \{ G \} \quad \{ F' \} c' \{ G' \}}{\{ F * F' \} c \parallel c' \{ G * G' \}}$$

Caveat: this rule disallows concurrent reads.

Fractional Permissions for Concurrent Reads

Superscripting points-to with fractions.

$$x.f \stackrel{\pi}{\mapsto} v \quad \pi \in \mathbb{Q} \cap (0, 1]$$

The split/merge law.

$$x.f \stackrel{\pi}{\mapsto} v \ * \!-\! * \ (x.f \stackrel{\frac{\pi}{2}}{\mapsto} v \ * \ x.f \stackrel{\frac{\pi}{2}}{\mapsto} v)$$

Permission 1 grants write access:

$$\{ x.f \stackrel{1}{\mapsto} _ \ * \ F \} \ x.f = v \ \{ x.f \stackrel{1}{\mapsto} v \ * \ F \}$$

Any permission grants read access:

$$\{ x.f \stackrel{\pi}{\mapsto} v \ * \ F \} \ y = x.f \ \{ x.f \stackrel{\pi}{\mapsto} v \ * \ y == v \ * \ F \}$$

This allows concurrent reads, while preventing data races.

Method Contracts

- **Preconditions** say what access tickets **callers** must hand to methods.
- **Postconditions** say what access tickets **methods** hand back to callers.

```
class C extends Thread {  
    int f,g;  
  
    //@ requires this.f  $\overset{1}{\mapsto}$  _ * this.g  $\overset{1}{\mapsto}$  _;  
    //@ ensures this.f  $\overset{1}{\mapsto}$  _ * this.g  $\overset{1}{\mapsto}$  _;  
    void run() { this.f = 1; this.g = 2; }  
  
    //@ requires this.f  $\overset{1}{\mapsto}$  _ * this.g  $\overset{1}{\mapsto}$  _;  
    //@ ensures true;  
    void m() { new C().start(); }  
}
```

Abstract Predicates

- Classes can define predicates.

```
class C extends Thread {  
    int f,g;  
  
    //@ pred state = this.f  $\overset{1}{\mapsto}$  _ * this.g  $\overset{1}{\mapsto}$  _;  
  
    //@ requires this.state; ensures this.state;  
    void run() { this.f = 1; this.g = 2; }  
  
    //@ requires this.state; ensures true;  
    void m() { new C().start(); }  
}
```

- Object clients treat predicates abstractly.
- `this` knows the definitions of its own predicates.

Outline

- 1 Background on Separation Logic
- 2 Class Axioms**
- 3 Abstract Predicates and Subclassing
- 4 Fork/Join
- 5 Conclusion

Class Axioms

```
class C {
    ...
    //@ axiom F;
    ...
}

interface I {
    ...
    //@ axiom F;
    ...
}
```

- Class axioms **export facts** about abstract predicates.
- Predicates definitions must make class axioms true.

Example: Split/Mergeable Predicates (Datagroups)

```
class Point {
  int x,y;

  //@ pred state<perm p> = this.x  $\overset{p}{\mapsto}$  _ * this.y  $\overset{p}{\mapsto}$  _;

  //@ axiom state<p> *-* ( state<p/2> * state<p/2> );

  //@ requires this.state<1>; ensures this.state<1>;
  void set(int x, int y) { this.x = x; this.y = y; }

  //@ requires this.state<p>; ensures this.state<p>;
  double distToOrigin() { return Math.sqrt(x*x + y*y); }
}
```

$\text{group } P < \bar{T} \bar{x} >; \triangleq \text{pred } P < \bar{T} \bar{x} >; \text{ axiom } P < \bar{x} > *-* (P < \bar{e} > * P < \bar{e} >);$
where $e_i \triangleq \begin{cases} x_i/2 & \text{if } T_i = \text{perm} \\ x_i & \text{otherwise} \end{cases}$

Example: Nested Datagroups

```
interface Sprite {
  //@ group state<perm p>;
  //@ group position<perm p>;
  //@ group color<perm p>;
  //@ axiom position<p> ispartof state<p>;
  //@ axiom color<p> ispartof state<p>;
  //@ requires position<1>; ensures position<1>;
  void updatePosition();
  //@ requires color<1>; ensures color<1>;
  void updateColor();
  //@ requires state<1>; ensures state<1>;
  void update();
  //@ requires state<p>; ensures state<p>;
  void display();
}
```

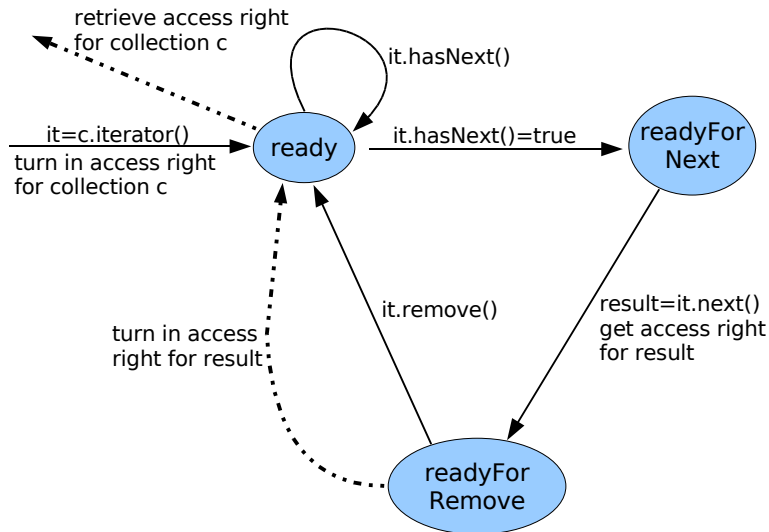
$$F \text{ ispartof } G \stackrel{\Delta}{=} G \text{ -* } (F * (F \text{ -* } G))$$

Example: Recursive and Overlapping Datagroups

```
interface StudentList {  
    //@ group state<perm p>;  
    //@ group ids_and_links<perm p, perm q>;  
    //@ group grades_and_links<perm p, perm q>;  
  
    //@ axiom  
    //@ state<p>  
    //@ *** (ids_and_links<p,p/2> * grades_and_links<p,p/2>);  
  
    //@ requires grades_and_links<1,p> * ids_and_links<q,r>;  
    //@ ensures grades_and_links<1,p> * ids_and_links<q,r>;  
    void updateGrade(int id, int grade);  
}
```

- The datagroups `ids_and_links` and `grades_and_links` overlap on the link fields.

Example: A Usage Protocol for Iterators



Example: Iterator Usage Protocol (Collection)

```
interface Collection {  
    //@ requires this.state<1> * e.state<1>;  
    //@ ensures this.state<1>;  
    void add(Object e);  
  
    //@ requires this.state<p>;  
    //@ ensures result.ready;  
    Iterator/*@<p,this>@*/ iterator();  
}
```

Example: Iterator Usage Protocol (Iterator)

```
interface Iterator/*@<perm p, Collection iteratee>@*/ {
    //@ pred ready;
    //@ pred readyForNext;
    //@ pred readyForRemove<Object element>;

    //@ axiom ready -* iteratee.state<p>;
    //@ axiom readyForRemove<e> * e.state<p> -* ready;

    //@ requires ready;
    //@ ensures ready & (result -* readyForNext);
    boolean hasNext();

    //@ requires readyForNext;
    //@ ensures readyForRemove<result> * result.state<p>;
    Object next();

    //@ requires readyForRemove<_> * p==1;
    //@ ensures ready;
    void remove();
}
```

Outline

- 1 Background on Separation Logic
- 2 Class Axioms
- 3 Abstract Predicates and Subclassing**
- 4 Fork/Join
- 5 Conclusion

Extending Predicates

- Classes can **extend abstract predicates**.
- Predicate extensions in subclasses get ***-conjoined** with predicate extensions in superclasses.

```
class C {  
    int f;  
    //@ pred state<perm p> = this.f  $\overset{P}{\vdash}$  _;  
}
```

```
class D extends C {  
    int g;  
    //@ pred state<perm p> = this.g  $\overset{P}{\vdash}$  _;  
    // total state: this.f  $\overset{P}{\vdash}$  _ * this.g  $\overset{P}{\vdash}$  _  
}
```

Axiomatizing the Stack of Class Frames

- $e.P@C\langle\bar{e}\rangle$ $e.P\langle\bar{e}\rangle$ holds “down to” class C
 $e.P\langle\bar{e}\rangle$ equivalent to $e.P@D\langle\bar{e}\rangle$ where D is e 's dynamic class

Axioms for opening/closing predicates class frame by class frame:

- $e.P@C\langle\bar{e}\rangle$ ispartof $e.P\langle\bar{e}\rangle$
recall that this desugars to
 $e.P\langle\bar{e}\rangle \text{ -* } (e.P@C\langle\bar{e}\rangle \text{ * } (e.P@C\langle\bar{e}\rangle \text{ -* } e.P\langle\bar{e}\rangle))$
- $\text{this}.P@C\langle\bar{e}\rangle \text{ *-} (F \text{ * } \text{this}.P@D\langle\bar{e}\rangle)$
if F is $P@C\langle\bar{e}\rangle$'s definition in C , and $C \preceq_1 D$

Axiom for promoting qualified to unqualified predicates:

- $(e.P@C\langle\bar{e}\rangle \text{ * } C \text{ isclassof } e) \text{ -* } e.P\langle\bar{e}\rangle$
this axiom is typically applied right after object constructors terminate

We also allow arity extensions in subclasses (not shown).

Outline

- 1 Background on Separation Logic
- 2 Class Axioms
- 3 Abstract Predicates and Subclassing
- 4 Fork/Join**
- 5 Conclusion

Join Permissions

Supporting multiple joiners.

- We want to allow several threads to join the same thread.
- This can, for instance, be useful when a worker thread populates a database that several threads read-share after the worker thread is done.

New formula.

$\text{Perm}(e.\text{join}, \pi)$ permission to use fraction π
of $e.\text{join}$'s postcondition

Split/merge axiom.

$$\text{Perm}(e.\text{join}, \pi) \text{ ** } \left(\text{Perm}(e.\text{join}, \frac{\pi}{2}) * \text{Perm}(e.\text{join}, \frac{\pi}{2}) \right)$$

Fork/Join

```
class Thread {  
    //@ pred preRun;  
    //@ group postRun<perm p>;  
  
    //@ requires preRun; ensures postRun<1>;  
    abstract void run();  
  
    final void start();    // native  
    final void join();    // native  
}
```

Forking.

$$\{ v \neq \text{null} * v.\text{preRun} \} v.\text{start}() \{ \text{true} \}$$

Joining.

$$\{ v \neq \text{null} * \text{Perm}(v.\text{join}, \pi) \} v.\text{join}() \{ v.\text{postRun}\langle \pi \rangle \}$$

Outline

- 1 Background on Separation Logic
- 2 Class Axioms
- 3 Abstract Predicates and Subclassing
- 4 Fork/Join
- 5 Conclusion**

Contributions

- Design of a specification and verification system for a multithreaded Java-like language with fork/join, based on concurrent separation logic.
- Features:
 - class axioms
 - value-parametrized types
 - modular support for subclassing
(avoiding reverification of inherited methods)
 - support for Java-style fork/join concurrency
(allowing multiple joiners for the same thread)
- Soundness proof for a Java-like model language.

Related Work

Work that we built on and that influenced us:

- fractional permissions (Boyland'03)
- concurrent separation logic (O'Hearn'05, Bornat et al'05)
- abstract predicates (Parkinson'05)
- tpestates for objects (DeLine/Fähndrich'04)

Recent closely related work (independent):

- concurrent separation logic for POSIX-style threads (Gotsman/Berdine/Cook/Rinetzky/Sagiv APLAS'07, Hobor/Appel/Nardelli ESOP'08)
- abstract predicates and inheritance (Parkinson/Bierman POPL'08, Chin/David/Nguyen/Qin POPL'08)
- combining fractional permissions and nesting (Boyland UWM Tech Report Dec'07)

Other related work:

- dynamic frames (Kassios FM'06, Smans/Jacobs/Piessens FASE'08, Banerjee/Nauman/Rosenberg ECOOP'08)
- modular tpestate checking (Bierhoff/Aldrich OOPSLA'07)
- expressing heap shape contracts in linear logic (Perry/Jia/Walker GPCE'06)

Future and Current Work

- support for Java's reentrant locks (current)
- automatic assertion checking