Specificaton and Verification of Heap Access Policies

A Variant of Separation Logic for a Java-like Language

Christian Haack

Radboud University, Nijmegen

ESF Winter School on Verification of Object-Oriented Programs
Viinistu, Estonia, Jan 25-29, 2009

Introduction

- On Monday, we talked about pluggable type systems. These are "type systems" that are more complicated than "normal" type systems because types must be established in and *initalization phase* (just like object invariants).
- Generally, modular verification systems often enforce a discipline where the life of objects proceeds in phases.

Examples.

- Pluggable type systems: `Uninitialized`, `Initalized`
  - Immutable objects may only be mutated in the `Uninitialized` state.
  - Object invariants are only guaranteed in the `Initialized` state.
- $\mathrm{Spec}^\sharp$ methodology: `Valid`, `Exposed` (and some more)
  - Objects may only be mutated in the `Exposed` state.
  - Object invariants are only guaranteed in the `Valid` state.
- Monitor invariants: `Locked`, `Unlocked`
  - Objects may only be accessed in the `Locked` state.
  - Monitor invariants are only guaranteed in the `Unlocked` state.

Enforcing such disciplines enables modular soundness of verification systems.

Separation Logic

- A verification formalism that very tightly combines program logic with heap access policy to achieve modularity is separation logic.

Today we'll talk about (a fragment of) separation logic.

This will:

- not be a tool-talk (no verification tools of comparable maturity to Krakatoa, Key or $\mathrm{Spec}^\sharp$ yet exist, at least not in an OO setting)
- be a talk on separation logic adapted to a Java-like setting (based on joint work with Clement Hurlin and Marieke Huisman)

Basic 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)

Rules for Writing and Reading

Writing.

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

- The precondition is required as an access ticket for $x.f$.
- $\{\ \texttt{true}\ \}\ x.f = v\ \{\ x.f \mapsto v\ \}$ is NOT derivable.

Reading.

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

- Again, the precondition is required as an access ticket for $x.f$.

Fine print:

- x, y, z range over local variables (stack variables).
- v ranges over values, stack variables and logical variables.
- We postulate that heap-independent formulas F (e.g., $y == v$) are idempotent: F is equivalent to $F * F$.

The Frame Rule

This should be derivable:

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

Intuition: This rule is sound because F's validity cannot depend on the value of $x.f$, as F "does not have access to $x.f$". The access ticket to $x.f$ is with the left argument of $*$.

The Frame Rule.

$$\frac{\{ \ F \ \} \ c \ \{ \ G \ \} \quad \mathrm{fv}(H) \cap \mathrm{writes}(hc) = \emptyset}{\{ \ F * H \ \} \ c \ \{ \ G * H \ \}}$$

Fine print:

- $\mathrm{fv}(F)$ is the set of stack variables occurring in F.
- $\mathrm{writes}(c)$ is the set of stack variables x that occur on the lhs of an assignment `x = v;` in c.
- We assume throughout this talk a Java-like language without static fields.

Rule for Sequential Composition

$$\frac{\{\ F\ \}\ c\ \{\ G\ \} \quad \{\ G\ \}\ c'\ \{\ H\ \}}{\{\ F\ \}\ c; c'\ \{\ H\ \}}$$

Example.

$\{\texttt{this.f} \mapsto _ * \texttt{this.g} \mapsto _\}$

$\{\texttt{this.f} \mapsto _ * \texttt{this.g} \mapsto _\}$

```
this.f = 1;
```

$\{\texttt{this.f} \mapsto 1 * \texttt{this.g} \mapsto _\}$

$\{\texttt{this.f} \mapsto 1 * \texttt{this.g} \mapsto _\}$

```
this.g = 2;
```

$\{\texttt{this.f} \mapsto 1 * \texttt{this.g} \mapsto 2\}$

$\{\texttt{this.f} \mapsto 1 * \texttt{this.g} \mapsto 2\}$

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 ↦ _ * this.g ↦ _;
    //@ ensures this.f ↦ _ * this.g ↦ _;
    void run() { this.f = 1; this.g = 2; }

    //@ requires x.f ↦ _ * x.g ↦ _;
    //@ ensures true;
    void m(C x) { x.start(); }
}
```

Examples

Does this verify? Yes, but it's a silly contract.

```
//@ requires this.f ↦ _ * this.g ↦ _;
//@ ensures this.f ↦ _;
void m() { this.f = 1; this.g = 2; }
```

Does this verify? No.

```
//@ requires this.f ↦ _;
//@ ensures this.f ↦ _;
void m() { this.f = 1; this.g = 2; }
```

No Modifies Clauses Needed for Instance Fields

A JML-like contract:

```
//@ modifies this.f;
void set(int x) { this.f = x; }
```

Separation logic can be used to a similar effect:

```
//@ requires this.f ↦ _;
//@ ensures this.f ↦ _;
void set(int x) { this.f = x; }
```

Fine print:

- We are assuming no static fields. For these we would need modifies-clauses (but they are unproblematic because static fields can't be aliased).

Modifies Clauses vs. Separation Logic

```
//@ modifies x.f, y.f;
void m(Object x, Object y);
```

```
//@ requires x.f ↦ _ * y.f ↦ _;
//@ ensures x.f ↦ _ * y.f ↦ _;
void m(Object x, Object y);
```

- In the former, a call `m`(x, y) is allowed for all x, y:
- In the latter, a call `m`(x, y) is only allowed when x != y:

In separation logic, one could write this:

```
//@ requires x.f ↦ _ * x = y; ensures x.f ↦ _ * x = y;
//@ also
//@ requires x.f ↦ _ * y.f ↦ _; ensures x.f ↦ _ * y.f ↦ _;
void m(Object x, Object y);
```

Separation Logic and \old

- Separation logic does not use \old.
- Instead one can parametrize method contracts with logical variables.

Example.

```
//@ <Object x, Object y>
//@ requires this.f |-> x * this.g |-> y;
//@ ensures ens this.f |-> y * this.g |-> x;
void swap(); // swaps this.f and this.g
```

- In examples, I will often leave the logical parameters implicit.

Abstract Predicates

- Classes can define predicates.

```
class C {

    int f;
    int g;

    //@ pred space = this.f ↦ _ * this.g ↦ _;

    //@ requires this.space;
    //@ ensures this.space;
    void m() { this.f = 1; this.g = 2; }

}
```

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

Example: Datagroups

```
interface Sprite {
   //@ pred space;
   //@ pred position;
   //@ pred color;

   //@ requires position; ensures position;
   void updatePosition();
   //@ requires color; ensures color;
   void updateColor();
   //@ requires space; ensures space;
   void update();
   //@ requires space; ensures space;
   void display();
}
```

We would like to express in the interface, that `position` and `color` are nested in a `Sprite`'s `space`.

Class Axioms

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

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

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

Resource Implication

Resource implication:

$$F \texttt{ -* } G$$

- ticket to trade ticket F for ticket G
- linear modus ponens: $F * (F \texttt{ -* } G) \texttt{ -* } G$

Derived Form:

$$F \texttt{ ispartof } G \triangleq G \texttt{ -* } (F * (F \texttt{ -* } G))$$

Example: Nested Datagroups

```
interface Sprite {
   //@ pred space;
   //@ pred position;
   //@ pred color;

   //@ axiom position ispartof space;
   //@ axiom color ispartof space;

   //@ requires position; ensures position;
   void updatePosition();

   //@ requires color; ensures color;
   void updateColor();

   //@ requires space; ensures space;
   void update();

   //@ requires space; ensures space;
   void display();
}
```

Example: Nested Datagroups (Implementation)

```
class SpriteImpl implements Sprite {
    Position pos;
    Color col;
    //@ pred space = this.pos ↦ _ * this.col ↦ _;
    //@ pred position = this.pos ↦ _;   // P
    //@ pred color = this.col ↦ _;      // C
    //@ axiom position ispartof space;
    ...
}
```

Code producer must prove the axiom (after substituting actual predicate definitions).

$$(P * C) \mathrel{-\!\!*} (P * (P \mathrel{-\!\!*} (P * C)))$$

$$\dfrac{\dfrac{}{P \vdash P} \qquad \dfrac{\dfrac{}{C, P \vdash P * C}}{C \vdash P \mathrel{-\!\!*} (P * C)}}{\dfrac{P, C \vdash P * (P \mathrel{-\!\!*} (P * C))}{\vdash (P * C) \mathrel{-\!\!*} (P * (P \mathrel{-\!\!*} (P * C)))}}$$

Example: Iterators

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

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

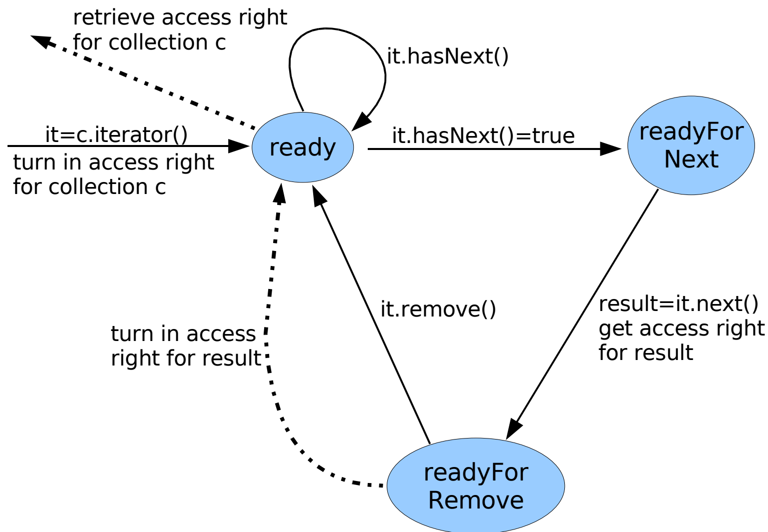
Prevent:

- concurrent modifications of collection
- concurrent mutations of collection elements

To this end:

- statically enforce disciplined iterator usage

Example: A Usage Protocol for Iterators



A Typical Use of Resource Implication

Resource implication is useful for representing the right to make a state transition:

- If F and G represent abstract program states, then $F \mathrel{-\!\!*} G$ represents the right to move from F to G.

Another Logical Operators: Choice

Resource conjunction:

$F * G$

- resources F and G are available and are independent
- ticket to use both resources F and G in any order

Choice:

$F \,\&\, G$

- resources F and G are available and are interdependent
- ticket to use one of F and G, we can choose which one

When representing state machines, & is useful for representing non-determinism.

Example: Iterator Usage Protocol (Collection)

- We assume that every class defines a special abstract predicate `space` in order to specify its associated heap space.

```
interface Collection {
    //@ requires this.space * e.space;
    //@ ensures this.space;
    void add(Object e);

    //@ requires this.space;
    //@ ensures result.ready;
    Iterator/*@<this>@*/ iterator();
}
```

- We parametrize the `Iterator` type by the underlying collection, because there is a natural semantic dependency.

Example: Iterator Usage Protocol (Iterator)

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

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

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

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

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

The Logical Consequence Rule

$$\frac{F \vdash F' \quad \{\ F'\ \}\ c\ \{\ G'\ \} \quad G' \vdash G}{\{\ F\ \}\ c\ \{\ G\ \}}$$

- Here, $F \vdash G$ means G is a logical consequence of F.
- But how do we formally define logical consequence in the presence of the funny resource resource operator?
- We define logical consequence proof-theoretically by a set of proof rules (soundly but incompletely).
- The proof rules are the natural deduction rules of a fragment of linear logic with weakening.

Natural Deduction Rules

Format:

$$\bar{F} \vdash G$$

where $\bar{F}$ is a *multiset* of formulas (not a set).

- Intuitively this means: $F_1 * \cdots * F_n$ implies G

Rules:

(Id)

$$\frac{}{\bar{F}, G \vdash G}$$

(Ax)

$$\frac{G \text{ is an axiom}}{\bar{F} \vdash G}$$

($*$ Intro)

$$\frac{\bar{F} \vdash H_1 \quad \bar{G} \vdash H_2}{\bar{F}, \bar{G} \vdash H_1 * H_2}$$

($*$ Elim)

$$\frac{\bar{F} \vdash G_1 * G_2 \quad \bar{E}, G_1, G_2 \vdash H}{\bar{F}, \bar{E} \vdash H}$$

(−∗ Intro)

$$\frac{\bar{F}, G_1 \vdash G_2}{\bar{F} \vdash G_1 \;-\!\!*\; G_2}$$

(−∗ Elim)

$$\frac{\bar{F} \vdash H_1 \;-\!\!*\; H_2 \quad \bar{G} \vdash H_1}{\bar{F}, \bar{G} \vdash H_2}$$

(& Intro)

$$\frac{\bar{F} \vdash G_1 \quad \bar{F} \vdash G_2}{\bar{F} \vdash G_1 \;\&\; G_2}$$

(& Elim 1)

$$\frac{\bar{F} \vdash G_1 \;\&\; G_2}{\bar{F} \vdash G_1}$$

(& Elim 2)

$$\frac{\bar{F} \vdash G_1 \;\&\; G_2}{\bar{F} \vdash G_2}$$

(| Intro 1)

$$\frac{\bar{F} \vdash G_1}{\bar{F} \vdash G_1 \;|\; G_2}$$

(| Intro 2)

$$\frac{\bar{F} \vdash G_2}{\bar{F} \vdash G_1 \;|\; G_2}$$

(| Elim)

$$\frac{\bar{F} \vdash G_1 \;|\; G_2 \quad \bar{E}, G_1 \vdash H \quad \bar{E}, G_2 \vdash H}{\bar{F}, \bar{E} \vdash H}$$

Fine print:

- I have omitted the quantifier rules.

Semantics of Formulas (a sketch)

$$
\begin{array}{rcl}
\mathsf{Val} & = & \mathsf{ObjId} \cup \mathsf{Integer} \cup \{\texttt{null}\} \\
h \in \mathsf{Heap} & = & \mathsf{ObjId} \times \mathsf{FieldId} \rightharpoonup \mathsf{Val} \qquad \text{(types omitted)} \\
s \in \mathsf{Stack} & = & \mathsf{Var} \rightharpoonup \mathsf{Val}
\end{array}
$$

e ranges over heap-independent expressions

$$
\begin{array}{lcll}
h; s & \models & e & \text{iff } [\![e]\!](s) = \texttt{true} \\
h; s & \models & F * G & \text{iff } (\exists h_1, h_2)(h_1 \cap h_2 = \emptyset, h = h_1 \cup h_2, h_1; s \models F \text{ and } h_2; s \models G) \\
h; s & \models & F \;\texttt{-*}\; G & \text{iff } (\forall h')(h \cap h' = \emptyset \text{ and } h'; s \models F \text{ implies } h \cup h'; s \models G) \\
h; s & \models & F \;\texttt{\&}\; G & \text{iff } h; s \models F \text{ and } h; s \models G \\
h; s & \models & F \;\texttt{|}\; G & \text{iff } h; s \models F \text{ or } h; s \models G
\end{array}
$$

- Universal quantifiers implicitly quantify over all heap extensions.
- With our semantics, we can only express properties that are closed under heap extensions.

- I omitted the semantics of abstract predicates.
- I implicitly assume that all heaps are well-typed.

Soundness of the Natural Deduction Rules

We define semantic entailment $\bar{F} \models G$:

$$F_1, \ldots F_2 \models G \qquad \text{iff} \qquad (\forall h, s)(h; s \models F_1 * \cdots * F_n \text{ implies } h; s \models G)$$

Theorem

If $(\bar{F} \vdash G)$, *then* $(\bar{F} \models G)$.

Extending Predicates

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

```
class C {
    int f;
    //@ pred space = this.f ↦ _;
}

class D extends C {
    int g;
    //@ pred space = this.g ↦ _;
      // total space: this.f ↦ _ * this.g ↦ _
}
```

- Note that the the extensions in different classes must be independent of each other by ∗-conjunction. (This is sometimes too restrictive.)

Axiomatizing a Stack of Class Frames

e.P@C<ē>	e.P<ē> holds "down to" class *C*
e.P<ē>	equivalent to e.P@D<ē> where *D* is *e*'s dynamic class

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

- e.P@C<ē> ispartof e.P<ē>

  recall that this desugars to

  e.P<ē> -* (e.P@C<ē> * (e.P@C<ē> -* e.P<ē>))

- this.P@C<ē> *-* (*F* * this.P@D<ē>)

  if *F* is *P@C<ē>*'s definition in *C*, and $C \preceq_1 D$

  F *-* *G* $\triangleq$ (*F* -* *G*) & (*G* -* *F*)

Axiom for promoting qualified to unqualified predicates:

- (e.P@C<ē> * *C* isclassof *e*) -* e.P<ē>

  this axiom is typically applied right after object constructors terminate

Forking Threads

The original rule from concurrent separation logic:

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

This rule:

- prevents data races (good)
- disallows concurrent reads (not good)

Fractional Permissions for Concurrent Reads

Due to John Boyland.

Superscripting points-to with fractions.

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

The split/merge axiom.

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

Permission 1 grants write access:

$$\{\ x.f \overset{1}{\mapsto} _\ \}\ x.f = v\ \{\ x.f \overset{1}{\mapsto} v\ \}$$

Any permission grants read access:

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

This allows concurrent reads, while preventing data races.

Example: Split/Mergeable Predicates (Datagroups)

```
class Point {
    int x,y;

    //@ pred space<perm p> = this.x ↦^p _ * this.y ↦^p _;

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

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

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

group $P<\bar{T}\ \bar{x}>;\quad \triangleq\quad$ pred $P<\bar{T}\ \bar{x}>;$ axiom $P<\bar{x}> *\!-\!* (P<\bar{e}> * P<\bar{e}>);$

$$\text{where } e_i \triangleq \begin{cases} x_i/2 & \text{if } T_i = \texttt{perm} \\ x_i & \text{otherwise} \end{cases}$$

Example: Recursive and Overlapping Datagroups

```
interface StudentList {
  //@ group space<perm p>;
  //@ group ids_and_links<perm p, perm q>;
  //@ group grades_and_links<perm p, perm q>;

  //@ axiom
  //@   space<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.
- One thread can safely update the grades, while another one reads the ids.

Fork/Join Instead of Parallel Composition

```
class Thread {
    //@ pred preRun;
    //@ pred postRun;

    //@ requires preRun; ensures postRun;
    abstract void run();

    final void start();   // native
    final void join();    // native
}
```

- Use `run`'s precondition as `start`'s precondition and `run`'s postcondition as `start`'s postcondition.
- But we need to avoid multiple `joiner`'s.

Proof Rules for Fork/Join

New formula.

`v.join` permission to join thread v

- This formulas serves as a ticket for joining a thread.
- It is part of the postcondition for creating a Thread object.

Forking.

$$\{\ v \texttt{ != null} * v.\texttt{preRun}\ \}\ v.\texttt{start()}\ \{\ \texttt{true}\ \}$$

Joining.

$$\{\ v.\texttt{join}\ \}\ v.\texttt{join()}\ \{\ v.\texttt{postRun}\ \}$$

Monitor Invariants (for Single-Entrant Locks)

- Every object extends a special abstract predicate, its *monitor invariant*.

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

- After you acquire a lock on an initialized object, you can assume the invariant (and obtain access to the associated heap space):

$$\{\ v.\texttt{initialized}\ \}\ v.\texttt{lock()}\ \{\ v.\texttt{inv}\ \}$$

- When you release a lock, you need to re-establish the invariant (and give up access to the associated heap space):

$$\{\ v.\texttt{inv}\ \}\ v.\texttt{unlock()}\ \{\ \texttt{true}\ \}$$

Lock Initialization

Two special formulas:

- v.fresh: v's resource invariant is not yet initialized.
- v.initialized: v's resource invariant has been initialized.

- v.initialized is copyable (by the following axiom):

$$v.\texttt{initialized} \;\texttt{-*}\; (v.\texttt{initialized} * v.\texttt{initialized})$$

- After creation, objects are fresh:

$$\{\texttt{true}\}v = \texttt{new}\; C\texttt{<}\bar{\pi}\texttt{>}\{v.\texttt{init} * C \;\texttt{isclassof}\; v * v.\texttt{fresh} * v.\texttt{join}\}$$

v.init is the $*$-conjunction of formulas $v.f \stackrel{1}{\mapsto} w$ where f is a field of v and w is the default value for f's type.

- A special specification command commit:

$$\{v.\texttt{inv} * v.\texttt{fresh}\}v.\texttt{commit}\{v.\texttt{initialized}\}$$

Reentrant Locks

The previous rules are unsound for reentrant locks.

Ways to make monitor invariants sound for reentrant locks:

- Prevent deadlocks (and reentries) by imposing an order on acquiring locks.
  - Caveat: reentries disallowed.
- Allow reentries by keeping track of a multiset of currently held locks for each thread, and based on this distinguish between entry and reentry.
  - Caveat: requires reasoning about the absence of aliasing.

Conclusion

- Separation logic tightly integrates heap access control with program logic.
- Writing specs in separation logic takes a slightly different mind set at first.
- It can be used very elegantly when combined with abstraction (e.g., the `Iterator` protocol).
- I have so far mostly used it for "lightweight specifications" and wonder how convenient it is for more expressive functional specs in an OO setting.
- There are not yet any separation-logic-based verification tools for OO languages whose maturity comes close to the tools presented in this course, but some are emerging (e.g., jStar by Distefano and Parkinson).
- I think it could be very rewarding to exploit separation logic or related alias control techniques in existing verification tools. See Claude Marché's region analysis in the Krakatoa tool, as presented in this winter school.

References

R. Bornat, P. W. O'Hearn, C. Calcagno, and M. Parkinson.
Permission accounting in separation logic.
In *Principles of Programming Languages*, pages 259–270, New York, NY, USA, 2005. ACM Press.

C. Haack and C. Hurlin.
Resource usage protocols for iterators.
In *International Workshop on Aliasing, Ownership and Confinement (IWACO)*, 2008.
Revised version available from `http://www.cs.ru.nl/~chaack/papers/iterators.pdf`.

C. Haack and C. Hurlin.
Separation logic contracts for a Java-like language with fork/join.
In *Algebraic Methodology and Software Technology*, number 5140 in Lecture Notes in Computer Science, pages 199–215. Springer-Verlag, 2008.

C. Haack, M. Huisman, and C. Hurlin.
Reasoning about Java's reentrant locks.
In *Asian Programming Languages and Systems Symposium*, December 2008.

M. Parkinson.
Local reasoning for Java.
Technical Report UCAM-CL-TR-654, University of Cambridge, 2005.

M. Parkinson and G. Bierman.
Separation logic, abstraction and inheritance.
In *Principles of Programming Languages*, pages 75–86, 2008.