Separation Logic

Niccolò Piazzesi

January 5, 2022

Outline

Outline

Brief recap: reasoning about code

- Program semantics described by logical conditions satisfied by language constructs
- Classical model, first put forward by Robert W. Floyd and Tony Hoare

Floyd-Hoare Logic in 1 slide

$$\{P\}S\{Q\}$$

- P : pre-conditions
- S : statement
- Q : post conditions

Partial correctness: **If the inital state fullfils pre-conditions and the statement terminates**, the final state satisfies the post conditions.

Total correctness: **If the initial state fullfils the pre-conditions** then the statement terminates and the final state satisfies the post-conditions.

Limitations

Does not work for non terminating programs

Limitations

Does not work for non terminating programs

Becomes complex with modular constructs such as objects and unconditional jumps

Limitations

Does not work for non terminating programs

Becomes complex with modular constructs such as objects and unconditional jumps

Global view of state becomes a burden when introducing pointers(think of pointer aliasing..)

Motivating example

```c
void deletetree(struct node *root){
    if(root != NULL){
    struct node *left = root->l;
    struct node *right = root->r;
    deletetree(left);
    deletetree(right);
    free(root);
    }
}
```

How can we prove memory safety?

First attempt

Specification

$\{h: \ tree(t, h)\}$
$deletetree(t)$
$\{h' : true\}$

Proof

First attempt

Specification

$$\{h: \ tree(t,h)\}$$
$$deletetree(t)$$
$$\{h': true\}$$

Proof

$$\{h : h[t] = [l, r]$$
$$\wedge\, tree(l, h)$$
$$\wedge\, tree(r, h)$$
$$\wedge\, t, l, r \ distinct\}$$

First attempt

Specification

$$\{h: \ tree(t,h)\}$$
$$deletetree(t)$$
$$\{h' : true\}$$

Proof

$$\{h : h[t] = [l, r]$$
$$\wedge\, tree(l, h)$$
$$\wedge\, tree(r, h)$$
$$\wedge\, t, l, r \ distinct\}$$

$$deletetree(l)$$

First attempt

Specification

$$\{h : \ tree(t, h)\}$$
$$deletetree(t)$$
$$\{h' : true\}$$

Proof

$$\{h : h[t] = [l, r]$$
$$\wedge\, tree(l, h)$$
$$\wedge\, tree(r, h)$$
$$\wedge\, t, l, r \ distinct\}$$

$$deletetree(l)$$
$$\{h' : true\}$$

First attempt

Specification

$$\{h: \; tree(t,h)\}$$
$$deletetree(t)$$
$$\{h' : true\}$$

Proof

$$\{h : h[t] = [l, r]$$
$$\wedge\, tree(l, h)$$
$$\wedge\, tree(r, h)$$
$$\wedge\, t, l, r \; distinct\}$$

$$deletetree(l)$$
$$\{h' : true\}$$

We can't prove safety of tree(r,h)!

First attempt

Specification

$\{h: \; tree(t, h)\}$
$deletetree(t)$

$\{h' : \forall p, h'[p] = h[p]$
if p is not in the tree $\}$

Proof

$$\{h : h[t] = [l, r]$$
$$\wedge\, tree(l, h)$$
$$\wedge\, tree(r, h)$$
$$\wedge\, t, l, r \;\, distinct\}$$

$$deletetree(l)$$

First attempt

Specification

$\{h: \ tree(t,h)\}$
$deletetree(t)$

$\{h' : \forall p, h'[p] = h[p]$
if p is not in the tree $\}$

Proof

$\{h : h[t] = [l, r]$
$\wedge\, tree(l, h)$
$\wedge\, tree(r, h)$
$\wedge\, t, l, r \ distinct\}$

$deletetree(l)$

How can we be sure that deletetree(l)
does not modify tree(r,h)? We should say that in tree(t,h)...

Outline

The model

Ints $\triangleq \{\cdots, -1, 0, 1, \cdots\}$

Atoms, Locations $\subseteq Ints$

Stores $\triangleq$ Variables $\rightharpoonup_{fin}$ Ints

Variables $\triangleq \{x, y, \cdots\}$

Locations $\cap$ Atoms $= \{\}$, nil $\in$ Atoms

Heaps $\triangleq$ Locations $\rightharpoonup_{fin}$ Ints

States $\triangleq$ Stores $\times$ Heaps

$[\![E]\!]_s \in Ints$, $[\![B]\!]_s \in \{\text{true}, \text{false}\}$

$h \in Heaps$, $h[E] \in Ints$

The language

Expressions:

$$E, F, G := x, y, \cdots \mid 0 \mid 1 \mid E + F \mid E \times F \mid E - F$$

$$B := false \mid B \Rightarrow B \mid E = F \mid E < F \mid isatom?(E) \mid isloc?(E)$$

The language

Expressions:

$$E, F, G := x, y, \cdots \mid 0 \mid 1 \mid E + F \mid E \times F \mid E - F$$
$$B := false \mid B \Rightarrow B \mid E = F \mid E < F \mid isatom?(E) \mid isloc?(E)$$

Assertions:

$$\begin{aligned} P, Q, R ::= {} & B \mid E \mapsto F && \textit{Atomic Formulae} \\ & \mid false \mid P \Rightarrow Q \mid \forall x.P && \textit{Classical Logic} \\ & \mid emp \mid P * Q \mid P \mathrel{-\!\!*} Q && \textit{Spatial Connectives} \end{aligned}$$

The language

Expressions:

$$E, F, G := x, y, \cdots \mid 0 \mid 1 \mid E + F \mid E \times F \mid E - F$$
$$B := false \mid B \Rightarrow B \mid E = F \mid E < F \mid isatom?(E) \mid isloc?(E)$$

Assertions:

$$\begin{array}{llr} P, Q, R ::= & B \mid E \mapsto F & \textit{Atomic Formulae} \\ & \mid false \mid P \Rightarrow Q \mid \forall x.P & \textit{Classical Logic} \\ & \mid emp \mid P * Q \mid P -\!\!* \; Q & \textit{Spatial Connectives} \end{array}$$

$\neg P = P \Rightarrow False$
$true = \neg(false)$
$P \vee Q = \neg(P) \Rightarrow Q$

The language

Expressions:

$$E, F, G := x, y, \cdots \mid 0 \mid 1 \mid E + F \mid E \times F \mid E - F$$
$$B := false \mid B \Rightarrow B \mid E = F \mid E < F \mid isatom?(E) \mid isloc?(E)$$

Assertions:

$$\begin{array}{llr} P, Q, R ::= B \mid E \mapsto F & & Atomic\ Formulae \\ & \mid false \mid P \Rightarrow Q \mid \forall x.P & Classical\ Logic \\ & \mid emp \mid P * Q \mid P -\!\!* \ Q & Spatial\ Connectives \end{array}$$

$$\neg P = P \Rightarrow False$$
$$true = \neg(false)$$
$$P \vee Q = \neg(P) \Rightarrow Q$$

$$P \wedge Q = \neg(\neg P \vee \neg Q)$$
$$\exists x.P = \neg \forall x. \neg P$$

Some notation

1. $dom(h)$ and $dom(s)$ denote the domain of definition for $h \in Heaps$ and $s \in Stores$, respectively
2. $h\#h' \to dom(h) \cap dom(h') = \emptyset$
3. $h * h'$ is the union of disjoint heaps
4. $(f|i \mapsto j)$ represent the partial function that behaves like f except that i goes to j.

$$E \mapsto F_0, \cdots, F_n \triangleq (E \mapsto F_0) * \cdots * (E + n \mapsto F_n)$$

$$E \doteq F \triangleq (E = F) \wedge emp$$

$$E \mapsto - \triangleq \exists y.E \mapsto y$$

Semantics

For store s and heap h

Semantics

For store s and heap h

$$s, h \models B \; iff \; [\![B]\!]_s = true$$

$$s, h \models E \mapsto F \; iff \; \{[\![E]\!]_s\} = dom(h) \; and \; h([\![E]\!]_s) = [\![F]\!]_s$$

$$s, h \models false \quad never$$

$$s, h \models P \Rightarrow Q \; iff \; if \; s, h \models P \; then \; s, h \models Q$$

$$s, h \models \forall x.P \; iff \; \forall v \in Ints.[s \mid x \mapsto v], h \models P$$

Semantics

For store s and heap h

$$s, h \models B \ iff \ [\![B]\!]_s = true$$

$$s, h \models E \mapsto F \ iff \ \{[\![E]\!]_s\} = dom(h) \ and \ h([\![E]\!]_s) = [\![F]\!]_s$$

$$s, h \models false \quad never$$

$$s, h \models P \Rightarrow Q \ iff \ if \ s, h \models P \ then \ s, h \models Q$$

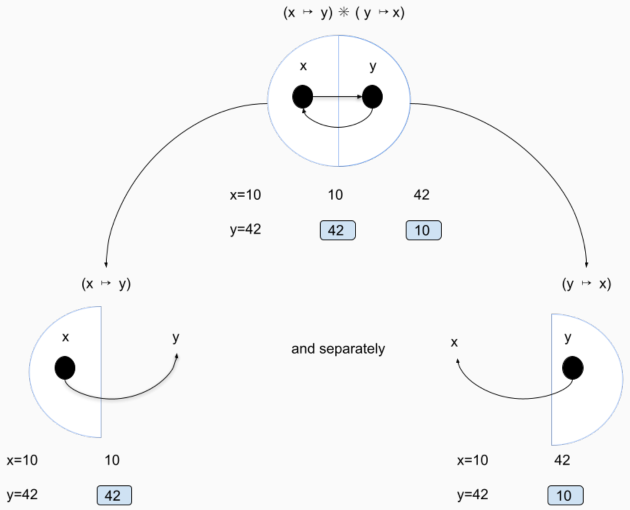
$$s, h \models \forall x.P \ iff \ \forall v \in Ints.[s \mid x \mapsto v], h \models P$$

$$s, h \models emp \ iff \ h = [] \ is \ the \ empty \ heap$$

$$s, h \models P * Q \ iff \ \exists h_0, h_1.h_0 \# h_1, \ h_0 * h_1 = h, \ s, h_0 \models P \ and \ s, h_1 \models Q$$

$$s, h \models P -\!\!* \ Q \ iff \ \forall h'. \ if \ h' \# h \ and \ s, h' \models P \ then \ s, h * h' \models Q$$

Visual example



Core system

Proof rules in seapartion logic are divide in:

1. Axioms for basic mutation commands $\rightarrow$ *Small axioms*
2. Inference rules for modular reasoing $\rightarrow$ *Structural rules*

Small axioms

Small axioms

$\{E \mapsto -\}[\mathbf{E}] := \mathbf{F}\{E \mapsto F\}$ ("Store")

Small axioms

$\{E \mapsto -\}[\mathbf{E}] := \mathbf{F}\{E \mapsto F\}$ ("Store")

$\{E \mapsto -\}\mathbf{free}(\mathbf{E})\{emp\}$ ("Reclaim memory")

Small axioms

$\{E \mapsto -\}[\mathbf{E}] := \mathbf{F}\{E \mapsto F\}$ ("Store")

$\{E \mapsto -\}\mathbf{free}(\mathbf{E})\{emp\}$ ("Reclaim memory")

$\{x \doteq m\}\mathbf{x} := \mathbf{cons}(\mathbf{E_1}, \cdots, \mathbf{E_k})\{x \mapsto E_1[m/x], \cdots, E_k[m/x]\}$ ("Allocate memory")

Small axioms

$\{E \mapsto -\}[\mathbf{E}] := \mathbf{F}\{E \mapsto F\}$ ("Store")

$\{E \mapsto -\}\mathbf{free}(\mathbf{E})\{emp\}$ ("Reclaim memory")

$\{x \doteq m\}\mathbf{x} := \mathbf{cons}(\mathbf{E_1}, \cdots, \mathbf{E_k})\{x \mapsto E_1[m/x], \cdots, E_k[m/x]\}$ ("Allocate memory")

$\{x \doteq n\}\mathbf{x} := \mathbf{E}\{x \doteq (E[n/x])\}$

Small axioms

$\{E \mapsto -\}[\mathbf{E}] := \mathbf{F}\{E \mapsto F\}$ ("Store")

$\{E \mapsto -\}\mathbf{free}(\mathbf{E})\{emp\}$ ("Reclaim memory")

$\{x \doteq m\}\mathbf{x} := \mathbf{cons}(\mathbf{E_1}, \cdots, \mathbf{E_k})\{x \mapsto E_1[m/x], \cdots, E_k[m/x]\}$
("Allocate memory")

$\{x \doteq n\}\mathbf{x} := \mathbf{E}\{x \doteq (E[n/x])\}$

$\{E \mapsto n \wedge x = m\}\mathbf{x} := [\mathbf{E}]\{x = n \wedge E[m/x] \mapsto n\}$ ("Load")

Structural rules

Frame Rule

$$\frac{\{P\}C\{Q\}}{\{P * \color{red}{frame}\}C\{Q * \color{red}{frame}\}}\ Mod(C) \cap Free(frame) = \emptyset$$

Structural rules

Frame Rule

$$\frac{\{P\}C\{Q\}}{\{P * \color{red}{frame}\color{black}\}C\{Q * \color{red}{frame}\color{black}\}} \; Mod(C) \cap Free(frame) = \emptyset$$

Auxiliary variable elimination

$$\frac{\{P\}C\{Q\}}{\{\exists x.P\}C\{\exists x.Q\}} \; x \notin Free(C)$$

Structural rules

Variable substitution

$$\frac{\{P\}C\{Q\}}{(\{P\}C\{Q\})[E_1/x_1, \cdots E_k/x_k]}$$

$$x_i \ free \ and \ if \ x_i \in Mod(C) \ then \ E_i \ is \ not \ free \ in \ any \ E_j$$

Structural rules

Variable substitution

$$\frac{\{P\}C\{Q\}}{(\{P\}C\{Q\})[E_1/x_1, \cdots E_k/x_k]}$$

$$x_i \ free\ and\ if\ x_i \in Mod(C)\ then\ E_i\ is\ not\ free\ in\ any\ E_j$$

Rule of consequence

$$\frac{P \Rightarrow P' \quad \{P\}C\{Q\} \quad Q \Rightarrow Q'}{\{P\}C\{Q'\}}$$

Derived laws

The structural rules can be used to obtain more convenient derived laws.
As an example, we can simplify the rule for memory allocation by assuming $x \notin Free(E_1, \cdots, E_k)$.

$$\{emp\}x := cons(E_1, \cdots, E_k)\{x \mapsto E_1, \cdots, E_k\}$$

Derived laws

The structural rules can be used to obtain more convenient derived laws.
As an example, we can simplify the rule for memory allocation by assuming $x \notin Free(E_1, \cdots, E_k)$.

$$\{emp\}x := cons(E_1, \cdots, E_k)\{x \mapsto E_1, \cdots, E_k\}$$

The core system can also be extended with the usual Hoare rules

$$\frac{\{P \wedge B\}C\{Q\} \quad \{P \wedge \neg B\}C\{Q\}}{\{P\}if\ B\ then\ C\ else\ C'\{Q\}}$$

Revisiting the tree example

```
void deletetree(struct node *root){
    if(root != NULL){
    struct node *left = root->l;
    struct node *right = root->r;
    deletetree(left);
    deletetree(right);
    free(root);
    }
}
```

Revisiting the tree example

```
void deletetree(struct node *root){
    if(root != NULL){
    struct node *left = root->l;
    struct node *right = root->r;
    deletetree(left);
    deletetree(right);
    free(root);
    }
}
```

Specification:

{tree(root)} deletetree(root) {emp}

$$\text{tree(root)} = if\ root == 0\ then\ emp$$
$$else\ \exists xy.root \mapsto [l : x, r : y] * tree(x) * tree(y)$$

Proof:

$$\{root \mapsto [l : left, r : right] * tree(left) * tree(red)\}$$

Proof:

$\{root \mapsto [l : left, r : right] * tree(left) * tree(red)\}$

$deletetree(left);$

Proof:

$\{root \mapsto [l : left, r : right] * tree(left) * tree(red)\}$

$deletetree(left);$

$\{root \mapsto [l : left, r : right] * emp * tree(red)\}$

Proof:

$\{root \mapsto [l : left, r : right] * tree(left) * tree(red)\}$

$deletetree(left);$

$\{root \mapsto [l : left, r : right] * emp * tree(red)\}$

$deletetree(right);$

Proof:

$\{root \mapsto [l : left, r : right] * tree(left) * tree(red)\}$

$deletetree(left);$

$\{root \mapsto [l : left, r : right] * emp * tree(red)\}$

$deletetree(right);$

$\{root \mapsto [l : left, r : right] * emp * emp\}$

Proof:

$\{root \mapsto [l : left, r : right] * tree(left) * tree(red)\}$

$deletetree(left);$

$\{root \mapsto [l : left, r : right] * emp * tree(red)\}$

$deletetree(right);$

$\{root \mapsto [l : left, r : right] * emp * emp\}$

$free(root);$

Proof:

$\{root \mapsto [l : left, r : right] * tree(left) * tree(red)\}$

$deletetree(left);$

$\{root \mapsto [l : left, r : right] * emp * tree(red)\}$

$deletetree(right);$

$\{root \mapsto [l : left, r : right] * emp * emp\}$

$free(root);$

$\{emp * emp * emp\}$
$\{emp\}$

Outline

Outline

Proofs in SL are very simple by design, but automation is needed to scale the analysis to large programs.
To fully automate proofs we need a way to infer pre and post conditions from bare code.

In SL this is solved with **bi-abduction**

Proofs in SL are very simple by design, but automation is needed to scale the analysis to large programs.
To fully automate proofs we need a way to infer pre and post conditions from bare code.

In SL this is solved with **bi-abduction**

$$(A * ?antiframe \vdash B * ?frame)$$

Can we find a pair of frame and antiframe that make the entailment valid?

Biabduction in action

Suppose we have the code

(closeResource(r1);closeResource(r2))

A human would say that we can execute closeResource(r1) only if we have $r1 \mapsto open$.
The equivalent biabduction question is

$$(emp * ?antiframe \vdash r1 \mapsto open * ?frame)$$

Biabduction in action

$$(emp * \color{red}{?antiframe} \vdash r1 \mapsto open * \color{blue}{?frame})$$

Biabduction in action

$$(emp * ?antiframe \vdash r1 \mapsto open * ?frame)$$

$$(antiframe = r1 \mapsto open),\ (frame = emp)$$

Biabduction in action

$$(emp * ?antiframe \vdash r1 \mapsto open * ?frame)$$

$$(antiframe = r1 \mapsto open),\ (frame = emp)$$

$$(emp * r1 \mapsto open \vdash r1 \mapsto open * emp)$$
$$(r1 \mapsto open \vdash r1 \mapsto open)$$

Biabduction in action

$$(emp * ?antiframe \vdash r1 \mapsto open * ?frame)$$

$$(antiframe = r1 \mapsto open),\ (frame = emp)$$

$$(emp * r1 \mapsto open \vdash r1 \mapsto open * emp)$$
$$(r1 \mapsto open \vdash r1 \mapsto open)$$

$$\{r1 \mapsto open\}$$
$$(closeResource(r1))$$
$$\{r1 \mapsto closed\}$$

Biabduction in action

$(emp * ?antiframe \vdash r1 \mapsto open * ?frame)$

$(antiframe = r1 \mapsto open),\ (frame = emp)$

$(emp * r1 \mapsto open \vdash r1 \mapsto open * emp)$
$(r1 \mapsto open \vdash r1 \mapsto open)$

$\{r1 \mapsto open\}$
$(closeResource(r1))$
$\{r1 \mapsto closed\}$

Let's now consider $closeResource(r2)$

Biabduction in action

$$(r1 \mapsto closed * ?antiframe \vdash r2 \mapsto open * ?frame)$$

Biabduction in action

$$(r1 \mapsto closed * ?antiframe \vdash r2 \mapsto open * ?frame)$$

$$(antiframe = r2 \mapsto open),\ (frame = r1 \mapsto closed)$$

Biabduction in action

$$(r1 \mapsto closed * ?antiframe \vdash r2 \mapsto open * ?frame)$$

$$(antiframe = r2 \mapsto open), \ (frame = r1 \mapsto closed)$$

$(\{r1 \mapsto open * r2 \mapsto open\})$
$(closeResource(r1))$
$(\{r1 \mapsto closed * r2 \mapsto open\})$
$(closeResource(r2))$
$\{r1 \mapsto closed, \ r2 \mapsto closed\}$

Outline

- Peter O'Hearn, John Reynolds, and Hongseok Yang.
  Local reasoning about programs that alter data structures.
  In *International Workshop on Computer Science Logic*, pages 1–19. Springer, 2001.

- Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez.
  Moving fast with software verification.
  In *NASA Formal Methods Symposium*, pages 3–11. Springer, 2015.

- Dino Distefano, Peter W. O'Hearn, and Hongseok Yang.
  A local shape analysis based on separation logic.
  In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 287–302, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn.
Smallfoot: Modular automatic assertion checking with separation logic.
In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 115–137, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

James Brotherston, Nikos Gorogiannis, Max Kanovich, and Reuben Rowe.
Model checking for symbolic-heap separation logic with inductive predicates.
ACM SIGPLAN Notices, 51(1):84–96, 2016.

Josh Berdine, Byron Cook, and Samin Ishtiaq.
Slayer: Memory safety for systems-level code.
In *International Conference on Computer Aided Verification*, pages 178–183. Springer, 2011.

Stephen Brookes and Peter W O'Hearn.
Concurrent separation logic.
ACM SIGLOG News, 3(3):47–65, 2016.