

Separation Logic

Niccolò Piazzesi

Università degli studi di Pisa

January 6, 2022



UNIVERSITÀ DI PISA

Outline

- 1 Introduction
- 2 Theoretical Foundations
- 3 Extension to concurrency
- 4 Biabduction
- 5 Tools

Outline

- 1 Introduction
- 2 Theoretical Foundations
- 3 Extension to concurrency
- 4 Biabduction
- 5 Tools

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

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

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

Partial correctness: **If the initial 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

```
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?

Specification

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

Proof

Specification

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

Proof

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

Specification

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

Proof

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

 $\text{deletetree}(l)$

Specification

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

Proof

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

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

Specification

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

Proof

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

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

We can't prove safety of $\text{tree}(r, h)$!

Specification

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

$\text{deletetree}(t)$

$\{h' : \forall p, h'[p] = h[p]$

if p is not in the tree $\}$

Proof

$\{h : h[t] = [l, r]$

$\wedge \text{tree}(l, h)$

$\wedge \text{tree}(r, h)$

$\wedge t, l, r \text{ distinct}\}$

$\text{deletetree}(l)$

Specification

$$\{h : \text{tree}(t, h)\}$$
$$\text{deletetree}(t)$$
$$\{h' : \forall p, h'[p] = h[p]$$
$$\text{if } p \text{ is not in the tree } \}$$

Proof

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

How can we be sure that

$\text{deletetree}(l)$

does not modify $\text{tree}(r, h)$? We should say that in $\text{tree}(t, h)$...

Outline

- 1 Introduction
- 2 Theoretical Foundations**
- 3 Extension to concurrency
- 4 Biabduction
- 5 Tools

Work on Separation Logic began in the late 90's and was focused on providing a simpler model for proofs of programs manipulating pointers.

Major contributors: **O'Hearn, Reynolds, Yang, Calcagno, Distefano, Brookes** et others

SL provides a language to describe properties and a proof system to validate such properties.

The model

$$\text{Ints} \triangleq \{\dots, -1, 0, 1, \dots\}$$

$$\text{Variables} \triangleq \{x, y, \dots\}$$

$$\text{Atoms}, \text{Locations} \subseteq \text{Ints}$$

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

$$\text{Stores} \triangleq \text{Variables} \rightarrow_{fin} \text{Ints}$$

$$\text{Heaps} \triangleq \text{Locations} \rightarrow_{fin} \text{Ints}$$

$$\text{States} \triangleq \text{Stores} \times \text{Heaps}$$

$$\llbracket E \rrbracket_s \in \text{Ints}, \llbracket B \rrbracket_s \in \{\text{true}, \text{false}\}$$

$$h \in \text{Heaps}, h[E] \in \text{Ints}$$

The language

Expressions:

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

The language

Expressions:

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

Assertions:

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

The language

Expressions:

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

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

Assertions:

$$P, Q, R ::= B \mid E \mapsto F \quad \textit{Atomic Formulae}$$

$$\mid false \mid P \Rightarrow Q \mid \forall x. P \quad \textit{Classical Logic}$$

$$\mid emp \mid P * Q \mid P - * Q \quad \textit{Spatial Connectives}$$

$$\neg P = P \Rightarrow False$$

$$true = \neg(false)$$

$$P \vee Q = \neg(P) \Rightarrow Q$$

The language

Expressions:

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

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

Assertions:

$$P, Q, R ::= B \mid E \mapsto F \quad \text{Atomic Formulae}$$

$$\mid false \mid P \Rightarrow Q \mid \forall x.P \quad \text{Classical Logic}$$

$$\mid emp \mid P * Q \mid P - * Q \quad \text{Spatial Connectives}$$

$$\neg P = P \Rightarrow False$$

$$true = \neg(false)$$

$$P \vee Q = \neg(P) \Rightarrow Q$$

$$P \wedge Q = \neg(\neg P \vee \neg Q)$$

$$\exists x.P = \neg \forall x. \neg P$$

Some notation

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

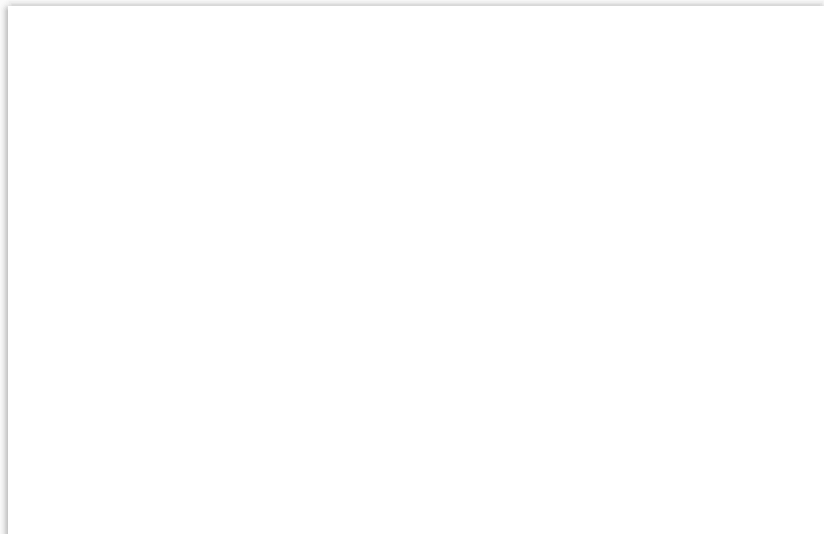
$$E \mapsto F_0, \dots, F_n \triangleq (E \mapsto F_0) * \dots * (E + n \mapsto F_n)$$

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

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

Semantics

For store s and heap h



For store s and heap h

$$s, h \models B \text{ iff } \llbracket B \rrbracket_s = \text{true}$$

$$s, h \models E \mapsto F \text{ iff } \{\llbracket E \rrbracket_s\} = \text{dom}(h) \text{ and } h(\llbracket E \rrbracket_s) = \llbracket F \rrbracket_s$$

$$s, h \models \text{false} \quad \text{never}$$

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

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

For store s and heap h

$$s, h \models B \text{ iff } \llbracket B \rrbracket_s = \text{true}$$

$$s, h \models E \mapsto F \text{ iff } \{\llbracket E \rrbracket_s\} = \text{dom}(h) \text{ and } h(\llbracket E \rrbracket_s) = \llbracket F \rrbracket_s$$

$$s, h \models \text{false} \quad \text{never}$$

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

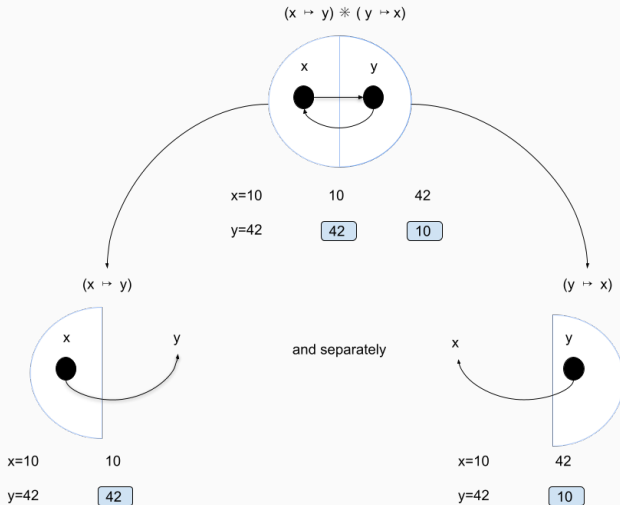
$$s, h \models \forall x. P \text{ iff } \forall v \in \text{Ints}. [s \mid x \mapsto v], h \models P$$

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

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

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

Visual example



Proof rules in separation logic are divided in:

- 1 Axioms for basic mutation commands \rightarrow *Small axioms*
- 2 Inference rules for modular reasoning \rightarrow *Structural rules*

Small axioms

Small axioms

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

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")

$\{x \doteq m\}\mathbf{x} := \mathbf{cons}(\mathbf{E}_1, \dots, \mathbf{E}_k)\{x \mapsto E_1[m/x], \dots, E_k[m/x]\}$
("Allocate memory")

Small axioms

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

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

$\{x \doteq m\}\mathbf{x} := \mathbf{cons}(\mathbf{E}_1, \dots, \mathbf{E}_k)\{x \mapsto E_1[m/x], \dots, E_k[m/x]\}$
("Allocate memory")

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

Small axioms

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

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

$$\{x \doteq m\}\mathbf{x} := \mathbf{cons}(\mathbf{E}_1, \dots, \mathbf{E}_k)\{x \mapsto E_1[m/x], \dots, E_k[m/x]\} \\ \text{("Allocate memory")}$$

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

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

Structural rules

Frame Rule

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

Structural rules

Frame Rule

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

Auxiliary variable elimination

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

Structural rules

Variable substitution

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

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

Structural rules

Variable substitution

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

x_i free and if $x_i \in \text{Mod}(C)$ 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 $x \notin \text{Free}(E_1, \dots, E_k)$.

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

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 $x \notin \text{Free}(E_1, \dots, E_k)$.

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

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:

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

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

Proof:

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

Proof:

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

deletetree(left);

Proof:

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

deletetree(left);

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

Proof:

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

deletetree(left);

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

deletetree(right);

Proof:

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

deletetree(left);

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

deletetree(right);

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

Proof:

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

$\textcolor{black}{deletetree}(\textcolor{black}{left});$

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

$\textcolor{black}{deletetree}(\textcolor{black}{right});$

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

$\textcolor{black}{free}(\textcolor{black}{root});$

Proof:

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

$\text{deletetree}(\text{left});$

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

$\text{deletetree}(\text{right});$

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

$\text{free}(\text{root});$

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

$\{\text{emp}\}$

Outline

- 1 Introduction
- 2 Theoretical Foundations
- 3 Extension to concurrency**
- 4 Biabduction
- 5 Tools

Concurrent separation logic

While Separation Logic was worked on primarily as a logic to reason about pointer manipulation, it has been successfully extended to handle concurrency of processes sharing some "resources" (usually memory).

Since 2002, work by Brookes, O'Hearn and others [1] has been fundamental for the advancement both in theory and applications of SL.
Here we will see the fundamentals aspect of this extension.

The new rules

There are two main new rules in CSL (Concurrent Separation Logic).

Parallel composition rule

$$\frac{\{P_1\}C_1\{Q_1\} \cdots \{P_n\}C_n\{Q_n\}}{\{P_1 * \cdots P_n\}C_1 || \cdots || C_n\{Q_1 * Q_n\}}$$

The new rules

There are two main new rules in CSL (Concurrent Separation Logic).

Parallel composition rule

$$\frac{\{P_1\}C_1\{Q_1\} \cdots \{P_n\}C_n\{Q_n\}}{\{P_1 * \cdots P_n\}C_1 || \cdots || C_n\{Q_1 * Q_n\}}$$

Critical region Rule

$$\frac{\{(P * RI_r) \wedge B\}C\{Q * RI_r\}}{\{P\} \text{ with } r \text{ when } B \text{ do } C \{Q\}}$$

RI_r is a "resource invariant" given to each resource r appearing in the program.

Proof outline for parallel mergeSort

$$\{array(a, i, m) * array(a, m + 1, j)\}$$

Proof outline for parallel mergeSort

$$\{array(a, i, m) * array(a, m + 1, j)\}$$

$$\{array(a, i, m)\}$$

$$\{array(a, m + 1, j)\}$$

Proof outline for parallel mergeSort

$$\{array(a, i, m) * array(a, m + 1, j)\}$$

$$\begin{array}{l} \{array(a, i, m)\} \\ ms(a, i, m) \end{array}$$

$$\begin{array}{l} \{array(a, m + 1, j)\} \\ ms(a, m + 1, j) \end{array}$$

Proof outline for parallel mergeSort

$$\{array(a, i, m) * array(a, m + 1, j)\}$$

$$\{array(a, i, m)\}$$

$$ms(a, i, m)$$

$$\{sorted(a, i, m)\}$$

$$\{array(a, m + 1, j)\}$$

$$ms(a, m + 1, j)$$

$$\{sorted(a, m + 1, j)\}$$

Proof outline for parallel mergeSort

$$\{array(a, i, m) * array(a, m + 1, j)\}$$

$$\{array(a, i, m)\}$$

$$ms(a, i, m)$$

$$\{sorted(a, i, m)\}$$

$$\{array(a, m + 1, j)\}$$

$$ms(a, m + 1, j)$$

$$\{sorted(a, m + 1, j)\}$$

$$\{sorted(a, i, m) * sorted(a, m + 1, j)\}$$

Based on two principles:

- *Ownership hypothesis.* A code fragment can access only those portions of state that it owns.
- *Separation Property.* At any time, the state can be partitioned into that owned by each process and each grouping of mutual exclusion

The key features of CSL semantics are:

- a compositional action-trace semantics
- a race detecting interpretation of parallel composition
- a global state interpretation of actions and traces
- a local-state interpretation of action and traces, that enables the formalization of ownership transfer and separation principle

Outline

- 1 Introduction
- 2 Theoretical Foundations
- 3 Extension to concurrency
- 4 Biabduction**
- 5 Tools

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**

$$(A * \textcolor{red}{?anti\textit{frame}} \vdash B * \textcolor{blue}{?frame})$$

*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 $r1 \mapsto \text{open}$.

The equivalent biabduction question is

$$(emp * \textcolor{red}{?anti\text{frame}} \vdash r1 \mapsto open * \textcolor{blue}{?frame})$$

Biabduction in action

$$(emp * ?\textit{anti\textcolor{red}{frame}} \vdash r1 \mapsto open * ?\textit{\textcolor{blue}{frame}})$$

Biabduction in action

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

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

Biabduction in action

$$(emp * \textcolor{red}{?anti\ frame} \vdash r1 \mapsto open * \textcolor{blue}{?frame})$$
$$(\textcolor{red}{anti\ frame} = r1 \mapsto open), (\textcolor{blue}{frame} = emp)$$
$$(emp * r1 \mapsto open \vdash r1 \mapsto open * emp)$$
$$(r1 \mapsto open \vdash r1 \mapsto open)$$

Biabduction in action

$$(emp * ?\textit{anti frame} \vdash r1 \mapsto open * ?\textit{frame})$$
$$(\textit{anti frame} = r1 \mapsto open), (\textit{frame} = emp)$$
$$(emp * r1 \mapsto open \vdash r1 \mapsto open * emp)$$
$$(r1 \mapsto open \vdash r1 \mapsto open)$$
$$\{\textit{r1} \mapsto \textit{open}\}$$
$$(\textit{closeResource}(r1))$$
$$\{\textit{r1} \mapsto \textit{closed}\}$$

Biabduction in action

$(emp * ?\text{anti frame} \vdash r1 \mapsto open * ?\text{frame})$

$(\text{anti frame} = r1 \mapsto open), (\text{frame} = emp)$

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

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

$\{r1 \mapsto open\}$

$(closeResource(r1))$

$\{r1 \mapsto closed\}$

Let's now consider $closeResource(r2)$

Biabduction in action

$$(r1 \mapsto \textit{closed} * \textcolor{red}{?anti\textit{frame}} \vdash r2 \mapsto \textit{open} * \textcolor{blue}{?frame})$$

Biabduction in action

$(r1 \mapsto \text{closed} * \textcolor{red}{?anti\ frame} \vdash r2 \mapsto \text{open} * \textcolor{blue}{?frame})$

$(\textcolor{red}{anti\ frame} = r2 \mapsto \text{open}), (\textcolor{blue}{frame} = r1 \mapsto \text{closed})$

Biabduction in action

$(r1 \mapsto \text{closed} * \text{?anti\textit{frame}} \vdash r2 \mapsto \text{open} * \text{?frame})$

$(\text{anti\textit{frame}} = r2 \mapsto \text{open}), (\text{frame} = r1 \mapsto \text{closed})$

$(\{r1 \mapsto \text{open} * r2 \mapsto \text{open}\})$

$(\text{closeResource}(r1))$

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

$(\text{closeResource}(r2))$

$\{r1 \mapsto \text{closed}, r2 \mapsto \text{closed}\}$

- Abstract interpretation:

Compositional Shape Analysis by means of Bi-Abduction [2]

- Model checking:

Model Checking for Symbolic-Heap Separation Logic with Inductive Predicates [3]

Outline

- 1 Introduction
- 2 Theoretical Foundations
- 3 Extension to concurrency
- 4 Biabduction
- 5 Tools

The first formal verification tool to make use of SL

It takes an input language that describes procedures together with pre and post conditions and discover proofs through symbolic execution

Although the tool is more demonstrative than practical, it was fundamental for future work, which were heavily inspired by the design and proof system of Smallfoot




An open source tool by Microsoft that uses separation logic to prove memory safety of C programs.




Inspired by the work on Smallfoot, it doesn't require any manual specification but works directly on code.

Infer is the static analysis tool used at Facebook.
It takes Java/C/C++/Obj-C code in inputs and produces
a list of potential bugs

It exploits SL locality and compositionality, and works on
source *diffs* instead of the entire codebase



-  Stephen Brookes and Peter W O'Hearn.
Concurrent separation logic.
ACM SIGLOG News, 3(3):47–65, 2016.
-  Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang.
Compositional shape analysis by means of bi-abduction.
In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 289–300, 2009.
-  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.

-  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.



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.