An Introduction to Separation Logic

Cristina SERBAN

December 10, 2014
Outline

1 Introduction
   - Motivation
   - Background overview
   - Proof example

2 The programming language
   - Syntax
   - Semantics

3 Assertions

4 Inference rules
   - Frame rule

5 Lists

6 Trees

7 Applications
Why do we need separation logic?

- Reasoning about **shared mutable data structures** - where updateable fields can be referenced from more than one point
- Supports **local reasoning** - specifications and proofs for a program component refer only to the memory that component accesses
- Previous methods
  - limited applicability
  - extreme complexity
  - scale poorly

  Reason: correctness depends upon complex restrictions on sharing

- Extended for **shared-variable concurrency** and **information hiding**
- Notion of **separation** - applicable to a wider conceptual range
  - permission to exercise capabilities or knowledge of structure instead of access to memory
What is separation logic?

- Modern system that extends **Hoare logic** and allows reasoning about the state of the heap
- Assertion language of separation logic is a special case of the logic of bunched implications (BI) - variety of substructural logic, with a tree-like context of hypotheses
Background overview

- **Propositional logic**
  Describes simple *propositions* (i.e. sentences) and the basic connectives for forming more complex sentences.

\[
\neg \quad \land \quad \lor \quad \Rightarrow \\
\neg p \quad p \land q \quad \neg p \lor q \quad p \Rightarrow q
\]

- **First-order logic**
  Extends propositional logic by adding *variables* and *quantifiers*. We can also define n-ary *predicates* and functions.

\[
\forall \text{ (for all)} \quad \exists \text{ (exists)} \\
P(x) \Rightarrow Q(x) \quad \exists x.(P(x) \land Q(x))
\]
Background overview

- **Hoare logic** (also known as Floyd-Hoare logic)
  Proofs for the correctness of computer programs using Hoare triples.

\[
\{ P \} \ c \ \{ Q \}
\]

Problematic when the heap is introduced - e.g. assignment rule:

\[
\{P[e/x]\} \ x := e \ {P}
\]

\[
\begin{align*}
\checkmark & \quad \{1 = 1\} \quad x := 1 \quad \{x = 1\} \\
\times & \quad \{?\} \quad [x] := 1 \quad \{[x] = 1\}
\end{align*}
\]

 Preconditions have to explicitly specify that a set of addresses do not overlap with the addresses affected by a command.

\[
\{x \mapsto - \land [y] = 2 \land x \neq y\} \ [x] := 1 \ \{[x] = 1 \land [y] = 2 \land x \neq y\}
\]
Background overview

- **Separation logic**
  Simplifies the problem of specifying preconditions by introducing two new logical connectives:
  - $\ast$: separating conjunction
  - $\Rightarrow\ast$: separating implication

  Now we can write:
  \[
  \{ x \mapsto \neg \ast y \mapsto 2 \} \ [x] := 1 \ \{ x \mapsto 1 \ast y \mapsto 2 \} \ \text{(* has } x \neq y \text{ built into it)}
  \]

  Also introduces a new value `emp` such that $P \ast emp \iff P$.

  $\ast$ is different from classic conjunction:
  \[
  \checkmark \quad \frac{p, q}{p \land q} \quad \text{(in propositional logic)} \quad \times \quad \frac{p, q}{p \ast q} \quad \text{(in separation logic)}
  \]

  Structural rules do not hold $\rightarrow$ *substructural logic*.
  \[
  \checkmark \quad p \Rightarrow p \ast p \quad \text{(contraction)} \quad \times \quad p \ast q \Rightarrow p \quad \text{(weakening)}
  \]
Proof example

- Constructing a 2-element cyclic structure with relative addresses

\[
\begin{align*}
\{ & \text{emp} \} \\
\{ & x := \text{cons}(a,a); \} \\
\{ & x \mapsto a,a \} \\
\{ & y := \text{cons}(b,b); \} \\
\{ & x \mapsto a,a \times y \mapsto b,b \} \\
\{ & x \mapsto a,- \times y \mapsto b,- \} \\
\{ & x + 1 \} := y - x; \quad \{ & x \mapsto a,y-x \times y \mapsto b,- \} \\
\{ & y + 1 \} := x - y; \quad \{ & x \mapsto a,y-x \times y \mapsto b,x-y \} \\
\{ & \exists o. x \mapsto a,o \times x+o \mapsto b,-o \}
\end{align*}
\]
Syntax of the programming language

- The simple imperative language originally axiomatized by Hoare, with new commands for manipulating shared mutable data structures

Syntax of new commands

\[
\langle \text{comm} \rangle ::= \ldots
\]

\[
| \langle \text{var} \rangle ::= \text{cons}(\langle \text{exp} \rangle, \ldots, \langle \text{exp} \rangle) \quad \text{allocation}
\]

\[
| \langle \text{var} \rangle ::= [\langle \text{exp} \rangle] \quad \text{lookup}
\]

\[
| [\langle \text{exp} \rangle] ::= \langle \text{exp} \rangle \quad \text{mutation}
\]

\[
| \text{dispose} \langle \text{exp} \rangle \quad \text{deallocation}
\]
Semantics of the programming language

- Computational states contain:
  - A store (stack) - maps variables into values
  - A heap - maps addresses into values

- Unrestricted address arithmetic
  - All values are integers, an infinite number of which are addresses
  - Atoms are integers that are not addresses
  - Heaps map addresses into single values
Semantics of the programming language

\[
\text{Values} = \text{Integers}
\]

\[
\text{Atoms} \cup \text{Addresses} \subseteq \text{Integers}
\]

\[
\text{Atoms} \cap \text{Addresses} = \emptyset
\]

\[
\text{nil} \in \text{Atoms}
\]

\[
\text{Stores}_V = V \rightarrow \text{Values}
\]

\[
\text{Heaps} = \bigcup_{A \subseteq \text{Addresses}}^{\text{fin}} (A \rightarrow \text{Values})
\]

\[
\text{States}_V = \text{Stores}_V \times \text{Heaps}
\]

(\text{where } V \text{ is a finite set of variables})
Semantics of the programming language

Semantics of ordinary and boolean expressions

\[
\begin{align*}
\llbracket e \in \langle \exp \rangle \rrbracket_{\exp} &\in (\bigcup_{V \ni FV(e)}^{\text{fin}} \text{Stores}_V) \rightarrow \text{Values} \\
\llbracket b \in \langle \text{boolexp} \rangle \rrbracket_{\text{bexp}} &\in (\bigcup_{V \ni FV(e)}^{\text{fin}} \text{Stores}_V) \rightarrow \{\text{true, false}\}
\end{align*}
\]

(where \( FV(p) \) is the set of variables occurring free in the phrase \( p \))

- Expressions do not depend upon the heap
  \( \Rightarrow \) always well-defined and never cause side-effects
Semantics of the programming language

- The operation \texttt{cons}(e_1, ..., e_n) activates and initializes \textit{n consecutive} and previously inactive cells in the heap.
- The other operations cause memory faults (i.e. terminal configuration \texttt{abort}) if an inactive address is dereferenced or deallocated.
New assertions describing the heap

- **emp**  
  The heap is empty  

- e → e'  
  The heap contains one cell, at address e with contents e'

- p₁ * p₂  
  The heap can be split into two disjoint parts such that p₁ holds for one part and p₂ holds for the other

- p₁ →* p₂  
  If the heap is extended with a disjoint part in which p₁ holds, then p₂ holds for the extended heap
Assertions

Abbreviations for assertions

\[ e \leftrightarrow e' \overset{\text{def}}{=} e \mapsto e' \overset{\ast}{\mapsto} \text{true} \]

(e points to \( e' \) somewhere in the heap)

\[ e \mapsto e_1, \ldots, e_n \overset{\text{def}}{=} e \mapsto e_1 \ast e + 1 \mapsto e_2 \ast \ldots \ast e + n - 1 \mapsto e_n \]

(e points to a record with several fields)

\[ e \leftrightarrow e_1, \ldots, e_n \overset{\text{def}}{=} e \leftrightarrow e_1 \ast e + 1 \leftrightarrow e_2 \ast \ldots \ast e + n - 1 \leftrightarrow e_n \]

iff \( e \mapsto e_1, \ldots, e_n \ast \text{true} \)
Sharing patterns

1. $x \mapsto 3, y$

2. $y \mapsto 3, x$

3. $x \mapsto 3, y \ast y \mapsto 3, x$

4. $x \mapsto 3, y \land y \mapsto 3, x$

5. $x \mapsto 3, y \land y \mapsto 3, x$

Either (3) or (4) may hold and the heap may contain additional cells.
Meaning of assertions

If $s$ is a store, $h$ is a heap, $p$ is an assertion and $\text{FV}(p) \in \text{dom} \ s$, then

$$s \mathbin{\models} h \models p$$

means that the state $s, h$ satisfies $p$, or $p$ is true in $s, h$, or $p$ holds in $s, h$.

$s, h \models p$ defined by induction on the shape of $p$

- $s, h \models b$ iff $\llbracket b \rrbracket_{\text{bexp}} s = \text{true}$
- $s, h \models \neg p$ iff $s, h \models p$ is false
- $s, h \models p_0 \land p_1$ iff $s, h \models p_0$ and $s, h \models p_1$
- $s, h \models \forall v. p$ iff $\forall x \in \mathbb{Z}. [s \mid v : x], h \models p$
- $s, h \models \exists v. p$ iff $\exists x \in \mathbb{Z}. [s \mid v : x], h \models p$ (similarly for $\lor$, $\Rightarrow$, $\Leftrightarrow$)
Meaning of assertions

$s, h \models p$ defined by induction on the shape of $p$ (continued)

$s, h \models \text{emp} \iff \text{dom } h = \{\}$
Meaning of assertions

$s, h \models p$ defined by induction on the shape of $p$ (continued)

$s, h \models e \leftrightarrow e'$ iff $\text{dom } h = \{ [e]_{\exp s} \}$ and $h([e]_{\exp s}) = [e']_{\exp s}$
Meaning of assertions

$s, h \models p$ defined by induction on the shape of $p$ (continued)

$h_0 \perp h_1 \equiv h_0$ and $h_1$ are heaps with disjoint domains

$h_0 \cdot h_1 \equiv$ union of heaps with disjoint domains

$s, h \models p_0 \ast p_1$ iff $\exists h_0, h_1. h_0 \perp h_1$ and $h_0 \cdot h_1 = h$

and $s, h_0 \models p_0$ and $s, h_1 \models p_1$
Meaning of assertions

\( s, h \models p \) defined by induction on the shape of \( p \) (continued)

- \( h_0 \perp h_1 \equiv h_0 \) and \( h_1 \) are heaps with disjoint domains
- \( h_0 \cdot h_1 \equiv \) union of heaps with disjoint domains

\[
s, h \models p_0 \mathcal{F} p_1 \iff \forall h'. (h' \perp h \text{ and } s, h' \models p_0) \text{ implies } s, h \cdot h' \models p_1
\]
Pure and spatial assertions

- **Pure assertions** are independent of the heap.

  An assertion \( p \) is pure iff, for all stores \( s \) and all heaps \( h \) and \( h' \),
  \[
  s, h \models p \iff s, h' \models p
  \]

  Syntactic criteria: a pure assertion does not contain `emp`, \( \rightarrow \), \( \hookrightarrow \).

  The distinction between separating and ordinary operations collapses:
  \( \ast \) and \( \land \) are interchangeable, as are \( \lnot \ast \) and \( \Rightarrow \).

- **Spatial assertions** depend on the heap.

  Example:
  \[
  \exists x \exists y. x \mapsto y
  \]
  This formula is true on a heap with a single allocated cell.

  This type of formulas become false when extending the heap.
Relationship between $\ast$ and $\neg\ast$

- Resource interpretation of the connectives:
  - $\ast$ decomposes current resources
  - $\neg\ast$ talks about new or fresh resources

- Adjunctive relationship:
  \[ s, h \cup h' \models P \ast Q \Rightarrow R \text{ iff } s, h \models P \Rightarrow Q \neg\ast R, \text{ where } h \perp h' \]

- Similarity to modus ponens:
  \[
  \frac{s, h \models P \ast (P \neg\ast Q)}{s, h \models Q}
  \]
Inference rules

- The *command-specific inference rules* of Hoare logic remain **sound**, as do the following *structural rules*:

\[
\begin{align*}
\frac{p \Rightarrow q}{\{p\} \ c \ \{r\}} & \quad \frac{\{p\} \ c \ \{q\}}{q \Rightarrow r} \\
\{p\} \ c \ \{r\} & \quad \{p\} \ c \ \{r\}
\end{align*}
\]  

(Strengthening Precedent)  
(Weakening Consequent)

\[
\begin{align*}
\frac{\{p\} \ c \ \{q\}}{\exists v. \ p} & \quad \frac{\exists v. \ p \ c \ \{\exists v. \ q\}}{\exists v. \ p \ c \ \{\exists v. \ q\}} \quad \text{where } v \text{ is not free in } c
\end{align*}
\]  

(Existential Quantification)

\[
\frac{\{p\} \ c \ \{q\}}{\{p\} \ c \ \{q_1 \land q_2\}} \quad \frac{\{p\} \ c \ \{q_1\} \quad \{p\} \ c \ \{q_2\}}{\{p\} \ c \ \{q_1 \land q_2\}}
\]  

(Conjunction)

\[
\begin{align*}
\frac{\{p\} \ c \ \{q\}}{\{p/\delta\} \ (c/\delta) \ \{q/\delta\}}
\end{align*}
\]  

where \(\delta\) is the substitution \(\nu_1 \rightarrow e_1, ..., \nu_n \rightarrow e_n\), \(\nu_1, ..., \nu_n\) are the variables occurring free in \(p, c,\) or \(q\), and, if \(\nu_i\) is modified by \(c\), then \(e_i\) is a variable that does not occur free in any other \(e_j\)  

(Substitution)
Frame rule

- Rule of constancy, vital for scalability:

\[
\frac{\{p\} \; c \; \{q\}}{\{p \land r\} \; c \; \{q \land r\}}
\]

(where no free variable occurring in \(r\) is modified by \(c\))

Becomes **unsound** when moving to separation logic.

- The ability to extend local specifications can be regained at a deeper level using the separating conjunction:

\[
\frac{\{p\} \; c \; \{q\}}{\{p \ast r\} \; c \; \{q \ast r\}}
\]

(where no free variable occurring in \(r\) is modified by \(c\))

Extend local specification, involving only variables and *heap cells* that may actually be used by \(c\) (i.e. its *footprint*), by adding predicates about variables and heap cells not modified or mutated by \(c\).
Inference rules

- **Mutation (local)**
  \[
  \{e \mapsto \_\} [e] := e' \{e \mapsto e'\}
  \]

- **Mutation (global)**
  \[
  \{(e \mapsto \_) \ast r\} [e] := e' \{(e \mapsto e') \ast r\}
  \]

- **Mutation (backwards reasoning)**
  \[
  \{(e \mapsto \_) \ast ((e \mapsto e') \ast p)\} [e] := e' \{p\}
  \]

\[
\{(e \mapsto \_) \ast ((e \mapsto e') \ast p)\}
\{\exists x. (e \mapsto x) \ast ((e \mapsto e') \ast p)\}
[e] := e'
\{\exists x. (e \mapsto e') \ast ((e \mapsto e') \ast p)\}
\{\exists x. p\}
\{p\}
\]
Inference rules

- Deallocation (local)
  \[
  \{ e \mapssto - \} \text{dispose } e \{ \text{emp} \}
  \]

- Deallocation (global, backwards reasoning)
  \[
  \{(e \mapssto -) \ast r\} \text{dispose } e \{ r \}
  \]

- Allocation (nonoverwriting, local)
  \[
  \{ \text{emp} \} \nu := \text{cons}(\bar{e}) \{ \nu \mapssto \bar{e} \}
  \]
  where \( \nu \) is not free in \( \bar{e} \).

- Allocation (nonoverwriting, global)
  \[
  \{ r \} \nu := \text{cons}(\bar{e}) \{(\nu \mapssto \bar{e}) \ast r\}
  \]
  where \( \nu \) is not free in \( \bar{e} \) or \( r \).
Singly-linked lists

- **Singly-linked list**: list $\alpha$ $i \iff i$ is a list representing the sequence $\alpha$

\[
\begin{array}{cccccc}
& \alpha_1 & \rightarrow & \alpha_2 & \rightarrow & \alpha_3 & \rightarrow & \ldots & \rightarrow & \alpha_{n-1} & \rightarrow & \alpha_n \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\
& \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ & \\
\end{array}
\]

- list $\epsilon$ $i \overset{\text{def}}{=} \text{emp} \land i = \text{nil}$

- list $(a \cdot \alpha)$ $i \overset{\text{def}}{=} \exists j. i \mapsto a,j \ast \text{list} \alpha j$
Singly-linked list segments

- **Singly-linked list segment**: \( \text{lseg } \alpha (i,j) \iff i \text{ to } j \text{ is a list segment representing the sequence } \alpha \)

\[
\begin{align*}
\text{lseg } \epsilon (i,j) & \overset{\text{def}}{=} \text{emp} \land i = j \\
\text{lseg } a \cdot \alpha (i,k) & \overset{\text{def}}{=} \exists j. \ i \mapsto a,j \land \text{lseg } \alpha (j,k)
\end{align*}
\]

\[
\begin{align*}
\text{lseg } a (i,j) & \iff i \mapsto a,j \\
\text{lseg } \alpha \cdot \beta (i,k) & \iff \exists j. \ \text{lseg } \alpha (i,j) \land \text{lseg } \beta (j,k) \\
\text{lseg } \alpha \cdot \text{b} (i,k) & \iff \exists j. \ \text{lseg } \alpha (i,j) \land j \mapsto b,k \\
\text{list } \alpha i & \iff \text{lseg } \alpha (i,\text{nil})
\end{align*}
\]
Singly-linked lists

- Deleting an element at the beginning of a non-empty list segment

\[
\begin{align*}
\{ \text{lseg } a \cdot \alpha (i,k) \} \\
\{ \exists j. \ i \rightarrow a,j \ast \text{lseg } \alpha (j,k) \} \\
\{ \exists j. \ i + 1 \rightarrow j \ast (i \rightarrow a \ast \text{lseg } \alpha (j,k)) \} \\
\text{j := } [i + 1]; \\
\{ i + 1 \rightarrow j \ast (i \rightarrow a \ast \text{lseg } \alpha (j,k)) \} \\
\{ i \rightarrow a \ast (i + 1 \rightarrow j \ast \text{lseg } \alpha (j,k)) \} \\
\text{dispose } i; \\
\{ i + 1 \rightarrow j \ast \text{lseg } \alpha (j,k) \} \\
\text{dispose } i + 1; \\
\{ \text{lseg } \alpha (j,k) \} \\
i := j; \\
\{ \text{lseg } \alpha (i,k) \}
\end{align*}
\]
Doubly-linked list segments

- **Doubly-linked list segment**: \( \text{dlseg } \alpha \ (i,i',j,j') \)

\[
dlseg \epsilon \ (i,i',j,j') \quad \text{def} \quad \text{emp} \land i = j \land i' = j'
\]

\[
dlseg a \cdot \alpha \ (i,i',k,k') \quad \text{def} \quad \exists j. \ i \mapsto a,j,i' \ast \text{dlseg } \alpha \ (j,i,k,k')
\]

\[
dlseg a \ (i,i',j,j') \iff i \mapsto a,j,i' \land i = j'
\]

\[
dlseg \alpha \cdot \beta \ (i,i',k,k') \iff \exists j,j'. \ \text{dlseg } \alpha \ (i,i',j,j') \ast \text{dlseg } \beta \ (j,j',k,k')
\]

\[
dlseg \alpha \cdot b \ (i,i',k,k') \iff \exists j'. \ \text{dlseg } \alpha \ (i,i',k',j') \ast k' \mapsto b,k,j'
\]
Doubly-linked lists

- A doubly-linked list can be defined by

\[ \text{dlist } \alpha \ (i,j') = \text{dlseg } \alpha \ (i,\text{nil},\text{nil},j') \]
Trees

S-expressions

The set $S$-exp of S-expressions is the least set such that

\[ \tau \in S\text{-exp} \iff \tau \in Atoms \]

or \( \tau = \tau_1 \cdot \tau_2 \) where \( \tau_1, \tau_2 \in S\text{-exp} \)

Trees

A tree is a representation of S-expressions by two-field records, in which there is no sharing between the representation of subexpressions.

Defining the predicate tree $\tau \ (i)$ (i.e. $i$ is the root of a tree representing the S-expression $\tau$) by structural induction on $\tau$:

\[
\begin{align*}
\text{tree } a \ (i) & \iff \text{emp} \land i=a \\
\text{tree } (\tau_1 \cdot \tau_2) \ (i) & \iff \exists i_1, i_2. \ i \mapsto i_1, i_2 \ast \text{tree } \tau_1 \ (i_1) \ast \text{tree } \tau_2 \ (i_2)
\end{align*}
\]
Proof for \{\text{tree } \tau\ (i)\} \ \text{copytree}(i;j) \ \{\text{tree } \tau\ (i) \ast \text{tree } \tau\ (j)\}

\begin{align*}
\{\text{tree } \tau\ (i)\} \\
\textbf{if} \ \text{isatom}(i) \ \textbf{then} \\
\quad \{\text{isatom}(\tau) \land \textbf{emp} \land i = \tau\} \\
\quad \{\text{isatom}(\tau) \land ((\textbf{emp} \land i = \tau) \ast (\textbf{emp} \land i = \tau))\} \\
\quad j := i; \\
\quad \{\text{isatom}(\tau) \land ((\textbf{emp} \land i = \tau) \ast (\textbf{emp} \land j = \tau))\} \\
\textbf{else} \\
\quad \ldots
\end{align*}

\begin{align*}
\{\text{tree } \tau\ (i) \ast \text{tree } \tau\ (j)\}
\end{align*}
Proof for \{tree \tau (i)\} copytree(i;j) \{tree \tau (i) \ast tree \tau (j)\}

\begin{align*}
\{tree \tau (i)\} \\
\text{if isatom}(i) \text{ then } & \ldots \\
\text{else} \\
\{\exists \tau_1, \tau_2. \tau = (\tau_1 \cdot \tau_2) \land \text{tree} \ (\tau_1 \cdot \tau_2) \ (i)\} \\
\text{newvar \ } i_1, i_2, j_1, j_2 \text{ in} \\
& (i_1 := [i]; i_2 := [i+1]; \\
& \{\exists \tau_1, \tau_2. \tau = (\tau_1 \cdot \tau_2) \land (i \mapsto i_1, i_2 \ast \text{tree} \ \tau_1 \ (i_1) \ast \text{tree} \ \tau_2 \ (i_2))\} \\
& \text{copytree}(i_1;j_1); \\
& \{\exists \tau_1, \tau_2. \tau = (\tau_1 \cdot \tau_2) \land (i \mapsto i_1, i_2 \ast \text{tree} \ \tau_1 \ (i_1) \ast \text{tree} \ \tau_2 \ (i_2) \ast \text{tree} \ \tau_1 \ (j_1))\} \\
& \text{copytree}(i_2;j_2); \\
& \{\exists \tau_1, \tau_2. \tau = (\tau_1 \cdot \tau_2) \land \\
& \quad (i \mapsto i_1, i_2 \ast \text{tree} \ \tau_1 \ (i_1) \ast \text{tree} \ \tau_2 \ (i_2) \ast \text{tree} \ \tau_1 \ (j_1) \ast \text{tree} \ \tau_2 \ (j_2))\} \\
& j := \text{cons}(j_1,j_2); \\
& \{\exists \tau_1, \tau_2. \tau = (\tau_1 \cdot \tau_2) \land \\
& \quad (i \mapsto i_1, i_2 \ast \text{tree} \ \tau_1 \ (i_1) \ast \text{tree} \ \tau_2 \ (i_2)) \ast \\
& \quad j \mapsto j_1, j_2 \ast \text{tree} \ \tau_1 \ (j_1) \ast \text{tree} \ \tau_2 \ (j_2))\} \\
& \{\exists \tau_1, \tau_2. \tau = (\tau_1 \cdot \tau_2) \land \text{tree} \ (\tau_1 \cdot \tau_2) \ (i) \ast \text{tree} \ (\tau_1 \cdot \tau_2) \ (j)\}\} \\
\{tree \ \tau \ (i) \ast tree \ \tau \ (j)\}
\end{align*}
Proof for the recursive calls of copytree

1. \{tree \tau (i)\} \text{ copytree}(i;j) \{tree \tau (i) \ast tree \tau (j)\}

2. \{tree \tau_1 (i_1)\} \text{ copytree}(i_1;j_1) \{tree \tau_1 (i_1) \ast tree \tau_1 (j_1)\}  \quad \text{(substitution)}

3. \{(\tau = (\tau_1 \cdot \tau_2) \land i \mapsto i_1,i_2) \ast tree \tau_1 (i_1) \ast tree \tau_2 (i_2)\}
   \text{ copytree}(i_1;j_1)
   \{(\tau = (\tau_1 \cdot \tau_2) \land i \mapsto i_1,i_2) \ast 
   tree \tau_1 (i_1) \ast tree \tau_2 (i_2) \ast tree \tau_1 (j_1)\} \quad \text{(frame rule)}

4. \{\tau = (\tau_1 \cdot \tau_2) \land (i \mapsto i_1,i_2 \ast tree \tau_1 (i_1) \ast tree \tau_2 (i_2))\}
   \text{ copytree}(i_1;j_1)
   \{\tau = (\tau_1 \cdot \tau_2) \land (i \mapsto i_1,i_2 \ast 
   tree \tau_1 (i_1) \ast tree \tau_2 (i_2) \ast tree \tau_1 (j_1))\} \quad \text{(consequence)}

5. \{\exists \tau_1, \tau_2. \tau = (\tau_1 \cdot \tau_2) \land (i \mapsto i_1,i_2 \ast tree \tau_1 (i_1) \ast tree \tau_2 (i_2))\}
   \text{ copytree}(i_1;j_1)
   \{\exists \tau_1, \tau_2. \tau = (\tau_1 \cdot \tau_2) \land (i \mapsto i_1,i_2 \ast 
   tree \tau_1 (i_1) \ast tree \tau_2 (i_2) \ast tree \tau_1 (j_1))\} \quad \text{(aux. var. elimination)}
Applications

Space Invader

- Cristiano Calcagno, Dino Distefano, Peter O’Hearn, Hongseok Yang
- Shape analysis - discovers the shapes of data structures at program points encountered during the execution of a program.
- Symbolic execution of separation logic formulae called symbolic heaps
- Abstract domain of linked lists + an abstraction or widening operator which converts a symbolic heap to a canonical form
- There are finitely many canonical terms $\Rightarrow$ termination of the fixed-point calculation in the abstract semantics of the while loops

Abduction

Given assumption $A$ and goal $G$, solve $A \ast ?? \vdash G$.

Bi-abduction

Given assumption $A$ and goal $G$, solve $A \ast ?\text{anti-frame} \vdash G \ast ?\text{frame}$.
Synthesizes both missing and additional, leftover portions of state.
Applications

**HIP/SLEEK**

- Wei-Ngan Chin, Huu Hai Nguyen, Cristina David, Cristian Gherghina, Quang Loc Le, Ton-Chanh Le, Asankhaya Sharma
- HIP - a separation logic based automated verification system
- SLEEK - a separation logic prover
- Conceived for deductive verification
- User-defined shape predicates allowing the description of a wide range of data structures together with their size properties
- *unfold/fold* reasoning for checking entailment of constraints
References


- Samin Ishtiaq and Peter O’Hearn, *Bi as an assertion language for mutable data structures*, Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL ’01)

References

- Dino Distefano, Peter W. O’Hearn, and Hongseok Yang, *A local shape analysis based on separation logic*, Proceedings of the 12th international conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’06)

- Cristiano Calcagno, Dino Distefano, Peter W. O’Hearn, and Hongseok Yang, *Compositional Shape Analysis by Means of Bi-Abduction*, J. ACM 58, 6, Article 26 (December 2011)

- Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin, *Automated verification of shape and size properties via separation logic*, Proceedings of the 8th international conference on Verification, model checking, and abstract interpretation (VMCAI’07)