



Semantics and Verification of Software

Summer Semester 2019

Lecture 18: Separation Logic II

(Recursive Data Structures and Partial Correctness Properties)

Thomas Noll

Software Modeling and Verification Group

RWTH Aachen University

<https://moves.rwth-aachen.de/teaching/ss-19/sv-sw/>

Recap: Pointer Programs and Separation Logic

Outline of Lecture 18

Recap: Pointer Programs and Separation Logic

Properties of Separating Conjunction

Recursive Data Structures

Partial Correctness Properties

The Proof System

Recap: Pointer Programs and Separation Logic

A Proper Rule for Local Reasoning

The **frame rule** of Separation Logic is:

$$\frac{\{A\} c \{B\} \quad FV(C) \cap Mod(c) = \emptyset}{\{A * C\} c \{B * C\}}$$

It allows to reason about the behaviour of c in any (heap) context C that is unaffected by the execution of c .

In particular,

$$\frac{\{x \mapsto 0\} [x] \quad := \quad 1 \quad \{x \mapsto 1\}}{\{x \mapsto 0 * y \mapsto 0\} [x] \quad := \quad 1 \quad \{x \mapsto 1 * y \mapsto 0\}}$$

is **valid** as $*$ excludes aliasing of x and y .

Recap: Pointer Programs and Separation Logic

Extending the Syntax of WHILE

Syntactic categories:

Category	Domain	Meta variable
Arithmetic expressions	$AExp$	a
Boolean expressions	$BExp$	b
Commands (statements)	Cmd	c

Context-free grammar:

$a ::= z \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \in AExp$

$b ::= t \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \in BExp$

$c ::= x := \text{alloc}(a_1, \dots, a_n) \mid \text{free}(a) \mid x := [a] \mid [a] := a' \mid \text{skip} \mid x := a \mid c_1; c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \text{ end} \mid \text{while } b \text{ do } c \text{ end} \in Cmd$

- $x := \text{alloc}(a_1, \dots, a_n)$ allocates $n \geq 1$ fresh addresses with initial values a_1, \dots, a_n
- $\text{free}(a)$ deallocates the address given by a
- $x := [a]$ stores the value at address a in variable x (lookup)
- $[a] := a'$ sets the value at address a to the value of a' (mutation)

Recap: Pointer Programs and Separation Logic

Stacks and Heaps

Approach: memory split into stack and heap:

- **stack** assigns values to (local) variables (just as previous program state)
- **heap** is array of memory controlled by program (allocation/deallocation)

Definition (Stacks and heaps)

- A **stack** is an element of the set

$$\text{Stack} := \{s \mid s : \text{Var} \rightarrow \mathbb{Z}\}$$

- A **heap** is an element of the set

$$\text{Heap} := \{h : \mathbb{N}_{>0} \dashrightarrow \mathbb{Z} \mid |\text{dom}(h)| < \infty\}$$

The **empty heap** is denoted by h_\emptyset (i.e., $\text{dom}(h_\emptyset) = \emptyset$).

- Two heaps $h_1, h_2 \in \text{Heap}$ are **disjoint** (notation: $h_1 \# h_2$) if $\text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset$. In this case, their **disjoint union** is given by

$$h_1 \uplus h_2 : \mathbb{N}_{>0} \dashrightarrow \mathbb{Z} : l \mapsto \begin{cases} h_1(l) & \text{if } l \in \text{dom}(h_1) \\ h_2(l) & \text{if } l \in \text{dom}(h_2) \end{cases}$$

- The set of **(program) states** is defined by

$$\Sigma := \text{Stack} \times \text{Heap}$$

Recap: Pointer Programs and Separation Logic

Execution of Statements I

Observation: values of (arithmetic and Boolean) expressions depend on the stack only

⇒ employ evaluation functions $\mathcal{A}[[a]] : Stack \rightarrow \mathbb{Z}$ and $\mathcal{B}[[b]] : Stack \rightarrow \mathbb{B}$

Definition (Execution relation for statements)

The **execution relation** $\rightarrow \subseteq (Cmd \times Stack \cup \{\downarrow\}) \times (Stack \cup \{\downarrow\})$ is defined by:

$$\text{(alloc)} \frac{l, l+1, \dots, l+n-1 \in \mathbb{N}_{>0} \setminus \text{dom}(h) \quad \mathcal{A}[[a_1]]s = z_1 \quad \dots \quad \mathcal{A}[[a_n]]s = z_n}{\langle x := \text{alloc}(a_1, \dots, a_n), (s, h) \rangle \rightarrow (s[x \mapsto l], h[l \mapsto z_1, \dots, l+n-1 \mapsto z_n])}$$

$$\text{(free)} \frac{\mathcal{A}[[a]]s = l \in \text{dom}(h)}{\langle \text{free}(a), (s, h) \rangle \rightarrow (s, h[l \mapsto \perp])}$$

$$\text{(lookup)} \frac{\mathcal{A}[[a]]s = l \in \text{dom}(h)}{\langle x := [a], (s, h) \rangle \rightarrow (s[x \mapsto h(l)], h)}$$

$$\text{(mutate)} \frac{\mathcal{A}[[a]]s = l \in \text{dom}(h) \quad \mathcal{A}[[a']]s = z}{\langle [a] := a', (s, h) \rangle \rightarrow (s, h[l \mapsto z])}$$

$$\text{(free-f)} \frac{\mathcal{A}[[a]]s \notin \text{dom}(h)}{\langle \text{free}(a), (s, h) \rangle \rightarrow \downarrow}$$

$$\text{(lookup-f)} \frac{\mathcal{A}[[a]]s \notin \text{dom}(h)}{\langle x := [a], (s, h) \rangle \rightarrow \downarrow}$$

$$\text{(mutate-f)} \frac{\mathcal{A}[[a]]s \notin \text{dom}(h)}{\langle [a] := a', (s, h) \rangle \rightarrow \downarrow}$$

$$\text{(fault)} \frac{}{\langle c, \downarrow \rangle \rightarrow \downarrow}$$

Recap: Pointer Programs and Separation Logic

Execution of Statements II

Definition (Execution relation for statements (continued))

$$\begin{array}{c} \text{(skip)} \frac{}{\langle \text{skip}, (s, h) \rangle \rightarrow (s, h)} \\ \text{(if-t)} \frac{\mathcal{B}[[b]]s = \text{true} \quad \langle c_1, (s, h) \rangle \rightarrow \sigma'}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2 \text{ end}, (s, h) \rangle \rightarrow \sigma'} \\ \text{(seq)} \frac{\langle c_1, \sigma \rangle \rightarrow \sigma' \quad \langle c_2, \sigma' \rangle \rightarrow \sigma''}{\langle c_1; c_2, \sigma \rangle \rightarrow \sigma''} \\ \text{(wh-t)} \frac{\mathcal{B}[[b]]s = \text{true} \quad \langle c, (s, h) \rangle \rightarrow \sigma' \quad \langle \text{while } b \text{ do } c \text{ end}, \sigma' \rangle \rightarrow \sigma''}{\langle \text{while } b \text{ do } c \text{ end}, (s, h) \rangle \rightarrow \sigma''} \end{array} \quad \begin{array}{c} \mathcal{A}[[a]]s = z \\ \text{(asgn)} \frac{}{\langle x := a, (s, h) \rangle \rightarrow (s[x \mapsto z], h)} \\ \text{(if-f)} \frac{\mathcal{B}[[b]]s = \text{false} \quad \langle c_2, (s, h) \rangle \rightarrow \sigma'}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2 \text{ end}, (s, h) \rangle \rightarrow \sigma'} \\ \text{(wh-f)} \frac{\mathcal{B}[[b]]s = \text{false}}{\langle \text{while } b \text{ do } c \text{ end}, (s, h) \rangle \rightarrow (s, h)} \end{array}$$

Remarks:

- Deallocation/lookup/mutation of unallocated heap addresses aborts execution and yields a memory fault (“ \perp ”).
- Allocation always succeeds (but yields non-deterministic results).
- Thus, for given $\langle c, (s, h) \rangle$ all outcomes (success/fault/non-termination) possible!

Recap: Pointer Programs and Separation Logic

Syntax of Separation Logic Assertions

Definition (Separation Logic assertions)

Separation Logic (SL) assertions are defined by the following context-free grammar (where $a, a_i \in AExp$ and $x \in Var$):

$$\begin{aligned} A ::= & \text{emp} \mid a \mapsto a' \mid A_1 * A_2 \mid t \mid a_1 = a_2 \mid a_1 > a_2 \mid \\ & \neg A \mid A_1 \wedge A_2 \mid A_1 \vee A_2 \mid \forall x : A \\ & \in SLA \end{aligned}$$

- **Abbreviations:** $a \mapsto (a_1, \dots, a_n) := a \mapsto a_1 * \dots * a_{n-1} \mapsto a_n$
 $a \mapsto - := \exists x : a \mapsto x$
 $\exists x : A := \neg(\forall x : \neg A)$
 $A_1 \Rightarrow A_2 := \neg A_1 \vee A_2$
 $A_1 \iff A_2 := A_1 \Rightarrow A_2 \wedge A_2 \Rightarrow A_1$
 $a_1 \geq a_2 := a_1 > a_2 \vee a_1 = a_2$
 \vdots

- **Remark:** recursive predicates for data structures will be introduced later

Recap: Pointer Programs and Separation Logic

Semantics of Separation Logic

Definition (Semantics of SL assertions)

Let $A \in SLA$ and $(s, h) \in \Sigma$. The relation $(s, h) \models A$ is inductively defined by:

$$\begin{aligned} (s, h) \models \text{emp} & \quad \text{if } \text{dom}(h) = \emptyset \\ (s, h) \models a \mapsto a' & \quad \text{if } h = h_\emptyset[\mathcal{A}[[a]]s \mapsto \mathcal{A}[[a']]s] \\ (s, h) \models A_1 * A_2 & \quad \text{if } \exists h_1, h_2 \in \text{Heap} : h = h_1 \uplus h_2, \\ & \quad (s, h_1) \models A_1 \text{ and } (s, h_2) \models A_2 \\ (s, h) \models \forall x : A & \quad \text{if } \forall z \in \mathbb{Z} : (s[x \mapsto z], h) \models A \\ (s, h) \models \text{true} & \\ (s, h) \models a_1 = a_2 & \quad \text{if } \mathcal{A}[[a_1]]s = \mathcal{A}[[a_2]]s \\ (s, h) \models a_1 > a_2 & \quad \text{if } \mathcal{A}[[a_1]]s > \mathcal{A}[[a_2]]s \\ (s, h) \models \neg A & \quad \text{if not } (s, h) \models A \\ (s, h) \models A_1 \wedge A_2 & \quad \text{if } (s, h) \models A_1 \text{ and } (s, h) \models A_2 \\ (s, h) \models A_1 \vee A_2 & \quad \text{if } (s, h) \models A_1 \text{ or } (s, h) \models A_2 \end{aligned}$$

Furthermore we let $\Sigma(A) := \{\sigma \in \Sigma \mid \sigma \models A\}$, and A is called **valid** (notation: $\models A$) if $\Sigma(A) = \Sigma$.

Recap: Pointer Programs and Separation Logic

Examples

Example

1. For any $s \in \text{Stack}$:

$$(s, h_{\emptyset}[4 \mapsto 7, 7 \mapsto 4]) \models 4 \mapsto 7 * 7 \mapsto 4 \quad \text{but} \quad (s, h_{\emptyset}[4 \mapsto 7, 7 \mapsto 4]) \not\models 4 \mapsto 7 \wedge 7 \mapsto 4$$

Conversely,

$$(s, h_{\emptyset}[4 \mapsto 7]) \models 4 \mapsto 7 \wedge 4 \mapsto 7 \quad \text{but} \quad (s, h_{\emptyset}[4 \mapsto 7]) \not\models 4 \mapsto 7 * 4 \mapsto 7$$

2. More generally, separating conjunction prevents unintended aliasing effects. For instance,

$$\begin{aligned} & (s, h) \models x \mapsto 0 * y \mapsto 0 \\ \iff & \exists h_1, h_2 \in \text{Heap} : h = h_1 \uplus h_2, (s, h_1) \models x \mapsto 0 \text{ and } (s, h_2) \models y \mapsto 0 \\ \iff & \exists h_1, h_2 \in \text{Heap} : h = h_1 \uplus h_2, h_1 = h_{\emptyset}[s(x) \mapsto 0] \text{ and } h_2 = h_{\emptyset}[s(y) \mapsto 0] \end{aligned}$$

But $h_1 \uplus h_2$ is defined only if $h_1 \# h_2$, which implies $s(x) \neq s(y)$.

Properties of Separating Conjunction

Outline of Lecture 18

Recap: Pointer Programs and Separation Logic

Properties of Separating Conjunction

Recursive Data Structures

Partial Correctness Properties

The Proof System

Properties of Separating Conjunction

Properties of Separating Conjunction

Theorem 18.1 (Properties of separating conjunction)

For all $A, B, C \in SLA$:

1. *Associativity*: $\models A * (B * C) \iff (A * B) * C$
2. *Commutativity*: $\models A * B \iff B * A$
3. *Neutrality of emp*: $\models A * \text{emp} \iff \text{emp} * A \iff A$
4. *Distributivity over \vee* : $\models (A \vee B) * C \iff (A * C) \vee (B * C)$
5. *Sub-distributivity over \wedge* : $\models (A \wedge B) * C \Rightarrow (A * C) \wedge (B * C)$
6. *Distributivity over \exists* : if $x \notin FV(B)$, then $\models (\exists x : A) * B \iff \exists x : (A * B)$
7. *Sub-distributivity over \forall* : if $x \notin FV(B)$, then $\models (\forall x : A) * B \Rightarrow \forall x : (A * B)$

Properties of Separating Conjunction

Properties of Separating Conjunction

Theorem 18.1 (Properties of separating conjunction)

For all $A, B, C \in SLA$:

1. *Associativity*: $\models A * (B * C) \iff (A * B) * C$
2. *Commutativity*: $\models A * B \iff B * A$
3. *Neutrality of emp*: $\models A * \text{emp} \iff \text{emp} * A \iff A$
4. *Distributivity over \vee* : $\models (A \vee B) * C \iff (A * C) \vee (B * C)$
5. *Sub-distributivity over \wedge* : $\models (A \wedge B) * C \Rightarrow (A * C) \wedge (B * C)$
6. *Distributivity over \exists* : if $x \notin FV(B)$, then $\models (\exists x : A) * B \iff \exists x : (A * B)$
7. *Sub-distributivity over \forall* : if $x \notin FV(B)$, then $\models (\forall x : A) * B \Rightarrow \forall x : (A * B)$

Proof.

on the board



Recursive Data Structures

Outline of Lecture 18

Recap: Pointer Programs and Separation Logic

Properties of Separating Conjunction

Recursive Data Structures

Partial Correctness Properties

The Proof System

Recursive Predicate Definitions I

Definition 18.2 (SL predicates)

- A **(recursive) predicate definition** is an equation of the form

$$P(x_1, \dots, x_n) := A$$

where P is a predicate symbol, $n \in \mathbb{N}$, $x_1, \dots, x_n \in \text{Var}$ and $A \in \text{SLA}$ an SL assertion, additionally containing recursive predicate calls of the form $P(a_1, \dots, a_n)$ with $a_1, \dots, a_n \in \text{AExp}$. Syntactic restriction (to ensure monotonicity): each call must be in the scope of an **even number of negations**.

Recursive Predicate Definitions I

Definition 18.2 (SL predicates)

- A **(recursive) predicate definition** is an equation of the form

$$P(x_1, \dots, x_n) := A$$

where P is a predicate symbol, $n \in \mathbb{N}$, $x_1, \dots, x_n \in \text{Var}$ and $A \in \text{SLA}$ an SL assertion, additionally containing recursive predicate calls of the form $P(a_1, \dots, a_n)$ with $a_1, \dots, a_n \in \text{AExp}$. Syntactic restriction (to ensure monotonicity): each call must be in the scope of an **even number of negations**.

- It induces the **functional** $\Phi : (\mathbb{Z}^n \rightarrow 2^\Sigma) \rightarrow (\mathbb{Z}^n \rightarrow 2^\Sigma)$, given by

$$\Phi(p)(z_1, \dots, z_n) := \Sigma(A[P \mapsto p, x_1 \mapsto z_1, \dots, x_n \mapsto z_n]).$$

Recursive Predicate Definitions I

Definition 18.2 (SL predicates)

- A **(recursive) predicate definition** is an equation of the form

$$P(x_1, \dots, x_n) := A$$

where P is a predicate symbol, $n \in \mathbb{N}$, $x_1, \dots, x_n \in \text{Var}$ and $A \in \text{SLA}$ an SL assertion, additionally containing recursive predicate calls of the form $P(a_1, \dots, a_n)$ with $a_1, \dots, a_n \in \text{AExp}$. Syntactic restriction (to ensure monotonicity): each call must be in the scope of an **even number of negations**.

- It induces the **functional** $\Phi : (\mathbb{Z}^n \rightarrow 2^\Sigma) \rightarrow (\mathbb{Z}^n \rightarrow 2^\Sigma)$, given by

$$\Phi(p)(z_1, \dots, z_n) := \Sigma(A[P \mapsto p, x_1 \mapsto z_1, \dots, x_n \mapsto z_n]).$$

- A state $(s, h) \in \Sigma$ **satisfies** a predicate call $P(a_1, \dots, a_n)$ (notation: $(s, h) \models P(a_1, \dots, a_n)$) if $(s, h) \in \text{fix}(\Phi)(\mathcal{A}[[a_1]]s, \dots, \mathcal{A}[[a_n]]s)$.

Recursive Predicate Definitions II

The previous definition is justified by the following properties:

Lemma 18.3

1. $(\mathbb{Z}^n \rightarrow 2^\Sigma, \sqsubseteq)$ is a **CCPO** where $p \sqsubseteq q$ iff $p(z_1, \dots, z_n) \subseteq q(z_1, \dots, z_n)$ for all $z_1, \dots, z_n \in \mathbb{Z}$.
2. $\Phi : (\mathbb{Z}^n \rightarrow 2^\Sigma) \rightarrow (\mathbb{Z}^n \rightarrow 2^\Sigma)$ is **monotonic and continuous** w.r.t. $(\mathbb{Z}^n \rightarrow 2^\Sigma, \sqsubseteq)$.

Proof.

omitted □

Recursive Predicate Definitions II

The previous definition is justified by the following properties:

Lemma 18.3

1. $(\mathbb{Z}^n \rightarrow 2^\Sigma, \sqsubseteq)$ is a **CCPO** where $p \sqsubseteq q$ iff $p(z_1, \dots, z_n) \subseteq q(z_1, \dots, z_n)$ for all $z_1, \dots, z_n \in \mathbb{Z}$.
2. $\Phi : (\mathbb{Z}^n \rightarrow 2^\Sigma) \rightarrow (\mathbb{Z}^n \rightarrow 2^\Sigma)$ is **monotonic and continuous** w.r.t. $(\mathbb{Z}^n \rightarrow 2^\Sigma, \sqsubseteq)$.

Proof.

omitted □

Thus, Theorem 8.1 is applicable and yields

$$\text{fix}(\Phi) = \bigsqcup \{ \Phi^n(p_\emptyset) \mid n \in \mathbb{N} \}$$

where $p_\emptyset(z_1, \dots, z_n) := \emptyset$ for all $z_1, \dots, z_n \in \mathbb{Z}$.

Recursive Predicate Definitions III

Example 18.4

A predicate defining **singly-linked list segments** from address x to y :

$$\text{sll}(x, y) := (x = y \wedge \text{emp}) \vee \exists x' : x \mapsto x' * \text{sll}(x', y)$$

Recursive Predicate Definitions III

Example 18.4

A predicate defining **singly-linked list segments** from address x to y :

$$\text{sll}(x, y) := (x = y \wedge \text{emp}) \vee \exists x' : x \mapsto x' * \text{sll}(x', y)$$

Fixpoint iteration:

$$\Phi^0(p_\emptyset)(x, y) = \emptyset$$

Recursive Predicate Definitions III

Example 18.4

A predicate defining **singly-linked list segments** from address x to y :

$$\text{sll}(x, y) := (x = y \wedge \text{emp}) \vee \exists x' : x \mapsto x' * \text{sll}(x', y)$$

Fixpoint iteration:

$$\Phi^0(p_\emptyset)(x, y) = \emptyset$$

$$\begin{aligned} \Phi^1(p_\emptyset)(x, y) &= \{(s, h) \in \Sigma \mid (s, h) \models (x = y \wedge \text{emp}) \vee \exists x' : x \mapsto x' * \Phi^0(p_\emptyset)(x', y)\} \\ &= \{(s, h) \in \Sigma \mid s(x) = s(y) \wedge h = h_\emptyset\} \end{aligned}$$

Recursive Predicate Definitions III

Example 18.4

A predicate defining **singly-linked list segments** from address x to y :

$$\text{sll}(x, y) := (x = y \wedge \text{emp}) \vee \exists x' : x \mapsto x' * \text{sll}(x', y)$$

Fixpoint iteration:

$$\Phi^0(p_\emptyset)(x, y) = \emptyset$$

$$\begin{aligned} \Phi^1(p_\emptyset)(x, y) &= \{(s, h) \in \Sigma \mid (s, h) \models (x = y \wedge \text{emp}) \vee \exists x' : x \mapsto x' * \Phi^0(p_\emptyset)(x', y)\} \\ &= \{(s, h) \in \Sigma \mid s(x) = s(y) \wedge h = h_\emptyset\} \end{aligned}$$

$$\begin{aligned} \Phi^2(p_\emptyset)(x, y) &= \{(s, h) \in \Sigma \mid (s, h) \models (x = y \wedge \text{emp}) \vee \exists x' : x \mapsto x' * \Phi^1(p_\emptyset)(x', y)\} \\ &= \{(s, h) \in \Sigma \mid (s, h) \models (x = y \wedge \text{emp}) \vee \exists x' : x \mapsto x' * (x' = y \wedge \text{emp})\} \\ &= \{(s, h) \in \Sigma \mid (s(x) = s(y) \wedge h = h_\emptyset) \vee h = h_\emptyset[s(x) \mapsto s(y)]\} \end{aligned}$$

Recursive Predicate Definitions III

Example 18.4

A predicate defining **singly-linked list segments** from address x to y :

$$\text{sll}(x, y) := (x = y \wedge \text{emp}) \vee \exists x' : x \mapsto x' * \text{sll}(x', y)$$

Fixpoint iteration:

$$\Phi^0(p_\emptyset)(x, y) = \emptyset$$

$$\begin{aligned} \Phi^1(p_\emptyset)(x, y) &= \{(s, h) \in \Sigma \mid (s, h) \models (x = y \wedge \text{emp}) \vee \exists x' : x \mapsto x' * \Phi^0(p_\emptyset)(x', y)\} \\ &= \{(s, h) \in \Sigma \mid s(x) = s(y) \wedge h = h_\emptyset\} \end{aligned}$$

$$\begin{aligned} \Phi^2(p_\emptyset)(x, y) &= \{(s, h) \in \Sigma \mid (s, h) \models (x = y \wedge \text{emp}) \vee \exists x' : x \mapsto x' * \Phi^1(p_\emptyset)(x', y)\} \\ &= \{(s, h) \in \Sigma \mid (s, h) \models (x = y \wedge \text{emp}) \vee \exists x' : x \mapsto x' * (x' = y \wedge \text{emp})\} \\ &= \{(s, h) \in \Sigma \mid (s(x) = s(y) \wedge h = h_\emptyset) \vee h = h_\emptyset[s(x) \mapsto s(y)]\} \end{aligned}$$

$$\begin{aligned} \Phi^3(p_\emptyset)(x, y) &= \{(s, h) \in \Sigma \mid (s, h) \models (x = y \wedge \text{emp}) \vee \exists x' : x \mapsto x' * \Phi^2(p_\emptyset)(x', y)\} \\ &= \{(s, h) \in \Sigma \mid (s(x) = s(y) \wedge h = h_\emptyset) \vee h = h_\emptyset[s(x) \mapsto s(y)] \vee \\ &\quad \exists x' \in \mathbb{Z} : h = h_\emptyset[s(x) \mapsto s(x'), s(x') \mapsto s(y)]\} \end{aligned}$$

⋮

Partial Correctness Properties

Outline of Lecture 18

Recap: Pointer Programs and Separation Logic

Properties of Separating Conjunction

Recursive Data Structures

Partial Correctness Properties

The Proof System

Partial Correctness Properties in Separation Logic

Here we take a **fault-avoiding** interpretation of Hoare triples:

- programs must be **memory-safe**, i.e., **never** reach a fault
- **all** successfully terminating programs must satisfy the postcondition

Partial Correctness Properties

Partial Correctness Properties in Separation Logic

Here we take a **fault-avoiding** interpretation of Hoare triples:

- programs must be **memory-safe**, i.e., **never** reach a fault
- **all** successfully terminating programs must satisfy the postcondition

Definition 18.5 (Partial correctness properties)

- For $A, B \in SLA$ and $c \in Cmd$, $\{A\} c \{B\}$ is called a **partial correctness property (PCP)** with **precondition A** and **postcondition B** .

Partial Correctness Properties

Partial Correctness Properties in Separation Logic

Here we take a **fault-avoiding** interpretation of Hoare triples:

- programs must be **memory-safe**, i.e., **never** reach a fault
- **all** successfully terminating programs must satisfy the postcondition

Definition 18.5 (Partial correctness properties)

- For $A, B \in SLA$ and $c \in Cmd$, $\{A\} c \{B\}$ is called a **partial correctness property (PCP)** with **precondition A** and **postcondition B** .
- A state $(s, h) \in \Sigma$ **satisfies** PCP $\{A\} c \{B\}$ (notation: $(s, h) \models \{A\} c \{B\}$) if, whenever $(s, h) \models A$,
 1. $\langle c, (s, h) \rangle \not\rightarrow \downarrow$ and
 2. if $\langle c, (s, h) \rangle \rightarrow (s', h')$, then $(s', h') \models B$.

Partial Correctness Properties

Partial Correctness Properties in Separation Logic

Here we take a **fault-avoiding** interpretation of Hoare triples:

- programs must be **memory-safe**, i.e., **never** reach a fault
- **all** successfully terminating programs must satisfy the postcondition

Definition 18.5 (Partial correctness properties)

- For $A, B \in SLA$ and $c \in Cmd$, $\{A\} c \{B\}$ is called a **partial correctness property (PCP)** with **precondition** A and **postcondition** B .
- A state $(s, h) \in \Sigma$ **satisfies** PCP $\{A\} c \{B\}$ (notation: $(s, h) \models \{A\} c \{B\}$) if, whenever $(s, h) \models A$,
 1. $\langle c, (s, h) \rangle \not\rightarrow \downarrow$ and
 2. if $\langle c, (s, h) \rangle \rightarrow (s', h')$, then $(s', h') \models B$.
- PCP $\{A\} c \{B\}$ is called **valid** (notation: $\models \{A\} c \{B\}$) if $(s, h) \models \{A\} c \{B\}$ for every $(s, h) \in \Sigma$.

The Proof System

Outline of Lecture 18

Recap: Pointer Programs and Separation Logic

Properties of Separating Conjunction

Recursive Data Structures

Partial Correctness Properties

The Proof System

The Proof System

The Proof System I

Definition 18.6 (SL proof rules)

$$\text{(alloc)} \frac{x \notin FV(\vec{a})}{\{\text{emp}\} x := \text{alloc}(\vec{a}) \{x \mapsto \vec{a}\}}$$

$$\text{(free)} \frac{}{\{a \mapsto -\} \text{free}(a) \{\text{emp}\}}$$

$$\text{(asgn)} \frac{x \notin FV(a)}{\{\text{emp}\} x := a \{\text{emp} \wedge x = a\}}$$

$$\text{(skip)} \frac{}{\{A\} \text{skip} \{A\}}$$

$$\text{(if)} \frac{\{A \wedge b\} c_1 \{B\} \quad \{A \wedge \neg b\} c_2 \{B\}}{\{A\} \text{if } b \text{ then } c_1 \text{ else } c_2 \text{ end } \{B\}}$$

$$\text{(cons)} \frac{\models (A \Rightarrow A') \quad \{A'\} c \{B'\} \quad \models (B' \Rightarrow B)}{\{A\} c \{B\}}$$

$$\text{(lookup)} \frac{x \notin FV(a)}{\{a \mapsto v\} x := [a] \{a \mapsto v \wedge x = v\}}$$

$$\text{(mutate)} \frac{}{\{a \mapsto -\} [a] := a' \{a \mapsto a'\}}$$

$$\text{(frame)} \frac{\{A\} c \{B\} \quad \text{Var}(C) \cap \text{Mod}(c) = \emptyset}{\{A * C\} c \{B * C\}}$$

$$\text{(seq)} \frac{\{A\} c_1 \{C\} \quad \{C\} c_2 \{B\}}{\{A\} c_1 ; c_2 \{B\}}$$

$$\text{(while)} \frac{\{A \wedge b\} c \{A\}}{\{A\} \text{while } b \text{ do } c \text{ end } \{A \wedge \neg b\}}$$

The Proof System

The Proof System II

Remarks:

- The preconditions imposed by (lookup), (free) and (mutate) ensure the fault-avoiding interpretation of PCPs.

The Proof System

The Proof System II

Remarks:

- The preconditions imposed by (lookup), (free) and (mutate) ensure the fault-avoiding interpretation of PCPs.
- The new rules only consider the “**memory footprint**” of the respective statement, i.e., mention only the cells accessed or (de)allocated (“**local reasoning**”).
- Therefore, the **frame rule**

$$\text{(frame)} \frac{\{A\} c \{B\} \quad \text{Var}(C) \cap \text{Mod}(c) = \emptyset}{\{A * C\} c \{B * C\}}$$

is crucial for embedding the local operation in a larger (heap) context C that is unaffected by the execution of c .

The Proof System

The Proof System II

Remarks:

- The preconditions imposed by (lookup), (free) and (mutate) ensure the fault-avoiding interpretation of PCPs.
- The new rules only consider the “memory footprint” of the respective statement, i.e., mention only the cells accessed or (de)allocated (“local reasoning”).
- Therefore, the **frame rule**

$$\text{(frame)} \frac{\{A\} c \{B\} \quad \text{Var}(C) \cap \text{Mod}(c) = \emptyset}{\{A * C\} c \{B * C\}}$$

is crucial for embedding the local operation in a larger (heap) context C that is unaffected by the execution of c .

- Some rules have refined variants that get rid of side conditions. For example, (asgn') can be used in place of (asgn) in case $x \in FV(a)$, where v denotes a fresh variable that stores the previous value of x (and similarly for (alloc) and (lookup)):

$$\text{(asgn)} \frac{x \notin FV(a)}{\{\text{emp}\} x := a \{\text{emp} \wedge x = a\}} \quad \text{(asgn')} \frac{}{\{\text{emp} \wedge x = v\} x := a \{\text{emp} \wedge x = a[x \mapsto v]\}}$$

However, these refined rules are not required for our examples.

The Proof System

Pointer Swap Example

Example 18.7 (Pointer swap)

$t := [x];$

$u := [y];$

$[x] := u;$

$[y] := t;$

The Proof System

Pointer Swap Example

Example 18.7 (Pointer swap)

$$\{x \mapsto v * y \mapsto w\}$$
$$t := [x];$$

Precondition

$$u := [y];$$
$$[x] := u;$$
$$[y] := t;$$
$$\{x \mapsto w * y \mapsto v\}$$

Postcondition

The Proof System

Pointer Swap Example

Example 18.7 (Pointer swap)

$$\begin{aligned} & \{x \mapsto v * y \mapsto w\} \\ & t := [x]; \\ & \{(x \mapsto v \wedge t = v) * y \mapsto w\} \\ & u := [y]; \end{aligned}$$

Precondition

(lookup, frame)

$$[x] := u;$$
$$[y] := t;$$
$$\{x \mapsto w * y \mapsto v\}$$

Postcondition

$$x \notin FV(a)$$
$$\frac{\text{(lookup)}}{\{a \mapsto v\} x := [a] \{a \mapsto v \wedge x = v\}}$$
$$\frac{\text{(frame)} \quad \{A\} c \{B\} \quad \text{Var}(C) \cap \text{Mod}(c) = \emptyset}{\{A * C\} c \{B * C\}}$$

The Proof System

Pointer Swap Example

Example 18.7 (Pointer swap)

$$\{x \mapsto v * y \mapsto w\}$$
$$t := [x];$$
$$\{(x \mapsto v \wedge t = v) * y \mapsto w\}$$
$$u := [y];$$
$$\{(x \mapsto v \wedge t = v) * (y \mapsto w \wedge u = w)\}$$
$$[x] := u;$$
$$[y] := t;$$
$$\{x \mapsto w * y \mapsto v\}$$
$$x \notin FV(a)$$
$$\frac{\text{(lookup)}}{\{a \mapsto v\} x := [a] \{a \mapsto v \wedge x = v\}}$$

Precondition

(lookup, frame)

(lookup, frame)

Postcondition

$$\frac{\text{(frame)} \quad \{A\} c \{B\} \quad \text{Var}(C) \cap \text{Mod}(c) = \emptyset}{\{A * C\} c \{B * C\}}$$

The Proof System

Pointer Swap Example

Example 18.7 (Pointer swap)

$\{x \mapsto v * y \mapsto w\}$	Precondition
$t := [x];$	
$\{(x \mapsto v \wedge t = v) * y \mapsto w\}$	(lookup, frame)
$u := [y];$	
$\{(x \mapsto v \wedge t = v) * (y \mapsto w \wedge u = w)\}$	(lookup, frame)
$\Rightarrow \{x \mapsto v * y \mapsto w \wedge t = v \wedge u = w\}$	(Theorem 18.1(5))
$[x] := u;$	
$[y] := t;$	
$\{x \mapsto w * y \mapsto v\}$	Postcondition

$$\models (A \wedge B) * C \Rightarrow (A * C) \wedge (B * C)$$

The Proof System

Pointer Swap Example

Example 18.7 (Pointer swap)

$$\{x \mapsto v * y \mapsto w\}$$
$$t := [x];$$
$$\{(x \mapsto v \wedge t = v) * y \mapsto w\}$$
$$u := [y];$$
$$\{(x \mapsto v \wedge t = v) * (y \mapsto w \wedge u = w)\}$$
$$\Rightarrow \{x \mapsto v * y \mapsto w \wedge t = v \wedge u = w\}$$
$$[x] := u;$$
$$\{x \mapsto u * y \mapsto w \wedge t = v \wedge u = w\}$$
$$[y] := t;$$
$$\{x \mapsto w * y \mapsto v\}$$

Precondition

(lookup, frame)

(lookup, frame)

(Theorem 18.1(5))

(mutate, frame)

Postcondition

$$\frac{\text{(mutate)} \quad \{a \mapsto -\} [a] := a' \{a \mapsto a'\}}{\text{(frame)} \quad \frac{\{A\} c \{B\} \quad \text{Var}(C) \cap \text{Mod}(c) = \emptyset}{\{A * C\} c \{B * C\}}}$$

The Proof System

Pointer Swap Example

Example 18.7 (Pointer swap)

$\{x \mapsto v * y \mapsto w\}$	Precondition
$t := [x];$	
$\{(x \mapsto v \wedge t = v) * y \mapsto w\}$	(lookup, frame)
$u := [y];$	
$\{(x \mapsto v \wedge t = v) * (y \mapsto w \wedge u = w)\}$	(lookup, frame)
$\Rightarrow \{x \mapsto v * y \mapsto w \wedge t = v \wedge u = w\}$	(Theorem 18.1(5))
$[x] := u;$	
$\{x \mapsto u * y \mapsto w \wedge t = v \wedge u = w\}$	(mutate, frame)
$[y] := t;$	
$\{x \mapsto u * y \mapsto t \wedge t = v \wedge u = w\}$	(mutate, frame)
$\{x \mapsto w * y \mapsto v\}$	Postcondition

$$\frac{\text{(mutate)}}{\{a \mapsto -\} [a] := a' \{a \mapsto a'\}} \quad \frac{\text{(frame)} \quad \{A\} c \{B\} \quad \text{Var}(C) \cap \text{Mod}(c) = \emptyset}{\{A * C\} c \{B * C\}}$$

The Proof System

Pointer Swap Example

Example 18.7 (Pointer swap)

$\{x \mapsto v * y \mapsto w\}$	Precondition
$t := [x];$	
$\{(x \mapsto v \wedge t = v) * y \mapsto w\}$	(lookup, frame)
$u := [y];$	
$\{(x \mapsto v \wedge t = v) * (y \mapsto w \wedge u = w)\}$	(lookup, frame)
$\Rightarrow \{x \mapsto v * y \mapsto w \wedge t = v \wedge u = w\}$	(Theorem 18.1(5))
$[x] := u;$	
$\{x \mapsto u * y \mapsto w \wedge t = v \wedge u = w\}$	(mutate, frame)
$[y] := t;$	
$\{x \mapsto u * y \mapsto t \wedge t = v \wedge u = w\}$	(mutate, frame)
$\Rightarrow \{x \mapsto w * y \mapsto v\}$	Postcondition