



# Semantics and Verification of Software

Summer Semester 2019

Lecture 17: Separation Logic I (Introduction)

Thomas Noll

Software Modeling and Verification Group

RWTH Aachen University

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

# Motivation

## Problems ...



<https://xkcd.com/371>

## Pointer programming errors

- Dereferencing null (or disposed) pointers
- Creation of memory leaks
- Accidental invalidation of data structures, ...

# Motivation

---

## ... and Possible Solutions

### Challenges

- **Finite representation** of unbounded state spaces (dynamic storage (de-)allocation)
- **Modular** reasoning
  - only consider the heap part affected by the program of interest (“memory footprint”)
  - enable compositional reasoning about procedures/threads (“procedure/thread summaries”)

### Analysis approaches

- **Separation Logic**
- Shape analysis
- (Regular tree) automata
- Hyperedge replacement grammars

## Motivation

---

### Classical Deficiency of Hoare Logic

The so-called **rule of constancy** in Hoare Logic,

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

(where  $Mod(c) \subseteq Var$  collects all variables written by  $c$ ) becomes **unsound** when we consider pointers.

E.g., for pointer variables  $x$  and  $y$ ,

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

(where “ $x \mapsto z$ ” stands for “ $x$  holds the address of a memory location where  $z$  is stored”, and  $[.]$  denotes the dereferencing operator) is **not valid** (because  $y$  could **alias**  $x$ , i.e., contain the same address).

# Motivation

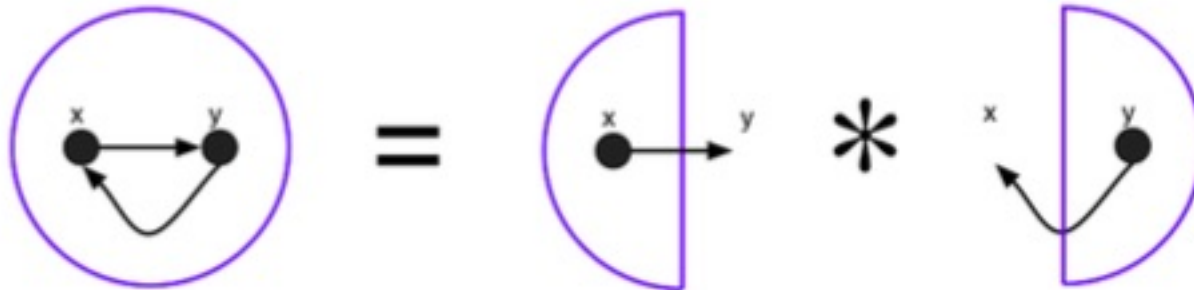
## Solution: Separating Conjunction

Key idea of Separation Logic:

Separating conjunction

New conjunction operator  $*$  meaning “and, *separately* in memory”

For example,  $x \mapsto y * y \mapsto x$ :



## The Power of Separating Conjunction

### Example 17.1 (Trees in Separation Logic)

Separation Logic specification of binary trees with root node  $x$ :

$$\text{tree}(x) := (x = 0 \wedge \text{emp}) \vee \exists y, z : x \mapsto (y, z) * \text{tree}(y) * \text{tree}(z)$$

where

- $\text{emp}$  denotes the empty heap
- $x \mapsto (y, z)$  denotes a pointer to a pair of data cells (left/right child)
- $*$  means “and, **separately** in memory”
- Thus tree property is ensured by disabling sharing between  $x$  and (the cells reachable from)  $y$  and  $z$

# Motivation

---

## 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  $y$ .

### Employing Separation Logic for Program Verification

#### Example 17.2 (Recursive tree disposal)

```
{tree(x)}
delete(*x) {
  if x = null then return {emp}
  else { { $\exists y, z : x \mapsto (y, z) * \text{tree}(y) * \text{tree}(z)$ }
    l := x.left; r := x.right;
    { $x \mapsto (l, r) * \text{tree}(l) * \text{tree}(r)$ }
    delete(l);
    { $x \mapsto (l, r) * \text{emp} * \text{tree}(r)$ }
    delete(r);
    { $x \mapsto (l, r) * \text{emp} * \text{emp}$ }
    free(x); free(x + 1)
    {emp * emp * emp}
  } {emp}
} {emp}
```



# Motivation

---

## Roadmap

1. Extension of WHILE language by pointer operations
2. Syntax and Semantics of Separation Logic
3. The Proof System

# Extending the Syntax of WHILE

---

## 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)

# Extending the Syntax of WHILE

---

## An Example

### Example 17.3 (List traversal)

- Variable `head` initially points to head of a singly-linked list
- Every list element consists of two addresses: “next” pointer and value
- Last element indicated by null pointer

The following program traverses the list, computing the sum of all values and deleting each element:

```
sum := 0;
while ¬(head = 0) do
  next := [head];
  val := [head + 1];
  sum := sum + val;
  free(head);
  free(head + 1);
  head := next
end
```

# Semantics of Pointer Programs

## 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 17.4 (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

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

# Semantics of Pointer Programs

## Execution of Statements I

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

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

### Definition 17.5 (Execution relation for statements)

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

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

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

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

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

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

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

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

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

# Semantics of Pointer Programs

## Execution of Statements II

### Definition 17.5 (Execution relation for statements (continued))

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

### 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!

# Semantics of Pointer Programs

## An Example

### Example 17.6

```

c : x := alloc(0); y := alloc(0); } c1
    [1] := 0; } c2
    while y = 3 do skip end } c3
  
```

Notation: for  $l, m \in \mathbb{N}_{>0}$ ,  $s_{lm} := s[x \mapsto l, y \mapsto m]$  and  $h_{lm} := h_\emptyset[l \mapsto 0, m \mapsto 0]$

#### 1. A successful computation:

$$\begin{array}{c}
 \text{(seq)} \frac{2 \times (\text{alloc}) \frac{\langle c_1, (s_0, h_\emptyset) \rangle \rightarrow (s_{12}, h_{12})}{\langle c_1, (s_0, h_\emptyset) \rangle \rightarrow (s_{12}, h_{12})}}{\langle c_1, (s_0, h_\emptyset) \rangle \rightarrow (s_{12}, h_{12})} \quad \text{(seq)} \frac{\text{(mutate)} \frac{\langle c_2, (s_{12}, h_{12}) \rangle \rightarrow (s_{12}, h_{12})}{\langle c_2, (s_{12}, h_{12}) \rangle \rightarrow (s_{12}, h_{12})} \quad \text{(wh-f)} \frac{\mathfrak{B}[[y=3]]s_{12} = \text{false}}{\langle c_3, (s_{12}, h_{12}) \rangle \rightarrow (s_{12}, h_{12})}}{\langle c_2; c_3, (s_1, h_1) \rangle \rightarrow (s_{12}, h_{12})}}{\langle c, (s_{0,0}, h_\emptyset) \rangle \rightarrow (s_{12}, h_{12})}
 \end{array}$$

#### 2. A failing computation:

$$\begin{array}{c}
 \text{(seq)} \frac{2 \times (\text{alloc}) \frac{\langle c_1, (s_0, h_\emptyset) \rangle \rightarrow (s_{23}, h_{23})}{\langle c_1, (s_0, h_\emptyset) \rangle \rightarrow (s_{23}, h_{23})}}{\langle c_1, (s_0, h_\emptyset) \rangle \rightarrow (s_{23}, h_{23})} \quad \text{(seq)} \frac{\text{(mutate)} \frac{\langle c_2, (s_{23}, h_{23}) \rangle \rightarrow \downarrow}{\langle c_2, (s_{23}, h_{23}) \rangle \rightarrow \downarrow} \quad \text{(fault)} \frac{\langle c_3, \downarrow \rangle \rightarrow \downarrow}{\langle c_3, \downarrow \rangle \rightarrow \downarrow}}{\langle c_2; c_3, (s_1, h_1) \rangle \rightarrow \downarrow}}{\langle c, (s_{0,0}, h_\emptyset) \rangle \rightarrow \downarrow}
 \end{array}$$

#### 3. A non-terminating computation:

$$\begin{array}{c}
 \text{(seq)} \frac{2 \times (\text{alloc}) \frac{\langle c_1, (s_0, h_\emptyset) \rangle \rightarrow (s_{13}, h_{13})}{\langle c_1, (s_0, h_\emptyset) \rangle \rightarrow (s_{13}, h_{13})}}{\langle c_1, (s_0, h_\emptyset) \rangle \rightarrow (s_{13}, h_{13})} \quad \text{(seq)} \frac{\text{(mutate)} \frac{\langle c_2, (s_{13}, h_{13}) \rangle \rightarrow (s_{13}, h_{13})}{\langle c_2, (s_{13}, h_{13}) \rangle \rightarrow (s_{13}, h_{13})} \quad \text{(wh-t)} \frac{\mathfrak{B}[[y=3]]s_{13} = \text{true} \dots}{\langle c_3, (s_{13}, h_{13}) \rangle \rightarrow \dots}}{\langle c_2; c_3, (s_1, h_1) \rangle \rightarrow \dots}}{\langle c, (s_{0,0}, h_\emptyset) \rangle \rightarrow \dots}
 \end{array}$$

# Separation Logic Assertions

## Syntax of Separation Logic Assertions

### Definition 17.7 (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



# Separation Logic Assertions

## Semantics of Separation Logic

### Definition 17.8 (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  $\Sigma(A) = \Sigma$ .

# Separation Logic Assertions

## Examples

### Example 17.9

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  $s(x) \neq s(y)$ .