Semantics and Verification of Software

Summer Semester 2019

Lecture 17: Separation Logic I (Introduction)

Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University

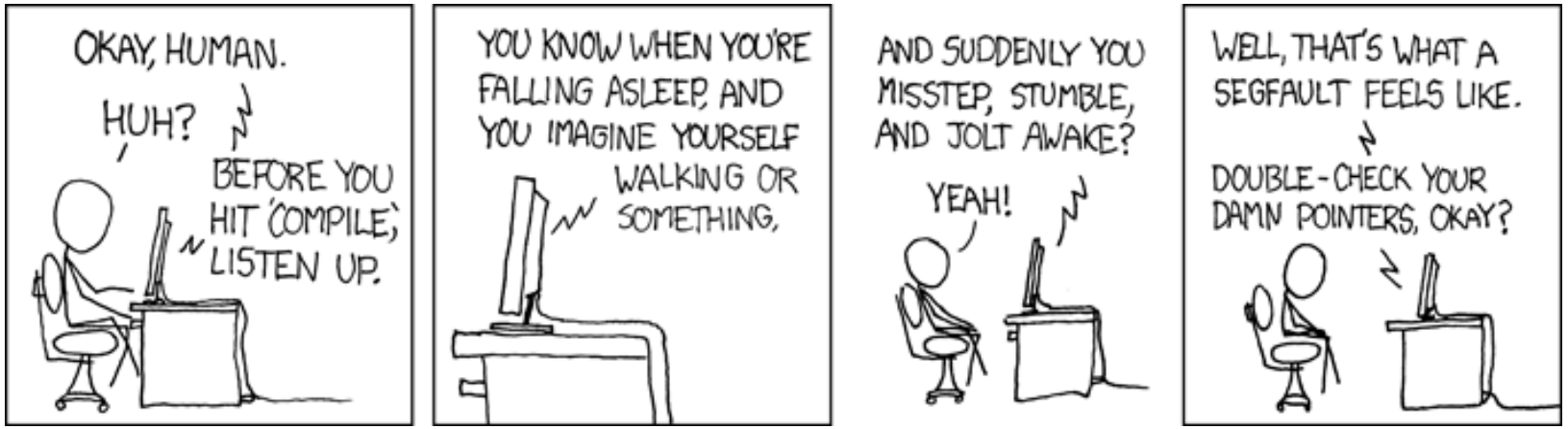
https://moves.rwth-aachen.de/teaching/ss-19/sv-sw/

Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

Outline of Lecture 17

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”)

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

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.

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 $\mathtt{x}$ and $\mathtt{y}$,

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

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

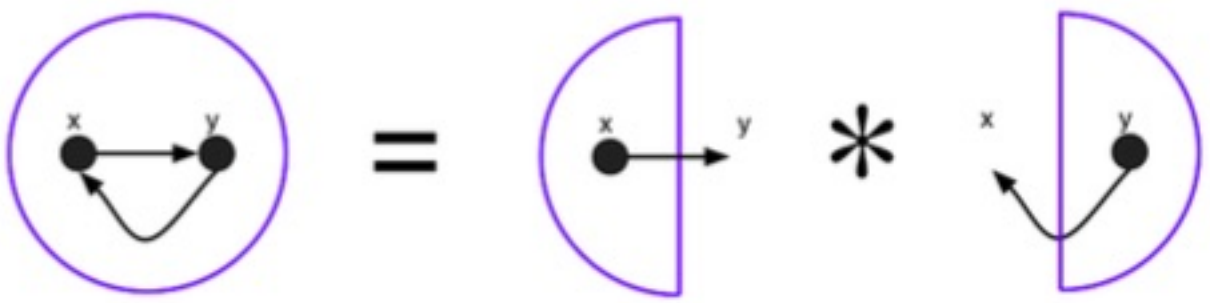
Solution: Separating Conjunction

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:

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

where

- 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

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.

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,

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

is valid as $*$ excludes aliasing of $\mathtt{x}$ and $\mathtt{y}$.

Employing Separation Logic for Program Verification

Example 17.2 (Recursive tree disposal)

```
delete(*x) {
  if x = null then return
  else {
    l := x.left; r := x.right;

    delete(l);

    delete(r);

    free(x); free(x + 1)

  }
}
```

Employing Separation Logic for Program Verification

Example 17.2 (Recursive tree disposal)

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

Roadmap

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

Outline of Lecture 17

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:

$$a ::= z \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \in AExp$$
$$b ::= t \mid a_1 = a_2 \mid a_1 > a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \in BExp$$
$$c ::= x := \texttt{alloc}(a_1, \ldots, a_n) \mid \texttt{free}(a) \mid x := [a] \mid [a] := a' \mid \texttt{skip} \mid x := a \mid c_1 ; c_2 \mid \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ end} \mid \texttt{while } b \texttt{ do } c \texttt{ end} \in Cmd$$

- $x := \texttt{alloc}(a_1, \ldots, a_n)$ allocates $n \geq 1$ fresh addresses with initial values $a_1, \ldots, a_n$
- $\texttt{free}(a)$ 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
```

Outline of Lecture 17

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
$$\mathit{Stack} := \{s \mid s : \mathit{Var} \to \mathbb{Z}\}$$
- A heap is an element of the set
$$\mathit{Heap} := \{h : \mathbb{N}_{>0} \dashrightarrow \mathbb{Z} \mid |\mathrm{dom}(h)| < \infty\}$$
The empty heap is denoted by $h_\emptyset$ (i.e., $\mathrm{dom}(h_\emptyset) = \emptyset$).
- Two heaps $h_1, h_2 \in \mathit{Heap}$ are disjoint (notation: $h_1 \# h_2$) if $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$. In this case, their disjoint union is given by
$$h_1 \uplus h_2 : \mathbb{N}_{>0} \dashrightarrow \mathbb{Z} : l \mapsto \begin{cases} h_1(l) & \text{if } l \in \mathrm{dom}(h_1) \\ h_2(l) & \text{if } l \in \mathrm{dom}(h_2) \end{cases}$$
- The set of (program) states is defined by
$$\Sigma := \mathit{Stack} \times \mathit{Heap}$$

Execution of Statements I

Observation: values of (arithmetic and Boolean) expressions depend on the stack only
$\Rightarrow$ employ evaluation functions $\mathfrak{A}[\![a]\!] : \mathit{Stack} \to \mathbb{Z}$ and $\mathfrak{B}[\![b]\!] : \mathit{Stack} \to \mathbb{B}$

Definition 17.5 (Execution relation for statements)

The execution relation $\to \subseteq (\mathit{Cmd} \times \Sigma \cup \{\lightning\}) \times (\Sigma \cup \{\lightning\})$ is defined by:

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

$$\text{(free)}\ \frac{\mathfrak{A}[\![a]\!]s = l \in \mathrm{dom}(h)}{\langle \mathtt{free}(a), (s,h)\rangle \to (s, h[l \mapsto \bot])} \qquad \text{(free-f)}\ \frac{\mathfrak{A}[\![a]\!]s \notin \mathrm{dom}(h)}{\langle \mathtt{free}(a), (s,h)\rangle \to \lightning}$$

$$\text{(lookup)}\ \frac{\mathfrak{A}[\![a]\!]s = l \in \mathrm{dom}(h)}{\langle x := [a], (s,h)\rangle \to (s[x \mapsto h(l)], h)} \qquad \text{(lookup-f)}\ \frac{\mathfrak{A}[\![a]\!]s \notin \mathrm{dom}(h)}{\langle x := [a], (s,h)\rangle \to \lightning}$$

$$\text{(mutate)}\ \frac{\mathfrak{A}[\![a]\!]s = l \in \mathrm{dom}(h) \quad \mathfrak{A}[\![a']\!]s = z}{\langle [a] := a', (s,h)\rangle \to (s, h[l \mapsto z])} \qquad \text{(mutate-f)}\ \frac{\mathfrak{A}[\![a]\!]s \notin \mathrm{dom}(h)}{\langle [a] := a', (s,h)\rangle \to \lightning}$$

$$\text{(fault)}\ \frac{}{\langle c, \lightning\rangle \to \lightning}$$

Execution of Statements II

Definition 17.5 (Execution relation for statements (continued))

$$\text{(skip)}\ \frac{}{\langle \texttt{skip}, (s,h)\rangle \to (s,h)} \qquad \text{(asgn)}\ \frac{\mathfrak{A}[\![a]\!]s = z}{\langle x := a, (s,h)\rangle \to (s[x \mapsto z], h)}$$

$$\text{(if-t)}\ \frac{\mathfrak{B}[\![b]\!]s = \text{true} \quad \langle c_1, (s,h)\rangle \to \sigma'}{\langle \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ end}, (s,h)\rangle \to \sigma'} \qquad \text{(if-f)}\ \frac{\mathfrak{B}[\![b]\!]s = \text{false} \quad \langle c_2, (s,h)\rangle \to \sigma'}{\langle \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ end}, (s,h)\rangle \to \sigma'}$$

$$\text{(seq)}\ \frac{\langle c_1, \sigma\rangle \to \sigma' \quad \langle c_2, \sigma'\rangle \to \sigma''}{\langle c_1 ; c_2, \sigma\rangle \to \sigma''} \qquad \text{(wh-f)}\ \frac{\mathfrak{B}[\![b]\!]s = \text{false}}{\langle \texttt{while } b \texttt{ do } c \texttt{ end}, (s,h)\rangle \to (s,h)}$$

$$\text{(wh-t)}\ \frac{\mathfrak{B}[\![b]\!]s = \text{true} \quad \langle c, (s,h)\rangle \to \sigma' \quad \langle \texttt{while } b \texttt{ do } c \texttt{ end}, \sigma'\rangle \to \sigma''}{\langle \texttt{while } b \texttt{ do } c \texttt{ end}, (s,h)\rangle \to \sigma''}$$

Remarks:

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

An Example

Example 17.6

$$
\begin{array}{ll}
c: & \texttt{x := alloc(0); y := alloc(0);} \quad \} c_1 \\
& \texttt{[1] := 0;} \quad \} c_2 \\
& \texttt{while y = 3 do skip end} \quad \} c_3
\end{array}
$$

An Example

Example 17.6

$$\begin{array}{ll} c: \texttt{x := alloc(0); y := alloc(0);} & \} c_1 \\ \quad \texttt{[1] := 0;} & \} c_2 \\ \quad \texttt{while y = 3 do skip end} & \} c_3 \end{array}$$

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

1. A successful computation:

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

An Example

Example 17.6

$$\begin{array}{ll} c: \texttt{x := alloc(0); y := alloc(0);} & \} c_1 \\ \quad \texttt{[1] := 0;} & \} c_2 \\ \quad \texttt{while y = 3 do skip end} & \} c_3 \end{array}$$

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

2. A failing computation:

$$\dfrac{\dfrac{}{\langle c_1, (s_0, h_\emptyset)\rangle \to (s_{23}, h_{23})}\,{\scriptstyle 2\times(\text{alloc})} \qquad \dfrac{\dfrac{}{\langle c_2, (s_{23}, h_{23})\rangle \to \lightning}\,{\scriptstyle (\text{mutate})} \qquad \dfrac{}{\langle c_3, \lightning\rangle \to \lightning}\,{\scriptstyle (\text{fault})}}{\langle c_2; c_3, (s_1, h_1)\rangle \to \lightning}\,{\scriptstyle (\text{seq})}}{\langle c, (s_{0,0}, h_\emptyset)\rangle \to \lightning}\,{\scriptstyle (\text{seq})}$$

An Example

Example 17.6

$$\begin{array}{ll} c: \texttt{x := alloc(0); y := alloc(0);} & \} c_1 \\ \quad \texttt{[1] := 0;} & \} c_2 \\ \quad \texttt{while y = 3 do skip end} & \} c_3 \end{array}$$

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

3. A non-terminating computation:

$$\text{(seq)}\ \frac{\text{2}\times\text{(alloc)}\ \dfrac{}{\langle c_1, (s_0, h_\emptyset)\rangle \to (s_{13}, h_{13})} \qquad \text{(seq)}\ \dfrac{\text{(mutate)}\ \dfrac{}{\langle c_2, (s_{13}, h_{13})\rangle \to (s_{13}, h_{13})} \qquad \text{(wh-t)}\ \dfrac{\mathfrak{B}[\![\mathrm{y}{=}3]\!]s_{13} = \text{true} \quad \ldots}{\langle c_3, (s_{12}, h_{12})\rangle \to \bot}}{\langle c_2; c_3, (s_1, h_1)\rangle \to \bot}}{\langle c, (s_{0,0}, h_\emptyset)\rangle \to \bot}$$

Outline of Lecture 17

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 $x \in Var$):

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

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**:
$$\begin{aligned} a \mapsto (a_1, \ldots, a_n) &:= a \mapsto a_1 * \ldots * a + n - 1 \mapsto a_n \\ a \mapsto - &:= \exists x : a \mapsto x \\ \exists x : A &:= \neg(\forall x : \neg A) \\ A_1 \Rightarrow A_2 &:= \neg A_1 \vee A_2 \\ A_1 \iff A_2 &:= A_1 \Rightarrow A_2 \wedge A_2 \Rightarrow A_1 \\ a_1 \geq a_2 &:= a_1 > a_2 \vee a_1 = a_2 \\ &\vdots \end{aligned}$$
- **Remark:** recursive predicates for data structures will be introduced later

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:

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

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

Examples

Example 17.9

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

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

Examples

Example 17.9

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

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

Conversely,

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

Examples

Example 17.9

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

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

   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,

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

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