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/

Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

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 $\mathtt{y}$.

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)

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
$$\mathit{Stack} := \{s \mid s : \mathit{Var} \to \mathbb{Z}\}$$
- 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 (Execution relation for statements)

The execution relation $\to \subseteq (\mathit{Cmd} \times \mathit{Stack} \cup \{\lightning\}) \times (\mathit{Stack} \cup \{\lightning\})$ 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 (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!

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 $x \in Var$):

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

- **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 (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

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

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 Predicate Definitions I

Definition 18.2 (SL predicates)

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

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

  where P is a predicate symbol, $n \in \mathbb{N}$, $x_1, \ldots, x_n \in \mathit{Var}$ and $A \in \mathit{SLA}$ an SL assertion, additionally containing recursive predicate calls of the form $P(a_1, \ldots, a_n)$ with $a_1, \ldots, a_n \in \mathit{AExp}$. Syntactic restriction (to ensure monotonicity): each call must be in the scope of an even number of negations.
- It induces the functional $\Phi : (\mathbb{Z}^n \to 2^\Sigma) \to (\mathbb{Z}^n \to 2^\Sigma)$, given by

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

- A state $(s, h) \in \Sigma$ satisfies a predicate call $P(a_1, \ldots, a_n)$ (notation: $(s, h) \models P(a_1, \ldots, a_n)$) if $(s, h) \in \mathrm{fix}(\Phi)(\mathfrak{A}[\![a_1]\!]s, \ldots, \mathfrak{A}[\![a_1]\!]s)$.

Recursive Predicate Definitions II

The previous definition is justified by the following properties:

Lemma 18.3

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

Proof.

omitted □

Thus, Theorem 8.1 is applicable and yields

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

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

Recursive Predicate Definitions III

Example 18.4

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

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

Fixpoint iteration:

$$\begin{aligned}
\Phi^0(p_\emptyset)(x, y) &= \emptyset \\
\Phi^1(p_\emptyset)(x, y) &= \{(s, h) \in \Sigma \mid (s, h) \models (x = y \wedge \mathsf{emp}) \vee \exists x' : x \mapsto x' * \Phi^0(p_\emptyset)(x', y)\} \\
&= \{(s, h) \in \Sigma \mid s(x) = s(y) \wedge h = h_\emptyset\} \\
\Phi^2(p_\emptyset)(x, y) &= \{(s, h) \in \Sigma \mid (s, h) \models (x = y \wedge \mathsf{emp}) \vee \exists x' : x \mapsto x' * \Phi^1(p_\emptyset)(x', y)\} \\
&= \{(s, h) \in \Sigma \mid (s, h) \models (x = y \wedge \mathsf{emp}) \vee \exists x' : x \mapsto x' * (x' = y \wedge \mathsf{emp})\} \\
&= \{(s, h) \in \Sigma \mid (s(x) = s(y) \wedge h = h_\emptyset) \vee h = h_\emptyset[s(x) \mapsto s(y)]\} \\
\Phi^3(p_\emptyset)(x, y) &= \{(s, h) \in \Sigma \mid (s, h) \models (x = y \wedge \mathsf{emp}) \vee \exists x' : x \mapsto x' * \Phi^2(p_\emptyset)(x', y)\} \\
&= \{(s, h) \in \Sigma \mid (s(x) = s(y) \wedge h = h_\emptyset) \vee h = h_\emptyset[s(x) \mapsto s(y)] \vee \\
&\qquad \exists x' \in \mathbb{Z} : h = h_\emptyset[s(x) \mapsto s(x'), s(x') \mapsto s(y)]\} \\
&\vdots
\end{aligned}$$

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. $\langle c, (s, h)\rangle \not\to \lightning$ and
  2. if $\langle c, (s, h)\rangle \to (s', h')$, 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 I

Definition 18.6 (SL proof rules)

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

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

$$\text{(asgn)}\ \frac{x \notin FV(a)}{\{\text{emp}\}\ x := a\ \{\text{emp} \wedge x = a\}} \qquad \text{(frame)}\ \frac{\{A\}\ c\ \{B\} \quad Var(C) \cap Mod(c) = \emptyset}{\{A * C\}\ c\ \{B * C\}}$$

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

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

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

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

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

  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)):

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

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

Pointer Swap Example

Example 18.7 (Pointer swap)

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

$$
\text{(lookup)}\ \frac{x \notin FV(a)}{\{a \mapsto v\}\, x := [a]\, \{a \mapsto v \wedge x = v\}}
\qquad
\text{(mutate)}\ \frac{}{\{a \mapsto -]\}\, [a] := a'\, \{a \mapsto a'\}}
\qquad
\text{(frame)}\ \frac{\{A\}}{}
$$