Semantics and Verification of Software

Summer Semester 2019

Lecture 19: Separation Logic III (Soundness) & Wrap-Up

Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University

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

- $(s, h) \models emp$ if $\text{dom}(h) = \emptyset$
- $(s, h) \models a \mapsto a'$ if $h = h_0[a]s \mapsto [a']s$
- $(s, h) \models A_1 \cdot A_2$ if $\exists h_1, h_2 \in \text{Heap} : h = h_1 \uplus h_2$
- $(s, h) \models \forall x : A$ if $\forall z \in \mathbb{Z} : (s[x \mapsto z], h) \models A$
- $(s, h) \models true$
- $(s, h) \models a_1 = a_2$ if $\mathcal{A}[a_1]s = \mathcal{A}[a_2]s$
- $(s, h) \models a_1 > a_2$ if $\mathcal{A}[a_1]s > \mathcal{A}[a_2]s$
- $(s, h) \models \neg A$ if not $(s, h) \models A$
- $(s, h) \models A_1 \land A_2$ if $(s, h) \models A_1$ and $(s, h) \models A_2$
- $(s, h) \models A_1 \lor A_2$ if $(s, h) \models A_1$ or $(s, h) \models A_2$

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

3 of 27 Semantics and Verification of Software  
Summer Semester 2019  
Lecture 19: Separation Logic III (Soundness) & Wrap-Up
## Recap: Separation Logic

### Recursive Predicate Definitions

**Definition (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, \( n \in \mathbb{N} \), \( x_1, \ldots, x_n \in Var \) and \( A \in SLA \) an SL assertion, additionally containing recursive predicate calls of the form \( P(a_1, \ldots, a_n) \) with \( a_1, \ldots, a_n \in AExp \). 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

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

- 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 \text{fix}(\Phi)\(A[a_1]s\), \ldots, A[a_1]s\)).
Recap: Separation Logic

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 (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. $\langle c, (s, h) \rangle \not\rightarrow \bot$ 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 $(s, h) \in \Sigma$. 
Recap: Separation Logic

The Proof System

Definition (SL proof rules)

\[
\begin{align*}
\text{(alloc)} & \quad x \notin FV(\bar{a}) \quad \frac{}{\{\text{emp}\} x := \text{alloc}(\bar{a}) \{x \mapsto \bar{a}\}} \\
\text{(free)} & \quad \frac{}{\{a \mapsto \bot\} \text{ free}(a) \{\text{emp}\}} \\
\text{(asgn)} & \quad x \notin FV(a) \quad \frac{}{\{\text{emp}\} x := a \{\text{emp} \land x = a\}} \\
\text{(skip)} & \quad \frac{}{\{A\} \text{ skip} \{A\}} \\
\text{(if)} & \quad \frac{}{\{A \land b\} c_1 \{B\} \quad \{A \land \neg b\} c_2 \{B\}} \\
\text{(while)} & \quad \frac{}{\{A\} \text{ while } b \text{ do } c \text{ end} \{A \land \neg b\}} \\
\text{(cons)} & \quad \frac{\Rightarrow (A \Rightarrow A')}{}{\{A'\} c \{B'\}} \\
\end{align*}
\]
A List Reversal Example

List Reversal Example I

Example 19.1 (List reversal: $\text{sll}(x, y) := (x = y \land \text{emp}) \lor \exists x' : x \mapsto x' \ast \text{sll}(x', y)$)

$$\{\text{head} \neq 0 \land \text{sll(\text{head}, 0)}\}$$

$\Rightarrow \{\text{head} \neq 0 \land \exists x' : \text{head} \mapsto x' \ast \text{sll}(x', 0)\}$

$\text{cur} := [\text{head}]$;

$$\{\text{head} \neq 0 \land \exists x' : \text{head} \mapsto x' \ast \text{sll}(x', 0) \land \text{cur} = x'\}$$

$\text{rev} := \text{head}$;

$$\{\text{head} \neq 0 \land \exists x' : \text{head} \mapsto x' \ast \text{sll}(x', 0) \land \text{cur} = x' \land \text{rev} = \text{head}\}$$

$\Rightarrow \{\text{sll}(\text{cur}, 0) \ast \text{sll(\text{rev}, \text{head})}\}$

while $\neg (\text{cur} = 0)$ do

$\text{next} := [\text{cur}]$;

$[\text{cur}] := \text{rev}$;

$\text{rev} := \text{cur}$;

$\text{cur} := \text{next}$

end

$$\{\text{sll(\text{rev}, \text{head})}\}$$

Precondition (unfold sll)

(lookup)

(asgn)

Invariant

Postcondition
Example 19.1 (List reversal: \( \text{sll}(x, y) := (x = y \land \text{emp}) \lor \exists x' : x \mapsto x' \ast \text{sll}(x', y) \))

\[
\begin{align*}
{sll}(cur, 0) \ast sll(rev, head) \\
\text{while } \neg (cur = 0) \text{ do} \\
\{sll(cur, 0) \ast sll(rev, head) \land cur \neq 0\} \\
\Rightarrow \exists x' : cur \mapsto x' \ast sll(x', 0) \ast sll(rev, head) \\
\text{next := [cur];} \\
\{\exists x' : cur \mapsto x' \ast sll(x', 0) \ast sll(rev, head) \land next = x'\} \\
[cur] := rev; \\
\{\exists x' : cur \mapsto rev \ast sll(x', 0) \ast sll(rev, head) \land next = x'\} \\
\Rightarrow \{sll(next, 0) \ast sll(cur, head)\} \\
\text{rev := cur;} \\
\{sll(next, 0) \ast sll(cur, head) \land rev = cur\} \\
\Rightarrow \{sll(next, 0) \ast sll(rev, head)\} \\
\text{cur := next;} \\
\{sll(next, 0) \ast sll(rev, head) \land cur = next\} \\
\Rightarrow \{sll(cur, 0) \ast sll(rev, head)\} \\
\text{end} \\
\{sll(cur, 0) \ast sll(rev, head) \land cur = 0\} \\
\Rightarrow \{sll(rev, head)\}
\end{align*}
\]
Theorem 19.2 (Soundness of Separation Logic)

For every partial correctness property \( \{ A \} \ c \ \{ B \} \) with \( A, B \in SLA \) and \( c \in Cmd \),

\[ \vdash \{ A \} \ c \ \{ B \} \quad \Rightarrow \quad \models \{ A \} \ c \ \{ B \} . \]

Proof.

We only consider the frame rule and use the auxiliary lemmas on the following slide.
Soundness of Separation Logic

Soundness of Frame Rule

The following operational facts about programs imply their locality:

**Lemma 19.3 (Monotonicity of memory safety)**

If \( \langle c, (s, h_1) \rangle \not\rightarrow \not\downarrow \) and \( h_1 \neq h_2 \), then \( \langle c, (s, h_1 \cup h_2) \rangle \not\rightarrow \not\downarrow \).

**Proof.**

Straightforward (increasing the heap cannot introduce memory faults, which are only caused by accessing unallocated memory addresses)

**Lemma 19.4 (Frame property)**

Suppose \( \langle c, (s, h_1) \rangle \not\rightarrow \not\downarrow \) and \( \langle c, (s, h_1 \cup h_2) \rangle \rightarrow (s', h') \).

Then there exists \( h'_1 \in \text{Stack} \) such that \( \langle c, (s, h_1) \rangle \rightarrow (s', h'_1) \) and \( h' = h'_1 \cup h_2 \).

**Proof.**

on the board
More Topics in Separation Logic

- Completeness of proof system
- Symbolic heaps, symbolic execution and abstract interpretation [P.W. O’Hearn: *A Primer on Separation Logic (and Automatic Program Verification and Analysis)*]
- Frame inference [A. Gotsman, J. Berdine, B. Cook: *Interprocedural Shape Analysis with Separated Heap Abstractions*]
  - Problem: given $A, B \in SLA$, find $C \in SLA$ such that $\models A \Rightarrow B \ast C$
  - Application: $A$ assertion before procedure call, $B$ precondition of procedure specification
  - Goal: find “leftover” heap $C$ needed for frame rule applied to procedure call
- Concurrent Separation Logic and permissions [V. Vafeiadis: *Concurrent Separation Logic and Operational Semantics*]
- ...
Operational Semantics of Functional Programming Languages

- Program = list of function definitions
- Simplest setting: first-order function definitions of the form

\[ f(x_1, \ldots, x_n) = t \]

- function name \( f \)
- formal parameters \( x_1, \ldots, x_n \)
- term \( t \) over (base and defined) function calls and \( x_1, \ldots, x_n \)

- Operational semantics (only function calls; for terms \( t_i \), numbers \( z_j \) and variables \( x_k \))
  - call-by-value case:

\[
\begin{align*}
  t_1 & \rightarrow z_1 \\
  \ldots & \\
  t_n & \rightarrow z_n \\
  t[x_1 \mapsto z_1, \ldots, x_n \mapsto z_n] & \rightarrow z \\
\end{align*}
\]

\[ f(t_1, \ldots, t_n) \rightarrow z \]

  - call-by-name case:

\[
\begin{align*}
  t[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n] & \rightarrow z \\
\end{align*}
\]

\[ f(t_1, \ldots, t_n) \rightarrow z \]
Outlook: Semantics of Functional Programming Languages

Denotational Semantics of Functional Programming Languages

- Denotational semantics
  - program = equation system (for functions)
  - induces call-by-value and call-by-name functional
  - monotonic and continuous w.r.t. graph inclusion
  - semantics := least fixpoint (Tarski/Knaster Theorem)
  - coincides with operational semantics

- Extensions: higher-order types, data types, ...

- see [Winskel 1996, Sct. 9] and Functional Programming course [Giesl]
Syntax of Logic Programming Languages

- **Program** = list of predicate definitions
- **Predicate definition** = sequence of clauses of the form $q_0: -q_1, \ldots, q_n$ with atoms $q_i$
- **Atom** = predicate call $p(t_1, \ldots, t_k)$ with predicate $p$ and terms $t_j$ over variables, constants and function symbols

Example 19.1

```prolog
father(tom, sally).
father(tom, erica).
father(mike, tom).
mother(anna, sally).
sibling(X, Y) :- parent(Z, X), parent(Z, Y).
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
```
Outlook: Semantics of Logic Programming Languages

Operational Semantics of Logic Programming Languages

- Defined by (SLD) resolution
- Starts with single goal, called query
- Iteratively apply clauses with matching heads
- Involves backtracking if several clause heads match

Example 19.2

<table>
<thead>
<tr>
<th>father(tom,sally).</th>
</tr>
</thead>
<tbody>
<tr>
<td>father(tom,erica).</td>
</tr>
<tr>
<td>father(mike,tom).</td>
</tr>
<tr>
<td>mother(anna,sally).</td>
</tr>
<tr>
<td>sibling(X,Y) :- parent(Z,X), parent(Z,Y).</td>
</tr>
<tr>
<td>parent(X,Y) :- mother(X,Y).</td>
</tr>
<tr>
<td>parent(X,Y) :- father(X,Y).</td>
</tr>
</tbody>
</table>

Refutation proof:

<table>
<thead>
<tr>
<th>sibling(sally,erica).</th>
</tr>
</thead>
<tbody>
<tr>
<td>⇐ parent(Z,sally), parent(Z,erica).</td>
</tr>
<tr>
<td>⇐ mother(Z,sally), parent(Z,erica).</td>
</tr>
<tr>
<td>⇐ parent(anna,erica).</td>
</tr>
<tr>
<td>⇐ mother(anna,erica). ⨿</td>
</tr>
<tr>
<td>⇐ father(anna,erica). ⨿</td>
</tr>
<tr>
<td>⇐ father(Z,sally), parent(Z,erica).</td>
</tr>
<tr>
<td>⇐ parent(tom,erica).</td>
</tr>
<tr>
<td>⇐ mother(tom,erica). ⨿</td>
</tr>
<tr>
<td>⇐ father(tom,erica).</td>
</tr>
<tr>
<td>⇐ □</td>
</tr>
</tbody>
</table>
Denotational Semantics of Logic Programming Languages

- meaning of program = \{ fully instantiated valid atoms \}
- fixpoint iteration:
  - start with empty set
  - 1st step: all instantiations of facts (i.e., clauses with empty RHS)
  - \( i + 1 \)st step: all instantiations of facts that can be derived from known facts
- monotonic and continuous w.r.t. set inclusion
- semantics := least fixpoint (Tarski/Knaster Theorem)
- coincides with operational semantics

Example 19.3

\[
\begin{align*}
\text{father}(\text{tom}, \text{sally}). & \\
\text{father}(\text{tom}, \text{erica}). & \\
\text{father}(\text{mike}, \text{tom}). & \\
\text{mother}(\text{anna}, \text{sally}). & \\
\text{Sibling}(\text{X}, \text{Y}) & : = \text{parent}(\text{Z}, \text{X}), \text{parent}(\text{Z}, \text{Y}). \\
\text{parent}(\text{X}, \text{Y}) & : = \text{mother}(\text{X}, \text{Y}). \\
\text{parent}(\text{X}, \text{Y}) & : = \text{father}(\text{X}, \text{Y}).
\end{align*}
\]

Fixpoint iteration:
\[
\begin{align*}
A_0 & = \emptyset \\
A_1 & = \{ \text{f(t,s)}, \text{f(t,e)}, \text{f(m,t)}, \text{m(a,s)} \} \\
A_2 & = A_1 \cup \{ \text{p(t,s)}, \text{p(t,e)}, \text{p(m,t)}, \text{p(a,s)} \} \\
A_3 & = A_2 \cup \{ \text{s(s,e)}, \text{s(e,s)} \} \\
A_4 & = A_3
\end{align*}
\]
## Summary

### Summary I

### Semantics of programming languages

- Models the **computational meaning** of each program
- Provides abstract entities that represent just the **relevant features** of all possible executions
  - relationship between input and output
  - whether execution terminates or not
- Ignores details that have no relevance to the correctness of implementations
- **Equivalences** identify programs with identical semantics

### (Structural) operational semantics

- Uses syntax-guided rules to specify **transition relations** (for evaluation/execution)
- Represents **scoping** of identifiers by splitting of data state into environment and store part
- **Big-step** and **small-step semantics** (latter required for modelling interaction between concurrent activities)
<table>
<thead>
<tr>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary II</td>
</tr>
</tbody>
</table>

## Denotational semantics
- Denotations are defined **inductively** on program structure
- Recursion (loops, procedures) handled as **least fixpoints** of continuous functions on CCPOs

## Axiomatic semantics
- A **Hoare Logic** gives rules for the relation between assertions about values of variables before and after execution of each construct
- Employs **logical variables** for remembering (initial) values of program variables
- **Partial vs. total** correctness
- Recursion handled by **invariants**

## Separation Logic
- Extension of Hoare Logic to deal with **pointer programs**
- Crucial: **frame rule** to support scalable modular reasoning
### Miscellaneous

### Oral Exam

- **Appointment via Foodle poll** at [https://terminplaner4.dfn.de/mW5s1bLwrApFk2Qs](https://terminplaner4.dfn.de/mW5s1bLwrApFk2Qs)

- **Contents:**
  - foundational concepts and connections between those
  - proof ideas
  - concrete (elaborated) examples
  - details of proofs
  - willing to omit correctness properties for execution time

- **Typical (non-)questions:**
  - operational/denotational/axiomatic semantics of `while` statement
  - alternative evaluation strategies for boolean expressions strict/sequential/parallel
  - algebraic foundations (partial orders, CCPOs, monotonicity, continuity, fixpoint theorem, ...)
  - idea of coincidence proof for operational/denotational semantics
  - (mild) extensions of programming language (side effects, ...)
  - Construct the derivation tree for `c := ...` (many lines) and `σ := ...`
  - Prove the following claim: ...
Master-Level Teaching in Winter 2019/20

Course **Model Checking** [Katoen]

- Interleaving semantics (process algebras)
- Property specifications (Hennessy-Milner Logic)
- Process equivalence (traces, bisimulation)
- True concurrency (Petri nets)

Course **Concurrency Theory** [Katoen/Noll]

1. Interleaving semantics (process algebras)
2. Property specifications (Hennessy-Milner Logic)
3. Process equivalence (traces, bisimulation)
4. True concurrency (Petri nets)

Seminar **[Advanced Topics in] Formal Semantics of Programming Languages** [Katoen, Noll, NN]

- Nondeterminism
- Concurrency
- Recursion
- Relaxed memory models
- Security and privacy