CS6202: Advanced Topics in Programming Languages and Systems

Lecture 8/9: **Separation Logic**

- Overview
- Assertion Logic
- Semantic Model
- Hoare-style Inference Rules
- Specification and Annotations
- Linked List and Segments
- Trees and Intuitionistic Logic
- (above from John Reynold’s mini-course)
- Automated Verification

---

**Motivation**

Program reasoning is important for:

- correctness of software
- safety (fewer or no bugs)
- performance guarantee
- optimization

---

**Hoare Logic**

Can handle reasoning of imperative programs well.

Notation:

\[ \{ P \} \text{ code } \{ Q \} \]

\( \{ P \} \) precondition before executing code

\( \{ Q \} \) postcondition after executing code

Some examples:

\( \{ x=1 \} \ x:=x+1 \ \{ x=2 \} \)

\( \{ x=x_0 \} \ x:=x+1 \ \{ x=x_i+1 \} \)

\( \{ Q[x+1/x] \} \ x:=x+1 \ \{ Q \} \)

\( \{ P \} \ x:=x+1 \ \{ \exists x_i. P[x_i/x] \land x=x_i+1 \} \)

---

**Problem**

Hoare logic can handle program variables but not heap objects well due to aliasing problems.

Consider an in-place list reversal algorithm

\[ j := \text{nil} \ ; \ \text{while } i \neq \text{nil do} \ (k := [i + 1] \ ; \ [i + 1] := j \ ; \ j := i \ ; \ i := k) \]

\([i]\) denotes a heap location at address i
**Loop Invariant**

Loop invariant is a statement that holds at the beginning of each iteration of the loop.

An inadequate invariant:

\[ \exists \alpha, \beta. \text{list } \alpha \ i \land \text{list } \beta \ j \land \alpha_0^{\top} = \alpha^{\top} \beta, \]

where

\[ \text{list } \epsilon \ i \overset{\text{def}}{=} \text{nil} \]

\[ \text{list}(a \cdot \alpha) \ i \overset{\text{def}}{=} \exists j. \ i \leftarrow a, j \land \text{list } \alpha \ j \]

heap predicate relates a list of elements and a pointer

---

**Basics of Separation Logic**

- Program specification and proof
  - Extension of Hoare logic
  - Separating (independent, spatial) conjunction (\(*\)) and implication (\(\rightarrow\))
- Inductive definitions over abstract structures

---

**Loop Invariant**

An adequate invariant:

\[ (\exists \alpha, \beta. \text{list } \alpha \ i \land \text{list } \beta \ j \land \alpha_0^{\top} = \alpha^{\top} \beta) \]

\[ \land (\forall k. \text{reachable}(i, k) \land \text{reachable}(j, k) \Rightarrow k = \text{nil}), \]

where

\[ \text{reachable}(i, j) \overset{\text{def}}{=} \exists n \geq 0. \text{reachable}_{n}(i, j) \]

\[ \text{reachable}_{0}(i, j) \overset{\text{def}}{=} i = j \]

\[ \text{reachable}_{n+1}(i, j) \overset{\text{def}}{=} \exists a, k. i \leftarrow a, k \land \text{reachable}_{n}(k, j). \]

in separation logic:

\[ (\exists \alpha, \beta. \text{list } \alpha \ i \ast \text{list } \beta \ j) \land \alpha_0^{\top} = \alpha^{\top} \beta \]

---

**Simple Language with Heap Store**

The simple imperative language:

\[ := \text{ skip } ; \text{ if } - \text{ then } - \text{ else } - \text{ while } - \text{ do } - \]

plus:

\[ \text{Store} : x: 3, y: 4 \]

\[ \text{Heap} : \text{empty} \]

\[ \text{Allocation} \quad x := \text{cons}(1, 2) ; \]

\[ \text{Store} : x: 37, y: 4 \]

\[ \text{Heap} : 37:1, 38:2 \]

\[ \text{Lookup} \quad y := [x] ; \]

\[ \text{Store} : x: 37, y: 1 \]

\[ \text{Heap} : 37:1, 38:2 \]

\[ \text{Mutation} \quad [x + 1] := 3 ; \]

\[ \text{Store} : x: 37, y: 1 \]

\[ \text{Heap} : 37:1, 38:3 \]

\[ \text{Deallocation} \quad \text{dispose}(x + 1) \]

\[ \text{Store} : x: 37, y: 1 \]

\[ \text{Heap} : 37:1 \]
**Memory Faults**

Can be caused by out of range look up of memory.

<table>
<thead>
<tr>
<th>Store</th>
<th>x: 3, y: 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>empty</td>
</tr>
</tbody>
</table>

Allocation  

\[ x := \text{cons}(1, 2) \]

<table>
<thead>
<tr>
<th>Store</th>
<th>x: 37, y: 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>37: 1, 38: 2</td>
</tr>
</tbody>
</table>

Lookup  

\[ y := [x]; \]

<table>
<thead>
<tr>
<th>Store</th>
<th>x: 37, y: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>37: 1, 38: 2</td>
</tr>
</tbody>
</table>

Mutation  

\[ [x + 2] := 3; \]

abort

**Semantic Model**

When \( s \) is a store, \( h \) is a heap, and \( p \) is an assertion whose free variables all belong to the domain of \( s \), we write

\[ s, h \models p \]

to indicate that the state \( s, h \) satisfies \( p \), or \( p \) is true in \( s, h \), or \( p \) holds in \( s, h \). Then:

\[ s, h \models b \iff \llbracket b \rrbracket_{\text{bool}} s = \text{true}, \]

\[ s, h \models \neg p \iff s, h \not\models p \text{ is false}, \]

\[ s, h \models p_0 \land p_1 \iff s, h \models p_0 \text{ and } s, h \models p_1 \]

(and similarly for \( \lor, \Rightarrow, \Leftrightarrow \)).

**Assertion Language**

Standard predicate calculus:

\[ \land \quad \land \quad \neg \quad \Rightarrow \quad \forall \quad \exists \]

plus:

- **emp**  
  The heap is empty.

- **e \mapsto e'**  
  The heap contains one cell, at address \( e \) with contents \( e' \).

- **p_1 \ast p_2**  
  The heap can be split into two disjoint parts such that \( p_1 \) holds for one part and \( p_2 \) holds for the other.

- **p_1 \leftarrow p_2**  
  If the current heap is extended with a disjoint part in which \( p_1 \) holds, then \( p_2 \) holds for the extended heap.

**Semantic Model**

\[ s, h \models \forall v. \ p \iff \forall x \in \mathbb{Z} \ [ s \mid v; x ], h \models p, \]

\[ s, h \models \exists v. \ p \iff \exists x \in \mathbb{Z} \ [ s \mid v; x ], h \models p, \]

\[ s, h \models \text{emp} \iff \text{dom} \ h = \{ \}, \]

\[ s, h \models e \mapsto e' \iff \text{dom} \ h = \{ [e]_{\text{exp}} s \} \text{ and } h([e]_{\text{exp}} s) = [e']_{\text{exp}} s, \]

\[ s, h \models p_0 \ast p_1 \iff \exists h_0, h_1. \ h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h \text{ and } s, h_0 \not\models p_0 \text{ and } s, h_1 \not\models p_1, \]

\[ s, h \models p_0 \leftarrow p_1 \iff \forall h'. (h' \perp h \text{ and } s, h' \not\models p_0) \text{ implies } s, h \cdot h' \not\models p_1. \]
### Separation Logic

#### Separation Conjunction - Examples

1. \( x \rightarrow 3, y \)
   - Store: \( x: \alpha, y: \beta \)
   - Heap: \( \alpha: 3, \alpha + 1: \beta \)
2. \( y \leftarrow 3, x \)
   - Store: \( x: \alpha, y: \beta \)
   - Heap: \( \beta: 3, \beta + 1: \alpha \)
3. \( x \rightarrow 3, y \land y \leftarrow 3, x \)
   - Store: \( x: \alpha, y: \beta \)
   - Heap: \( \alpha: 3, \alpha + 1: \beta, \beta: 3, \beta + 1: \alpha \)
   - where \( \alpha, \alpha + 1, \beta, \beta + 1 \) are distinct

#### Separation Implication - Examples

\( p_1 \rightarrow p_2 \)
- If the current heap extended with a disjoint part in which \( p_1 \) holds, then \( p_2 \) holds for the extended heap.

Suppose \( p \) holds for
- Store: \( x: \alpha, \ldots \)
- Heap: \( \alpha: 3, 3, \alpha + 1: 4, \text{rest of heap} \)
- Rest of Heap

Then \( (x \leftarrow 3, 4) \rightarrow p \) holds for
- Store: \( x: \alpha, \ldots \)
- Heap: \( \alpha: 1, 3, 4, \text{rest of heap} \)
- Rest of Heap

and \( x \leftarrow 1, 2 \rightarrow ((x \leftarrow 3, 4) \rightarrow p) \) holds for
- Store: \( x: \alpha, \ldots \)
- Heap: \( \alpha: 1, 1, 2, \text{rest of heap} \)
- Rest of Heap

#### Conjunction - Examples

Conjunction describes the same heap space.

4. \( x \rightarrow 3, y \land y \leftarrow 3, x \)
   - Store: \( x: \alpha, y: \alpha \)
   - Heap: \( \alpha: 3, \alpha + 1: \alpha \)
5. \( x \leftarrow 3, y \land y \leftarrow 3, x \)
   - Store: \( x: \alpha, y: \beta \)
   - Heap: \( \alpha: 3, \alpha + 1: \beta, \beta: 3, \beta + 1: \alpha, \ldots \)
   - As in (3) or (4), possibly with additional cells

#### Inference Rules

Reasoning with normalization, weakening and strengthening.

\[
\begin{align*}
    p_0 & * p_1 \Rightarrow p_1 * p_0 \\
    (p_0 * p_1) & * p_2 \Leftarrow p_0 * (p_1 * p_2) \\
    p & * \text{emp} \Leftarrow p \\
    (p_0 \lor p_1) & * q \Leftarrow (p_0 * q) \lor (p_1 * q) \\
    (p_0 \land p_1) & * q \Rightarrow (p_0 * q) \land (p_1 * q) \\
    (\exists x. p_0) & * p_1 \Leftarrow \exists x. (p_0 * p_1) \quad \text{when } x \text{ not free in } p_1 \\
    (\forall x. p_0) & * p_1 \Rightarrow \forall x. (p_0 * p_1) \quad \text{when } x \text{ not free in } p_1 \\
    p_0 & \Rightarrow p_1 \quad q_0 \Rightarrow q_1 \\
    p_0 * q_0 & \Rightarrow p_1 * q_1 \quad \text{(monotonicity)} \\
    \frac{p_0 * p_1 \Rightarrow p_2}{p_0 \Rightarrow (p_1 \rightarrow p_2)} \quad \text{(currying)} \\
    \frac{p_0 \Rightarrow (p_1 \rightarrow p_2)}{p_0 * p_1 \Rightarrow p_2} \quad \text{(decurrying)}
\end{align*}
\]
**Pure Assertion**

- An assertion is *pure* iff, for any store, it is independent of the heap.
- Syntactically, an assertion is pure if it does not contain `emp`, `⇒`, or `⇐`.

Axiom schematic guided by pure formulae

\[
\begin{align*}
    p_0 \land p_1 &\Rightarrow p_0 \ast p_1 & \text{when } p_0 \text{ or } p_1 \text{ is pure} \\
    p_0 \ast p_1 &\Rightarrow p_0 \land p_1 & \text{when } p_0 \text{ and } p_1 \text{ are pure} \\
    (p \land q) \ast r &\Leftrightarrow (p \ast r) \land q & \text{when } q \text{ is pure} \\
    (p_0 \Rightarrow p_1) &\Rightarrow (p_0 \Rightarrow p_1) & \text{when } p_0 \text{ is pure} \\
    (p_0 \Rightarrow p_1) &\Rightarrow (p_0 \ast p_1) & \text{when } p_0 \text{ and } p_1 \text{ are pure}.
\end{align*}
\]

**Partial Correctness Specification**

\[
\{p\} \ c \ \{q\}
\]

is *valid* iff, starting in any state in which \(p\) holds:

- No execution of \(c\) aborts.
- When some execution of \(c\) terminates in a final state, then \(q\) holds in the final state.

**Two Unsound Axiom Schemata**

\[
p \neq p \ast p \quad \text{(Contraction)} \\
\begin{align*}
    e.g. &\quad p : x \mapsto 1
\end{align*}
\]

\[
p \ast q \neq p \quad \text{(Weakening)} \\
\begin{align*}
    e.g. &\quad p : x \mapsto 1 \\
    &\quad q : y \mapsto 2
\end{align*}
\]

---

**Total Correctness Specification**

\[
[p] \ c \ [q]
\]

is *valid* iff, starting in any state in which \(p\) holds:

- No execution of \(c\) aborts.
- Every execution of \(c\) terminates.
- When some execution of \(c\) terminates in a final state, then \(q\) holds in the final state.
Examples of Valid Specifications

\[
\{ x - y > 3 \} \ x := x - y \ \{ x > 3 \} \\
\{ x + y \geq 17 \} \ x := x + 10 \ \{ x + y \geq 27 \} \\
\{ \text{emp} \} \ x := \text{cons}(1, 2) \ \{ x \mapsto 1, 2 \} \\
\{ x \mapsto 1, 2 \} \ y := [x] \ \{ x \mapsto 1, 2 \land y = 1 \} \\
\{ x \mapsto 1, 2 \land y = 1 \} \ [x + 1] := 3 \ \{ x \mapsto 1, 3 \land y = 1 \} \\
\{ x \mapsto 1, 3 \land y = 1 \} \ \text{dispose} \ x \ \{ x + 1 \mapsto 3 \land y = 1 \} \\
\{ x \leq 10 \} \ \text{while} \ x \neq 10 \ \text{do} \ x := x + 1 \ \{ x = 10 \} \\
\{ \text{true} \} \ \text{while} \ x \neq 10 \ \text{do} \ x := x + 1 \ \{ x = 10 \} \\
\{ x > 10 \} \ \text{while} \ x \neq 10 \ \text{do} \ x := x + 1 \ \{ \text{false} \}
\]

Hoare Inference Rules

Structural rules are applicable to any commands.

\[
\frac{p \Rightarrow q \quad \{ q \} \ c \ \{ r \}}{\{ p \} \ c \ \{ r \}} \quad \text{Strengthening Precedent (SP)}
\]

\[
\frac{\{ p \} \ c \ \{ q \} \quad q \Rightarrow r}{\{ p \} \ c \ \{ r \}} \quad \text{Weakening Consequent (WC)}
\]

Partial Correctness of While Loop

\[
\frac{\{ i \land b \} \ c \ \{ i \}}{\{ i \} \ \text{while} \ b \ \text{do} \ c \ \{ i \land \neg b \}}
\]

Here \( i \) is the invariant.

An Instance

\[
\{ y = 2^k \land k \leq n \land k \neq n \} \ k := k + 1 ; y := 2 \times y \ \{ y = 2^k \land k \leq n \} \\
\{ y = 2^k \land k \leq n \} \\
\text{while} \ k \neq n \ \text{do} \ (k := k + 1 ; y := 2 \times y) \\
\{ y = 2^k \land k \leq n \land \neg k \neq n \}
\]
**Total Correctness of While Loop**

\[
\begin{align*}
& [i \land b \land e = v_0] \quad [i \land e < v_0] \\
& \quad (i \land b) \Rightarrow e \geq 0 \\
\end{align*}
\]

\[
\begin{align*}
& \text{when } v_0 \text{ does not occur free in } i, b, c, \text{ or } e.
\end{align*}
\]

\[
\begin{align*}
& [x \leq 10] \quad \textbf{while} \quad x \neq 10 \quad \textbf{do} \quad x := x + 1 \\
& \quad [x = 10]
\end{align*}
\]

**Hoare Inference Rules**

Variable Declaration (DC)

\[
\begin{align*}
& \{p\} \quad c \quad \{q\} \\
& \{p\} \quad \text{newvar } v \quad \text{in} \quad \{q\}
\end{align*}
\]

when \( v \) does not occur free in \( p \) or \( q \).

Here the requirement on the declared variable \( v \) formalizes the concept of *locality*, i.e., that the value of \( v \) when \( c \) begins execution has no effect on this execution, and that the value of \( v \) when \( c \) finishes execution has no effect on the rest of the program.

**Annotated Specifications**

In annotated specifications, additional assertions called *annotations* are placed in command in such a way that it assist proof construction process.

Examples:

**Conditional (CD)**

\[
\begin{align*}
& \{p \land b\} \quad c_1 \quad \{q\} \\
& \{p \land \neg b\} \quad c_2 \quad \{q\} \\
\end{align*}
\]

\[
\begin{align*}
& \{p\} \quad \text{if } b \quad \text{then} \quad c_1 \quad \text{else} \quad c_2 \quad \{q\}
\end{align*}
\]

**Skip (SK)**

\[
\begin{align*}
& \{p\} \quad \text{skip} \quad \{p\}
\end{align*}
\]

**Sequential Composition (SQAN)**

\[
\begin{align*}
& \{p\} \quad c_1 \quad \{q\} \quad c_2 \quad \{r\} \\
\end{align*}
\]

\[
\begin{align*}
& \{p\} \quad c_1 \; : \; \{q\} \quad c_2 \; \{r\}
\end{align*}
\]

**Strengthening Precedent (SPAN)**

\[
\begin{align*}
& p \Rightarrow q \\
& \{q\} \quad c \quad \{r\} \\
\end{align*}
\]

\[
\begin{align*}
& \{p\} \quad \{q\} \quad c \quad \{r\}
\end{align*}
\]
**Minimal Annotated Specifications**

Should attempt to minimise annotations where possible.

Restrict to pre/post of methods and invariant of loops.

\[
\{ n \geq 0 \} \\
\k := 0 ; y := 1 ; \\
\{ y = 2^k \land k \leq n \} \\
\textbf{while } k \neq n \textbf{ do } (k := k + 1 ; y := 2 \times y) \\
\{ y = 2^n \}
\]

Further advances:  
(i) intraprocedural inference  
(ii) interprocedural inference.

---

**Structural Inference Rules**

**Conjunction (CONJ)**

\[
\frac{\{ p_1 \} \rightarrow q_1 \hspace{1cm} \{ p_2 \} \rightarrow q_2}{\{ p_1 \land p_2 \} \rightarrow q_1 \land q_2}
\]

**Disjunction (DISJ)**

\[
\frac{\{ p_1 \} \rightarrow q_1 \hspace{1cm} \{ p_2 \} \rightarrow q_2}{\{ p_1 \lor p_2 \} \rightarrow q_1 \lor q_2}
\]

---

**Structural Inference Rules**

**Renaming (RN)**

\[
\frac{\{ p \} \rightarrow q}{\{ p' \} \rightarrow q'}
\]

where \( p', c', \) and \( q' \) are obtained from \( p, c, \) and \( q \) by zero or more renamings of bound variables.

**Substitution (SUB)**

\[
\frac{\{ p \} \rightarrow q}{(\{ p \} \rightarrow q)/v_1 \rightarrow e_1, \ldots, v_n \rightarrow e_n}
\]

where \( v_1, \ldots, v_n \) are the variables occurring free in \( p, c, \) or \( q, \) and, if \( v_i \) is modified by \( c, \) then \( e_i \) is a variable that does not occur free in any other \( e_j, \)

---

**Structural Inference Rules**

**Universal Quantification (UQ)**

\[
\frac{\{ p \} \rightarrow q}{\{ \forall v. \ p \} \rightarrow \forall v. \ q},
\]

where \( v \) is not free in \( c. \)

**Existential Quantification (EQ)**

\[
\frac{\{ p \} \rightarrow q}{\{ \exists v. \ p \} \rightarrow \exists v. \ q},
\]

where \( v \) is not free in \( c. \)
**Rule of Constancy from Hoare Logic**

- Rule of Constancy

\[
\frac{\{p\} \ c \ \{q\}}{\{p \land r\} \ c \ \{q \land r\}},
\]

where no variable occurring free in \(r\) is modified by \(c\).

that is *unsound* in separation logic, since, for example

\[
\frac{\{x \leftarrow \_\} \ [x] := 4 \ \{x \leftarrow 4\}}{\{x \leftarrow \_ \land y \leftarrow 3\} \ [x] := 4 \ \{x \leftarrow 4 \land y \leftarrow 3\}}
\]

fails when \(x = y\).

**Local Specifications**

- The *footprint* of a command is the variables and the parts of the heap that are actually used by the command.
- A specification of a command is *local* when it mentions only the footprint.
- By using the frame rule, one can move from local to non-local specifications.

For example,

\[
\frac{\{\text{list} \ \alpha \ i\} \ \text{“Reverse List”} \ \{\text{list} \ \alpha^{\dagger} \ j\}}{\{\text{list} \ \alpha \ i \ast \ \text{list} \ \gamma \ k\} \ \text{“Reverse List”} \ \{\text{list} \ \alpha^{\dagger} \ j \ast \ \text{list} \ \gamma \ k\}}.
\]

**Frame Rule of Separation Logic**

- Frame Rule (O’Hearn) (FR)

\[
\frac{\{p\} \ c \ \{q\}}{\{p \ast r\} \ c \ \{q \ast r\}},
\]

where no variable occurring free in \(r\) is modified by \(c\).

This facilitates local reasoning and specification

**Inference Rules for Mutation**

The local form (MUL):

\[
\frac{}{\{e \leftarrow \_\} \ [e] := e' \ \{e \leftarrow e'\}}.
\]

The global form (MUG):

\[
\frac{}{\{(e \leftarrow \_) \ast r\} \ [e] := e' \ \{(e \leftarrow e') \ast r\}}.
\]

The backward-reasoning form (MUBR):

\[
\frac{}{\{(e \leftarrow \_) \ast ((e \leftarrow e') \rightarrow p)\} \ [e] := e' \ \{p\}}.
\]
**Inference Rules for Deallocation**

The local form (DISL):

\[ \{ e \mapsto - \} \text{dispose } e \{ \text{emp} \}. \]

The global (and backward-reasoning) form (DISG):

\[ \{(e \mapsto -) \star r\} \text{dispose } e \{r\}. \]

One can derive (DISG) from (DISL) by using (FR); one can go in the opposite direction by taking \( r \) to be \( \text{emp} \).

**Inference Rules for Noninterfering Allocation**

The local form (CONSNIL):

\[ \{\text{emp}\} \ v := \text{cons}(c_0, \ldots, c_{n-1}) \{v \mapsto c_0, \ldots, c_{n-1}\}, \]

where \( v \notin \text{FV}(c_0, \ldots, c_{n-1}) \).

The global form (CONSNIG):

\[ \{r\} \ v := \text{cons}(c_0, \ldots, c_{n-1}) \{(v \mapsto c_0, \ldots, c_{n-1}) \star r\}, \]

where \( v \notin \text{FV}(c_0, \ldots, c_{n-1}, r) \).

**Inference Rules for Lookup**

The local form (LKL):

\[ \{v = v' \land (c \mapsto v'')\} \ v := [c] \{v = v'' \land (e' \mapsto v'')\}, \]

where \( v, v', \) and \( v'' \) are distinct, and \( e' \) denotes \( e/v \rightarrow v' \).

The global form (LKG):

\[ \{\exists v''. (e \mapsto v'') \star (r/v' \rightarrow v)\} \ v := [c] \{\exists v'. (e' \mapsto v) \star (r/v'' \rightarrow v)\}, \]

where \( v, v', \) and \( v'' \) are distinct, \( v', v'' \notin \text{FV}(c) \), \( v \notin \text{FV}(r) \), and \( e' \) denotes \( e/v \rightarrow v' \).

The backward-reasoning form (LKBKR):

\[ \{\exists v''. (e \mapsto v'') \land p''\} \ v := [c] \{p\}, \]

where \( v'' \notin \text{FV}(e) \cup \text{FV}(p) - \{v\} \), and \( p'' \) denotes \( p/v \rightarrow v'' \).

**Notation for Sequences**

When \( \alpha \) and \( \beta \) are sequences, we write

- \( \epsilon \) for the empty sequence.
- \( [x] \) for the single-element sequence containing \( x \). (We will omit the brackets when \( x \) is not a sequence.)
- \( \alpha \cdot \beta \) for the composition of \( \alpha \) followed by \( \beta \).
- \( \alpha^\dagger \) for the reflection of \( \alpha \).
- \( \#\alpha \) for the length of \( \alpha \).
- \( \alpha_i \) for the \( i \)th component of \( \alpha \).
**Singly Linked List**

**list α i:**

\[
\begin{array}{c}
\text{i} \\
\alpha_1 \\
\alpha_2 \\
\vdots \\
\alpha_n \\
\text{nil}
\end{array}
\]

is defined by

\[
\text{list } \epsilon \ i \ \text{def} \ \text{emp} \land i = \text{nil}
\]

\[
\text{list } (\alpha \cdot \alpha) \ i \ \text{def} \ \exists j. \ i \mapsto a, j \ast \text{list } \alpha j.
\]

What is the default property (invariant) of this predicate?

---

**Singly Linked List Segment**

**Properties**

\[
\text{lseg } \alpha \ (i, j) \iff i \mapsto a, j
\]

\[
\text{lseg } \alpha \cdot \beta \ (i, k) \iff \exists j. \ \text{lseg } \alpha \ (i, j) \ast \text{lseg } \beta \ (j, k)
\]

\[
\text{lseg } \alpha \cdot b \ (i, k) \iff \exists j. \ \text{lseg } \alpha \ (i, j) \ast j \mapsto b, k
\]

\[
\text{list } \alpha i \iff \text{lseg } \alpha \ (i, \text{nil}).
\]

**Emptyness Conditions**

\[
\text{lseg } \alpha \ (i, j) \Rightarrow (i = \text{nil} \Rightarrow (\alpha = \epsilon \land j = \text{nil}))
\]

\[
\text{lseg } \alpha \ (i, j) \Rightarrow (i \neq j \Rightarrow \alpha \neq \epsilon).
\]

---

**Non-Touching Linked List Segment**

We can define nontouching list segments in terms of \text{lseg}:

\[
\text{ntlseg } \alpha \ (i, j) \ \text{def} \ \text{lseg } \alpha \ (i, j) \land j \mapsto -
\]

or we can define them inductively:

\[
\text{ntlseg } \epsilon \ (i, j) \ \text{def} \ \text{emp} \land i = j
\]

\[
\text{ntlseg } a \cdot \alpha \ (i, k) \ \text{def} \ i \neq k \land i \neq k + 1 \land (\exists j. i \mapsto a, j \ast \text{ntlseg } \alpha \ (j, k)).
\]

**Easier test for emptiness**

\[
\text{ntlseg } \alpha \ (i, j) \Rightarrow (\alpha = \epsilon \Rightarrow i = j)
\]
**Braced List Segment**

A *braced list segment* is a list segment with an interior pointer \( j \) to its last element; in the special case where the list segment is empty, \( j \) is \( \text{nil} \). Formally,

\[
\text{brlseg} \; \epsilon (i, j, k) \overset{\text{def}}{=} \text{emp} \land i = k \land j = \text{nil} \\
\text{brlseg} \; \alpha\cdot a (i, j, k) \overset{\text{def}}{=} \text{lseg} \; \alpha (i, j) \times j \mapsto a, k.
\]

**Doubly Linked List**

\[
dlseg \; \alpha (i, i', j, j'):
\]

is defined by

\[
dlseg \; \epsilon (i, i', j, j') \overset{\text{def}}{=} \text{emp} \land i = j \land i' = j' \\
dlseg \; \alpha\cdot a (i, j', k, k') \overset{\text{def}}{=} \exists j. \; i \mapsto a, j, i' \times \text{dlseg} \; \alpha (j, i, k, k').
\]

\[
dlseg \; \alpha\cdot \beta (i, i', k, k') \Leftrightarrow \exists j, j'. \; \text{dlseg} \; \alpha (i, i', j, j') + \text{dlseg} \; \beta (j, j', k, k').
\]

**Bomat List**

\[
\text{listN} \; \sigma \; i:
\]

is defined by

\[
\text{listN} \; \epsilon \; i \overset{\text{def}}{=} \text{emp} \land i = \text{nil} \\
\text{listN} \; (a\cdot \sigma) \; i \overset{\text{def}}{=} a = i \land \exists j. \; i + 1 \mapsto j \times \text{listN} \; \sigma \; j.
\]

**XOR-Linked List Segment**

\[
xlseg \; \alpha (i, i', j, j'):
\]

is defined by

\[
xlseg \; \epsilon (i, i', j, j') \overset{\text{def}}{=} \text{emp} \land i = j \land i' = j' \\
xlseg \; a\cdot \sigma (i, j', k, k') \overset{\text{def}}{=} \exists j. \; i \mapsto a, (j \oplus i') \times xlseg \; \alpha (j, i, k, k').
\]

\[
xlseg \; \alpha\cdot \beta (i, i', k, k') \Leftrightarrow \exists j, j'. \; xlseg \; \alpha (i, i', j, j') + xlseg \; \beta (j, j', k, k').
\]
**Array Allocation**

\[
\langle \text{comm} \rangle ::= \cdots \mid \langle \text{var} \rangle ::= \text{allocate} \ (\text{exp})
\]

\[
x := \text{allocate} \ y
\]

Inference rule:

Noninterfering:

\[
\{ r \} \ v := \text{allocate} \ e \ \{ \bigotimes_{i \vDash v}^{r+e-1} (i \mapsto -) \ast r \},
\]

where \( v \) does not occur free in \( r \) or \( e \).

---

**DAGs**

\[
dag \ a \ (i) \ \text{iff} \ i = a
\]

\[
dag \ (\tau_1 \cdot \tau_2) \ (i) \ \text{iff}
\]

\[
\exists i_1, i_2. \ i \mapsto i_1, i_2 \ast (\text{dag} \ \tau_1 \ (i_1) \land \text{dag} \ \tau_2 \ (i_2)).
\]

Here, since \texttt{emp} is omitted from its definition, \( \text{dag} \ a \ (i) \) is pure, and therefore intuitionistic. By induction, it is easily seen that \( \text{dag} \ \tau \ i \) is intuitionistic for all \( \tau \). In fact, this is vital, since we want \( \text{dag} \ \tau_1 \ (i_1) \land \text{dag} \ \tau_2 \ (i_2) \) to hold for a heap that contains the (possibly overlapping) sub-dags, but not to assert that the sub-dags are identical.

---

**Intuitionistic Separation Logic**

Supports justification rather than truth.

Things that no longer hold include:

- law of excluded middle \((P \lor \neg P)\)
- double negation \((\neg \neg P \equiv P)\)
- Pierce’s law \(((P \Rightarrow Q) \Rightarrow P) \Rightarrow P)\)

Formulae valid in intuitionistic separation logic but not the classical one.

\[
\begin{align*}
x \mapsto 1. y & \Rightarrow \text{emp} \\
x \mapsto 1. y \ast y \mapsto \text{nil} & \Rightarrow x \mapsto 1. _-
\end{align*}
\]
**Intuitionistic Assertion**

An assertion $p$ is intuitionistic iff, for all stores $s$ and heaps $h$ and $h'$:

$$h \subseteq h' \text{ and } s, h \models p \text{ implies } s, h' \models p.$$  

An assertion $p$ is intuitionistic iff

$$p * \text{true} \Rightarrow p.$$  

(The opposite implication always holds.)

---

**Inference for Procedures**

A simple procedure definition has the form

$$h(x_1, \ldots, x_m; y_1, \ldots, y_n) = c,$$

where $y_1, \ldots, y_n$ are the free variables modified by $c$, and $x_1, \ldots, x_m$ are the other free variables of $c$.

When $h(x_1, \ldots, x_m; y_1, \ldots, y_n) = c$,

$$\frac{\{p\} c \{q\}}{\{p\} h(x_1, \ldots, x_m; y_1, \ldots, y_n) \{q\}}.$$  

From the conclusion of this rule, one can reason about other calls by using the rule for free variable substitution (FVS), assuming that the variables modified by $h(x_1, \ldots, x_m; y_1, \ldots, y_n)$ are $y_1, \ldots, y_n$.

---

**Copying Tree**

```plaintext
{tree \tau(i)} \ \text{copytree}(i; j) \ {tree \tau(i) * tree \tau(j)}.
```

```plaintext
\text{copytree}(i; j) =
\begin{align*}
&\text{if isatm}(i) \ \text{then} \ j := i \ \text{else} \\
&\text{newvar} \ i_1, i_2, j_1, j_2 \ \text{in} \\
&(i_1 := [i] \ ; i_2 := [i + 1] ; \\
&\text{copytree}(i_1; j_1) ; \text{copytree}(i_2; j_2) ; \\
&j := \text{cons}(j_1, j_2))
\end{align*}
```

---

**Copying Tree (Proof)**

```plaintext
{\{tree \tau(i)\} \\
\text{if isatm}(i) \ \text{then} \\
\{\text{isatm}(\tau) \land \text{emp} \land i = \tau\} \\
\{\text{isatm}(\tau) \land ((\text{emp} \land i = \tau) \ast (\text{emp} \land i = \tau))\} \\
j := i \\
\{\text{isatm}(\tau) \land ((\text{emp} \land i = \tau) \ast (\text{emp} \land j = \tau))\} \\
\text{else} \\
\{\exists \tau_1, \tau_2. \ \tau = (\tau_1 \land \tau_2) \land \text{tree} ((\tau_1 \land \tau_2)(i))\} \\
\text{newvar} \ i_1, i_2, j_1, j_2 \ \text{in} \\
(i_1 := [i] \ ; i_2 := [i + 1] ; \\
\{\exists \tau_1, \tau_2. \ \tau = (\tau_1 \land \tau_2) \land (i \mapsto i_1, i_2 \ast \\
\text{tree} \tau_1 (i_1) \ast \text{tree} \tau_2 (i_2))\} \\
\text{copytree}(i_1; j_1) ;
```

---
Automated Verification

(i) Given pre/post conditions for each method and loop
(ii) Determine each postcondition is sound for method body.
(iii) Each precondition is satisfied for each call site.

Why Verification?
(i) can handle more complex examples
(ii) can be used to check inference algorithm
(iii) grand challenge of verifiable software

Data Nodes and Notation

data node { int val; node next }
data node2 { int val; node2 prev; node2 next }
data node3 { int val; node3 left; node3 right; node3 parent }

We use p;c(v^*) to denote two things in our system. When c is a data name, p;c(v^*) stands for singleton heap p→[(f v)^*] where f^* are fields of data declaration c. When c is a predicate name, p;c(v^*) stands for the formula c(p,v^*).
**Shape Predicates**

Linked-list with size

\[ \text{ll}(n) \equiv (\text{self} = \text{null} \land n = 0) \lor (\exists i, m, q \cdot \text{self} \cdot \text{node}(i, q) \land \text{ll}(m) \land m = n + 1) \land n \geq 0 \]

Double linked-list (right traversal) with size

\[ \text{dll}(p, n) \equiv (\text{self} = \text{null} \land n = 0) \lor (\text{self} \cdot \text{node}(\_p, q) \land \text{dll}(	ext{self}, n - 1)) \land n \geq 0 \]

Sorted linked-list with size, min, max

\[ \text{sortl}(n, \text{min}, \text{max}) \equiv (\text{self} \cdot \text{node}(\text{min}, \text{null}) \land \text{min} = \text{max} \land n = 1) \]

\[ \lor (\text{self} \cdot \text{node}(\text{min}, q) \land q \cdot \text{sortl}(n - 1, k, \text{max}) \land \text{min} \leq k) \land \text{inv} \land \text{min} \leq \text{max} \land n \geq 1 \]

**Insertion Sort Algorithm**

```plaintext
node insert(node x, node vn) where
  x::sortl(n, sm, lg) \land vn::node(v, \_)
  \rightarrow res::sortl(n+1, min(v, sm), max(v, lg))
  \{
    \text{if (vn.val} \leq \text{x.val) then} \{ \text{vn.next}:=x; \text{vn} \}
    \text{else if (x.next=null) then} \{ x.next:=vn; \text{vn.next}:=null; \text{x} \}
    \text{else} \{ x.next:=insert(x.next, vn); \text{x} \}
  \}

node insertion_sort(node y) where y::ll(n) \land n>0 \rightarrow res::sortl(n, \_)
  \{ if (y.next=null) then y
    \text{else} \{ y.next:=insertion_sort(y.next); insert(y.next, y) \} \}
```

**Prime Notation**

Prime notation is used to capture the latest values of each program variable. This allows a state transition to be expressed since the unprimed form denotes original values.

While \( x < 0 \) where true \( \rightarrow (x > 0 \land x' = x) \lor (x \leq 0 \land x' = 0) \) do \{ \( x := x + 1 \) \}

Here \( x \) and \( x' \) denote the old and new values of variable \( x \) at the entry and exit of the loop, respectively.

**Example**:

\[ \{ x' = x \land y' = y \} \]

\[ x := x + 1 \]

\[ \{ x' = x + 1 \land y' = y \} \]

\[ x := x + y \]

\[ \{ x' = x + 1 + y \land y' = y \} \]

\[ y := 2 \]

\[ \{ x' = x + 1 + y \land y' = 2 \} \]
**Forward Verification**

Given $\Delta_1$, infer $\Delta_2$:

$$\vdash \{\Delta_1\} \vDash \{\Delta_2\}$$

- **[FV-METH]**
  
  $V = \{v_1, \ldots, v_n\}$ \hspace{1em} $W = \text{prime}(V)$ \hspace{1em} $\Delta = \Phi_{pr} \land \text{unchange}(V)$ \hspace{1em} $\vdash \{\Delta\} \vDash \{\Delta_1\} \hspace{1em} (\exists W \cdot \Delta_1) \vdash \Phi_{po} \land \Delta_2$
  
  $\vdash t_0 \text{ mn}(t_1, \ldots, t_n \cdot v_n)$ where $\Phi_{pr} \leftrightarrow \Phi_{po} \vDash \{c\}$

- **[FV-CALL]**
  
  $t \text{ mn}(\{v_i\}_{i=1}^n)$ where $\Phi_{pr} \leftrightarrow \Phi_{po} \{\}
  
  $\Delta \vdash \rho \cdot \Phi_{po} \land \Delta_1$ \hspace{1em} $W = \{v_1, \ldots, v_n\}$ \hspace{1em} $\Delta_2 = (\Delta_1 \cdot W) \Phi_{po}$
  
  $\vdash \{\Delta\} \vDash \{\Delta_2\}$

**Separation Constraint Normalization Rules**

**Target:**

- $\Phi ::= \forall (\exists v^* \cdot \Delta \cdot \pi)^*$
- $\pi ::= \gamma \cdot \phi$
- $\gamma ::= v_1 = v_2 \mid v = \text{null} \mid v_1 \neq v_2 \mid v \neq \text{null} \mid v_1 \land v_2$
- $\Delta ::= \text{emp} \mid v : c(v^*) \mid \Delta_1 \land \Delta_2$

- $(\Delta_1 \lor \Delta_2) \land \pi \sim (\Delta_1 \land \pi) \lor (\Delta_2 \land \pi)$
- $(\Delta_1 \lor \Delta_2) \land \Delta \sim (\Delta_1 \land \Delta) \lor (\Delta_2 \land \Delta)$
- $(\kappa_1 \land \kappa_2) \land (\kappa_2 \land \pi_2) \sim (\kappa_1 \land \kappa_2 \land (\pi_1 \land \pi_2)\Delta_1 \land \Delta)$
- $(\exists x \cdot \Delta) \land \pi \sim \exists y \cdot ([y/x] \Delta \land \pi)$
- $(\exists x \cdot \Delta_1) \land \Delta_2 \sim \exists y \cdot ([y/x] \Delta_1 \land \Delta_2)$

**Separation Constraint Approximation**

XPure$_n(\Phi)$ returns a sound approximation of the form:

$$\text{ex } i^* \cdot \sqrt{(\exists v^* \cdot \pi)^*}$$

**Normalization:**

- $(\text{ex } I \cdot \phi_1) \lor (\text{ex } J \cdot \phi_2) \sim \text{ex } I \cup J \cdot (\phi_1 \lor \phi_2)$
- $(\exists v \cdot (\text{ex } I \cdot \phi)) \sim \text{ex } I \cdot (\exists v \cdot \phi)$
- $(\text{ex } I \cdot \phi_1) \land \text{ex } J \cdot \phi_2) \sim \text{ex } I \cup J \cdot \phi_1 \land \phi_2 \land \forall i \in I, j \in J \cdot i \neq j$
Translating to Pure Form

\[
\begin{align*}
(c(v^*)) \equiv \Phi & \Rightarrow \text{inv } \pi_0 \in P \\
\text{Inv}_0(p;c(v^*)) = [p/\text{self},0/\text{null}]\pi_0 \\
(c(v^*)) \equiv \Phi & \Rightarrow \text{inv } \pi_0 \in P \\
\text{Inv}_n(p;c(v^*)) = [p/\text{self},0/\text{null}]XPurc_{n-1}(\Phi) \\
XPurc_n(\exists v^* \cdot \kappa \land \pi)^* = & \exists v^* \cdot XPurc_n(\kappa) \land (0/\text{null})^* \\
XPurc_n(\text{emp}) = & \exists v \\
\text{IsData}(c) \text{ fresh } i & \Rightarrow XPurc_n(p;c(v^*)) = \exists v^* \cdot \text{Inv}_1(i) \cdot \exists v \\
\text{IsPred}(c) \text{ fresh } i^* & \Rightarrow XPurc_n(p;c(v^*)) = \exists v^* \cdot \text{Inv}_1(i) \cdot \exists v \\
XPurc_n(\text{emp}) = & \exists v \\
XPurc_n(\kappa_1 \land \kappa_2) = & \exists v \cdot \text{Inv}_1(\kappa_1) \land \text{Inv}_1(\kappa_2)
\end{align*}
\]

Deriving Shape Invariant

From each pure invariant, such as \((n \geq 0)\) for \(\ll<n>\)

We use \text{Inv}_1(...) to obtain a more precise invariant:

\[
\begin{align*}
\textbf{ex } i \cdot (\text{self}=0 \land n=0 \lor \text{self}=i \land i>0 \land n>0)
\end{align*}
\]

\[
\begin{align*}
(c(v^*)) \equiv \Phi & \Rightarrow \text{inv } \pi_0 \in P \\
\text{Inv}_0(p;c(v^*)) = [p/\text{self},0/\text{null}]\pi_0 \\
(c(v^*)) \equiv \Phi & \Rightarrow \text{inv } \pi_0 \in P \\
\text{Inv}_n(p;c(v^*)) = [p/\text{self},0/\text{null}]XPurc_{n-1}(\Phi)
\end{align*}
\]

Separation Constraint Entailment

\[
\begin{align*}
\Delta_A \vdash^* \kappa \Delta_C * \Delta_R
\end{align*}
\]

\[
\begin{align*}
\kappa * \Delta_A \vdash \exists V \cdot (\kappa * \Delta_C) * \Delta_R
\end{align*}
\]

The purpose of heap entailment is to check that heap nodes in the antecedent \(\Delta_A\) are sufficiently precise to cover all nodes from the consequent \(\Delta_C\), and to compute a residual heap state \(\Delta_R\). \(\kappa\) is the history of nodes from the antecedent that have been used to match nodes from the consequent, V is the list of existentially quantified variables from the consequent. Note that \(\kappa\) and \(V\) are derived.

The entailment checking procedure is invoked with \(\kappa = \text{emp}\) and \(V = \emptyset\). The en-
Unfolding Predicate in Antecedent

We apply an unfold operation on a predicate in the antecedent that matches with a data node in the consequent. Consider:

\[ x::ll(n) \land n>3 \vdash (\exists r::node(\_\_r) \land r::node(y) \land y \neq null) \set A_R \]

\[ \exists q_1::node(\_\_q_1) \land q_1::ll(n-1) \land n>3 \vdash (\exists r::node(\_\_r) \land r::node(y) \land y \neq null) \set A_R \]

\[ q_1::ll(n-1) \land n>3 \vdash (q_1::node(y) \land y \neq null) \set A_R \]

\[ \exists q_2::node(\_\_q_2) \land q_2::ll(n-2) \land n>3 \vdash q_1::node(y) \land y \neq null \set A_R \]

\[ q_2::ll(n-2) \land n>3 \land q_2=y \vdash y \neq null \set A_R \]

\[
\begin{align*}
\text{[UNFOLDING]} & \quad c(v^*) \equiv \Phi \in P \\
\text{unfold}(p::c(v^*)) & \quad =_{df} [p/self] \Phi
\end{align*}
\]

Folding a Predicate in Consequent

Folding is recursively applied until \( x::ll <n> \) matches with the two data nodes in the antecedent, resulting in:

\[ y::node(3, \_\_null) \land n=2 \vdash n>1 \set A_R \]

Effect of folding is not the same as unfolding a predicate in consequent as values of derived variable may be lost!

Soundness of Entailment

Theorem 6.1 (Soundness) If entailment check \( \Delta_1 \vdash \Delta_2 \set A \) succeeds, we have: for all \( s, h \), if \( s, h \models \Delta_1 \) then \( s, h \models \Delta_2 \set A \).

Theorem 6.2 (Termination) The entailment check \( \Delta_1 \vdash \Delta_2 \set A \) always terminates.

Proof sketch: A well-founded measure exists for heap entailment. Matching and unfolding decrease nodes from the consequent. Fold operation has bounded recursive depth as each recursive fold operation always decreases the antecedent since shape predicate has the well-founded property. The size of antecedent is bounded despite unfolding since each unfold is always followed by a decrease of a data node from the consequent. At the end of a fold, a node from the consequent is also removed. A detailed proof is given in the technical report [15].