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 Instuitionistic 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} 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_0+1}

{Q[x+1/x]} x:=x+1 {Q}

{P} x:=x+1 {$\exists x_1$. P[x_1/x] $\wedge$ x=x_1+1}

Problem

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

Consider an in-place list reversal algorithm

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

[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:

$$\exists \alpha, \beta.\ \mathsf{list}\ \alpha\ \mathsf{i} \wedge \mathsf{list}\ \beta\ \mathsf{j} \wedge \alpha_0^{\dagger} = \alpha^{\dagger} \cdot \beta,$$

where

$$\mathsf{list}\ \epsilon\ \mathsf{i} \overset{\text{def}}{=} \mathsf{i} = \mathbf{nil} \qquad \mathsf{list}(\mathsf{a} \cdot \alpha)\ \mathsf{i} \overset{\text{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \hookrightarrow \mathsf{a}, \mathsf{j} \wedge \mathsf{list}\ \alpha\ \mathsf{j}$$

heap predicate relates a list of elements and a pointer

Loop Invariant

An adequate invariant:

$$(\exists \alpha, \beta.\ \mathsf{list}\ \alpha\ \mathsf{i} \wedge \mathsf{list}\ \beta\ \mathsf{j} \wedge \alpha_0^{\dagger} = \alpha^{\dagger} \cdot \beta)$$
$$\wedge\ (\forall \mathsf{k}.\ \mathbf{reachable}(\mathsf{i}, \mathsf{k}) \wedge \mathbf{reachable}(\mathsf{j}, \mathsf{k}) \Rightarrow \mathsf{k} = \mathbf{nil}),$$

where

$$\mathbf{reachable}(\mathsf{i}, \mathsf{j}) \overset{\text{def}}{=} \exists n \geq 0.\ \mathbf{reachable}_n(\mathsf{i}, \mathsf{j})$$
$$\mathbf{reachable}_0(\mathsf{i}, \mathsf{j}) \overset{\text{def}}{=} \mathsf{i} = \mathsf{j}$$
$$\mathbf{reachable}_{n+1}(\mathsf{i}, \mathsf{j}) \overset{\text{def}}{=} \exists \mathsf{a}, \mathsf{k}.\ \mathsf{i} \hookrightarrow \mathsf{a}, \mathsf{k} \wedge \mathbf{reachable}_n(\mathsf{k}, \mathsf{j}).$$

in separation logic :

$$(\exists \alpha, \beta.\ \mathsf{list}\ \alpha\ \mathsf{i}\ *\ \mathsf{list}\ \beta\ \mathsf{j}) \wedge \alpha_0^{\dagger} = \alpha^{\dagger} \cdot \beta$$

Basics of Separation Logic

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

Simple Language with Heap Store

The simple imperative language:

$:=$ **skip** ; **if** $-$ **then** $-$ **else** $-$ **while** $-$ **do** $-$

plus:

		Store : x: 3, y: 4 Heap : empty
Allocation	$\mathsf{x} := \mathbf{cons}(1, 2)$;	
		Store : x: 37, y: 4 Heap : 37: 1, 38: 2
Lookup	$\mathsf{y} := [\mathsf{x}]$;	
		Store : x: 37, y: 1 Heap : 37: 1, 38: 2
Mutation	$[\mathsf{x} + 1] := 3$;	
		Store : x: 37, y: 1 Heap : 37: 1, 38: 3
Deallocation	$\mathbf{dispose}(\mathsf{x} + 1)$	
		Store : x: 37, y: 1 Heap : 37: 1

Memory Faults

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

		Store : x: 3, y: 4 Heap : empty
Allocation	$\mathsf{x} := \mathbf{cons}(1, 2)$;	
		Store : x: 37, y: 4 Heap : 37: 1, 38: 2
Lookup	$\mathsf{y} := [\mathsf{x}]$;	
		Store : x: 37, y: 1 Heap : 37: 1, 38: 2
Mutation	$[\mathsf{x} + 2] := 3$;	
		abort

Assertion Language

Standard predicate calculus:

$$\wedge \quad \vee \quad \neg \quad \Rightarrow \quad \forall \quad \exists$$

plus:

- **emp**
  The heap is empty.
- $e \mapsto e'$
  The heap contains one cell, at address e with contents e'.
- $p_1 * p_2$
  The heap can be split into two disjoint parts such that p_1 holds for one part and p_2 holds for the other.
- $p_1 -\!\!* \; p_2$
  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

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:

$$s, h \models b \text{ iff } [\![b]\!]_{\text{boolexp}} s = \mathbf{true},$$

$$s, h \models \neg p \text{ iff } s, h \models p \text{ is false},$$

$$s, h \models p_0 \wedge p_1 \text{ iff } s, h \models p_0 \text{ and } s, h \models p_1$$

(and similarly for $\vee$, $\Rightarrow$, $\Leftrightarrow$),

Semantic Model

$$s, h \models \forall v.\, p \text{ iff } \forall x \in \mathbf{Z}.\, [\, s \mid v\!: x \,], h \models p,$$

$$s, h \models \exists v.\, p \text{ iff } \exists x \in \mathbf{Z}.\, [\, s \mid v\!: x \,], h \models p,$$

$$s, h \models \mathbf{emp} \text{ iff } \operatorname{dom} h = \{\},$$

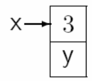
$$s, h \models e \mapsto e' \text{ iff } \operatorname{dom} h = \{[\![e]\!]_{\text{exp}} s\} \text{ and } h([\![e]\!]_{\text{exp}} s) = [\![e']\!]_{\text{exp}} s,$$

$$s, h \models p_0 * p_1 \text{ iff } \exists h_0, h_1.\, h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h \text{ and } s, h_0 \models p_0 \text{ and } s, h_1 \models p_1,$$

$$s, h \models p_0 -\!\!* \; p_1 \text{ iff } \forall h'.\, (h' \perp h \text{ and } s, h' \models p_0) \text{ implies } s, h \cdot h' \models p_1.$$

Separation Conjunction - Examples

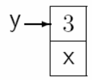
1. $\mathsf{x} \mapsto 3, \mathsf{y}$



Store : $\mathsf{x}\colon \alpha,\ \mathsf{y}\colon \beta$

Heap : $\alpha\colon 3,\ \alpha{+}1\colon \beta$

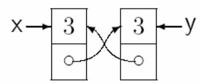
2. $\mathsf{y} \mapsto 3, \mathsf{x}$



Store : $\mathsf{x}\colon \alpha,\ \mathsf{y}\colon \beta$

Heap : $\beta\colon 3,\ \beta{+}1\colon \alpha$

3. $\mathsf{x} \mapsto 3, \mathsf{y} \ * \ \mathsf{y} \mapsto 3, \mathsf{x}$



Store : $\mathsf{x}\colon \alpha,\ \mathsf{y}\colon \beta$

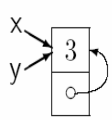
Heap : $\alpha\colon 3,\ \alpha{+}1\colon \beta,\ \beta\colon 3,\ \beta{+}1\colon \alpha$

where $\alpha,\ \alpha+1,\ \beta,\ \beta+1$ are distinct

Conjunction - Examples

Conjunction describes the same heap space.

4. $\mathsf{x} \mapsto 3, \mathsf{y} \wedge \mathsf{y} \mapsto 3, \mathsf{x}$



Store : $\mathsf{x}\colon \alpha,\ \mathsf{y}\colon \alpha$

Heap : $\alpha\colon 3,\ \alpha{+}1\colon \alpha$

5. $\mathsf{x} \hookrightarrow 3, \mathsf{y} \wedge \mathsf{y} \hookrightarrow 3, \mathsf{x}$

Store : $\mathsf{x}\colon \alpha,\ \mathsf{y}\colon \beta$

Heap : $\alpha\colon 3,\ \alpha{+}1\colon \beta,\ \beta\colon 3,\ \beta{+}1\colon \alpha,\ \ldots$

As in (3) or (4), possibly with additional cells

Separation Implication - Examples

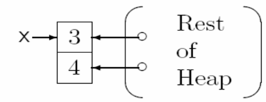
$p_1 \mathrel{-\!\!*} p_2$

If the current heap is extended with a disjoint part in which p_1 holds, then p_2 holds for the extended heap.

Suppose p holds for

Store : $\mathsf{x}\colon \alpha,\ \ldots$

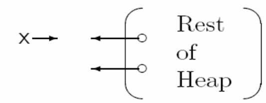
Heap : $\alpha\colon 3, \alpha + 1\colon 4$, rest of heap



Then $(\mathsf{x} \mapsto 3, 4) \mathrel{-\!\!*} p$ holds for

Store : $\mathsf{x}\colon \alpha,\ \ldots$

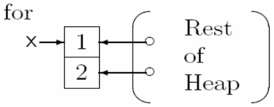
Heap : rest of heap



and $\mathsf{x} \mapsto 1, 2 \ * \ ((\mathsf{x} \mapsto 3, 4) \mathrel{-\!\!*} p)$ holds for

Store : $\mathsf{x}\colon \alpha,\ \ldots$

Heap : $\alpha\colon 1, \alpha + 1\colon 2$, rest of heap



Inference Rules

Reasoning with normalization, weakening and strengthening.

$$p_0 \ * \ p_1 \Leftrightarrow p_1 \ * \ p_0$$

$$(p_0 \ * \ p_1) \ * \ p_2 \Leftrightarrow p_0 \ * \ (p_1 \ * \ p_2)$$

$$p \ * \ \mathbf{emp} \Leftrightarrow p$$

$$(p_0 \vee p_1) \ * \ q \Leftrightarrow (p_0 \ * \ q) \vee (p_1 \ * \ q)$$

$$(p_0 \wedge p_1) \ * \ q \Rightarrow (p_0 \ * \ q) \wedge (p_1 \ * \ q)$$

$$(\exists x.\ p_0) \ * \ p_1 \Leftrightarrow \exists x.\ (p_0 \ * \ p_1) \quad \text{when } x \text{ not free in } p_1$$

$$(\forall x.\ p_0) \ * \ p_1 \Rightarrow \forall x.\ (p_0 \ * \ p_1) \quad \text{when } x \text{ not free in } p_1$$

$$\frac{p_0 \Rightarrow p_1 \qquad q_0 \Rightarrow q_1}{p_0 \ * \ q_0 \Rightarrow p_1 \ * \ q_1} \ \text{(monotonicity)}$$

$$\frac{p_0 \ * \ p_1 \Rightarrow p_2}{p_0 \Rightarrow (p_1 \mathrel{-\!\!*} p_2)} \ \text{(currying)} \qquad \frac{p_0 \Rightarrow (p_1 \mathrel{-\!\!*} p_2)}{p_0 \ * \ p_1 \Rightarrow p_2.} \ \text{(decurrying)}$$

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**, $\mapsto$, or $\hookrightarrow$.

Axiom schematic guided by pure formulae

$p_0 \wedge p_1 \Rightarrow p_0 * p_1$	when p_0 or p_1 is pure
$p_0 * p_1 \Rightarrow p_0 \wedge p_1$	when p_0 and p_1 are pure
$(p \wedge q) * r \Leftrightarrow (p * r) \wedge q$	when q is pure
$(p_0 -\!\!* p_1) \Rightarrow (p_0 \Rightarrow p_1)$	when p_0 is pure
$(p_0 \Rightarrow p_1) \Rightarrow (p_0 -\!\!* p_1)$	when p_0 and p_1 are pure.

Two Unsound Axiom Schemata

$p \not\Rightarrow p * p$	(Contraction)
	e.g. $p : \mathsf{x} \mapsto 1$
$p * q \not\Rightarrow p$	(Weakening)
	e.g. $p : \mathsf{x} \mapsto 1$
	$q : \mathsf{y} \mapsto 2$

Structural logic without contraction and weakening.

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.

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

$$\{\mathsf{x}-\mathsf{y}>3\}\ \mathsf{x}:=\mathsf{x}-\mathsf{y}\ \{\mathsf{x}>3\}$$

$$\{\mathsf{x}+\mathsf{y}\geq 17\}\ \mathsf{x}:=\mathsf{x}+10\ \{\mathsf{x}+\mathsf{y}\geq 27\}$$

$$\{\mathbf{emp}\}\ \mathsf{x}:=\mathbf{cons}(1,2)\ \{\mathsf{x}\mapsto 1,2\}$$

$$\{\mathsf{x}\mapsto 1,2\}\ \mathsf{y}:=[\mathsf{x}]\ \{\mathsf{x}\mapsto 1,2\wedge \mathsf{y}=1\}$$

$$\{\mathsf{x}\mapsto 1,2\wedge \mathsf{y}=1\}\ [\mathsf{x}+1]:=3\ \{\mathsf{x}\mapsto 1,3\wedge \mathsf{y}=1\}$$

$$\{\mathsf{x}\mapsto 1,3\wedge \mathsf{y}=1\}\ \mathbf{dispose}\ \mathsf{x}\ \{\mathsf{x}+1\mapsto 3\wedge \mathsf{y}=1\}$$

$$\{\mathsf{x}\leq 10\}\ \mathbf{while}\ \mathsf{x}\neq 10\ \mathbf{do}\ \mathsf{x}:=\mathsf{x}+1\ \{\mathsf{x}=10\}$$

$$\{\mathbf{true}\}\ \mathbf{while}\ \mathsf{x}\neq 10\ \mathbf{do}\ \mathsf{x}:=\mathsf{x}+1\ \{\mathsf{x}=10\}$$

$$\{\mathsf{x}>10\}\ \mathbf{while}\ \mathsf{x}\neq 10\ \mathbf{do}\ \mathsf{x}:=\mathsf{x}+1\ \{\mathbf{false}\}$$

Hoare Inference Rules

Assignment (AS)

$$\frac{}{\{p/v\to e\}\ v:=e\ \{p\}} \qquad \frac{}{[\,p/v\to e\,]\ v:=e\ [\,p\,]}$$

Sequential Composition (SQ)

$$\frac{\{p\}\ c_1\ \{q\} \qquad \{q\}\ c_2\ \{r\}}{\{p\}\ c_1\,;c_2\ \{r\}} \qquad \frac{[\,p\,]\ c_1\ [\,q\,] \qquad [\,q\,]\ c_2\ [\,r\,]}{[\,p\,]\ c_1\,;c_2\ [\,r\,]}$$

Hoare Inference Rules

Structural rules are applicable to any commands.

Strengthening Precedent (SP)

$$\frac{p\Rightarrow q \qquad \{q\}\ c\ \{r\}}{\{p\}\ c\ \{r\}}$$

Weakening Consequent (WC)

$$\frac{\{p\}\ c\ \{q\} \qquad q\Rightarrow r}{\{p\}\ c\ \{r\}}$$

Partial Correctness of While Loop

$$\frac{\{i\wedge b\}\ c\ \{i\}}{\{i\}\ \mathbf{while}\ b\ \mathbf{do}\ c\ \{i\wedge\neg b\}}$$

Here i is the *invariant*.

An Instance

$$\frac{\{\mathsf{y}=2^{\mathsf{k}}\wedge\mathsf{k}\leq\mathsf{n}\wedge\mathsf{k}\neq\mathsf{n}\}\ \mathsf{k}:=\mathsf{k}+1\,;\mathsf{y}:=2\times\mathsf{y}\ \{\mathsf{y}=2^{\mathsf{k}}\wedge\mathsf{k}\leq\mathsf{n}\}}{\begin{array}{l}\{\mathsf{y}=2^{\mathsf{k}}\wedge\mathsf{k}\leq\mathsf{n}\}\\ \mathbf{while}\ \mathsf{k}\neq\mathsf{n}\ \mathbf{do}\ (\mathsf{k}:=\mathsf{k}+1\,;\mathsf{y}:=2\times\mathsf{y})\\ \{\mathsf{y}=2^{\mathsf{k}}\wedge\mathsf{k}\leq\mathsf{n}\wedge\neg\,\mathsf{k}\neq\mathsf{n}\}\end{array}}$$

Total Correctness of While Loop

$$\frac{[\, i \wedge b \wedge e = v_0 \,]\; c\; [\, i \wedge e < v_0 \,] \qquad (i \wedge b) \Rightarrow e \geq 0}{[\, i \,]\; \textbf{while}\; \underline{[\textbf{vrnt}\!: e]}\; b\; \textbf{do}\; c\; [\, i \wedge \neg b \,]}$$

when v_0 does not occur free in i, b, c, or e.

$$[\, \mathsf{x} \leq 10 \,]\; \textbf{while}\; \mathsf{x} \neq 10\; \textbf{do}\; \mathsf{x} := \mathsf{x} + 1\; [\, \mathsf{x} = 10 \,]$$

Hoare Inference Rules

Conditional (CD)

$$\frac{\{p \wedge b\}\; c_1\; \{q\} \qquad \{p \wedge \neg\, b\}\; c_2\; \{q\}}{\{p\}\; \textbf{if}\; b\; \textbf{then}\; c_1\; \textbf{else}\; c_2\; \{q\}}$$

skip (SK)

$$\frac{}{\{p\}\; \textbf{skip}\; \{p\}}$$

Hoare Inference Rules

Variable Declaration (DC)

$$\frac{\{p\}\; c\; \{q\}}{\{p\}\; \textbf{newvar}\; v\; \textbf{in}\; c\; \{q\}}$$

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 :

Sequential Composition (SQAN)

$$\frac{\{p\}\; c_1\; \{q\} \qquad \{q\}\; c_2\; \{r\}}{\{p\}\; c_1\, ;\, \underline{\{q\}}\; c_2\; \{r\}}$$

Strengthening Precedent (SPAN)

$$\frac{p \Rightarrow q \qquad \{q\}\; c\; \{r\}}{\{p\}\; \underline{\{q\}}\; c\; \{r\}}$$

Minimal Annotated Specifications

Should attempt to minimise annotations where possible.

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

$\{\mathsf{n} \geq 0\}$
$\mathsf{k} := 0\,;\, \mathsf{y} := 1\,;$
$\{\mathsf{y} = 2^{\mathsf{k}} \wedge \mathsf{k} \leq \mathsf{n}\}$
$\textbf{while } \mathsf{k} \neq \mathsf{n} \textbf{ do } (\mathsf{k} := \mathsf{k} + 1\,;\, \mathsf{y} := 2 \times \mathsf{y})$
$\{\mathsf{y} = 2^{\mathsf{n}}\}$

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

Structural Inference Rules

Renaming (RN)

$$\frac{\{p\}\ c\ \{q\}}{\{p'\}\ c'\ \{q'\},}$$

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

Substitution (SUB)

$$\frac{\{p\}\ c\ \{q\}}{(\{p\}\ c\ \{q\})/v_1 \to e_1,\ \ldots\ ,\ v_n \to e_n,}$$

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

Conjunction (CONJ)

$$\frac{\{p_1\}\ c\ \{q_1\} \qquad \{p_2\}\ c\ \{q_2\}}{\{p_1 \wedge p_2\}\ c\ \{q_1 \wedge q_2\}}$$

Disjunction (DISJ)

$$\frac{\{p_1\}\ c\ \{q_1\} \qquad \{p_2\}\ c\ \{q_2\}}{\{p_1 \vee p_2\}\ c\ \{q_1 \vee q_2\}}$$

Structural Inference Rules

Universal Quantification (UQ)

$$\frac{\{p\}\ c\ \{q\}}{\{\forall v.\ p\}\ c\ \{\forall v.\ q\},}$$

where v is not free in c.

Existential Quantification (EQ)

$$\frac{\{p\}\ c\ \{q\}}{\{\exists v.\ p\}\ c\ \{\exists v.\ q\},}$$

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

$$\frac{\{\mathsf{x} \mapsto -\}\ [\mathsf{x}] := 4\ \{\mathsf{x} \mapsto 4\}}{\{\mathsf{x} \mapsto - \land \mathsf{y} \mapsto 3\}\ [\mathsf{x}] := 4\ \{\mathsf{x} \mapsto 4 \land \mathsf{y} \mapsto 3\}}$$

fails when $\mathsf{x} = \mathsf{y}$.

Frame Rule of Separation Logic

- Frame Rule (O'Hearn) (FR)

$$\frac{\{p\}\ c\ \{q\}}{\{p\ *\ r\}\ c\ \{q\ *\ r\}},$$

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

This facilitates local reasoning and specification

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,

$$\frac{\{\textbf{list}\ \alpha\ \mathsf{i}\}\ \text{“Reverse List”}\ \{\textbf{list}\ \alpha^{\dagger}\ \mathsf{j}\}}{\{\textbf{list}\ \alpha\ \mathsf{i}\ *\ \textbf{list}\ \gamma\ \mathsf{k}\}\ \text{“Reverse List”}\ \{\textbf{list}\ \alpha^{\dagger}\ \mathsf{j}\ *\ \textbf{list}\ \gamma\ \mathsf{k}\}}.$$

Inference Rules for Mutation

The local form (MUL):

$$\overline{\{e \mapsto -\}\ [e] := e'\ \{e \mapsto e'\}}.$$

The global form (MUG):

$$\overline{\{(e \mapsto -)\ *\ r\}\ [e] := e'\ \{(e \mapsto e')\ *\ r\}}.$$

The backward-reasoning form (MUBR):

$$\overline{\{(e \mapsto -)\ *\ ((e \mapsto e') -\!\!*\ p)\}\ [e] := e'\ \{p\}}.$$

Inference Rules for Deallocation

The local form (DISL):

$$\frac{}{\{e \mapsto -\}\ \mathbf{dispose}\ e\ \{\mathbf{emp}\}.}$$

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

$$\frac{}{\{(e \mapsto -)\ *\ r\}\ \mathbf{dispose}\ e\ \{r\}.}$$

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

Inference Rules for Noninterfering Allocation

The local form (CONSNIL):

$$\frac{}{\{\mathbf{emp}\}\ v := \mathbf{cons}(e_0, \ldots, e_{n-1})\ \{v \mapsto e_0, \ldots, e_{n-1}\},}$$

where $v \notin \mathrm{FV}(e_0, \ldots, e_{n-1})$.

The global form (CONSNIG):

$$\frac{}{\{r\}\ v := \mathbf{cons}(e_0, \ldots, e_{n-1})\ \{(v \mapsto e_0, \ldots, e_{n-1})\ *\ r\},}$$

where $v \notin \mathrm{FV}(e_0, \ldots, e_{n-1}, r)$.

Inference Rules for Lookup

The local form (LKL):

$$\frac{}{\{v = v' \wedge (e \mapsto v'')\}\ v := [e]\ \{v = v'' \wedge (e' \mapsto v'')\},}$$

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

The global form (LKG):

$$\frac{}{\begin{array}{l}\{\exists v''.\ (e \mapsto v'')\ *\ (r/v' \rightarrow v)\}\ v := [e] \\ \qquad\qquad\qquad \{\exists v'.\ (e' \mapsto v)\ *\ (r/v'' \rightarrow v)\},\end{array}}$$

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

The backward-reasoning form (LKBR):

$$\frac{}{\{\exists v''.\ (e \hookrightarrow v'') \wedge p''\}\ v := [e]\ \{p\},}$$

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

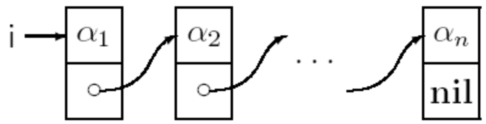
Notation for Sequences

When α and β are sequences, we write

- ϵ 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 α followed by β.
- $\alpha^\dagger$ for the reflection of α.
- $\#\alpha$ for the length of α.
- α_i for the ith component of α.

Singly Linked List

list α i:



is defined by

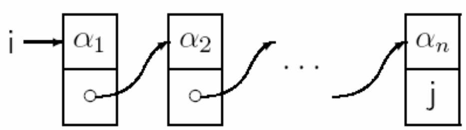
$$\text{list}\ \epsilon\ \text{i} \stackrel{\text{def}}{=} \textbf{emp} \wedge \text{i} = \textbf{nil}$$

$$\text{list}\ (\text{a}\cdot\alpha)\ \text{i} \stackrel{\text{def}}{=} \exists \text{j}.\ \text{i} \mapsto \text{a}, \text{j} \ast \text{list}\ \alpha\ \text{j}.$$

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

Singly Linked List Segment

lseg α (i, j):



is defined by induction on the length of the sequence α (i.e., by structural induction on α):

$$\text{lseg}\ \epsilon\ (\text{i}, \text{j}) \stackrel{\text{def}}{=} \textbf{emp} \wedge \text{i} = \text{j}$$

$$\text{lseg}\ \text{a}\cdot\alpha\ (\text{i}, \text{k}) \stackrel{\text{def}}{=} \exists \text{j}.\ \text{i} \mapsto \text{a}, \text{j} \ast \text{lseg}\ \alpha\ (\text{j}, \text{k}).$$

Singly Linked List Segment

Properties

$$\text{lseg}\ \text{a}\ (\text{i}, \text{j}) \Leftrightarrow \text{i} \mapsto \text{a}, \text{j}$$

$$\text{lseg}\ \alpha\cdot\beta\ (\text{i}, \text{k}) \Leftrightarrow \exists \text{j}.\ \text{lseg}\ \alpha\ (\text{i}, \text{j}) \ast \text{lseg}\ \beta\ (\text{j}, \text{k})$$

$$\text{lseg}\ \alpha\cdot\text{b}\ (\text{i}, \text{k}) \Leftrightarrow \exists \text{j}.\ \text{lseg}\ \alpha\ (\text{i}, \text{j}) \ast \text{j} \mapsto \text{b}, \text{k}$$

$$\text{list}\ \alpha\ \text{i} \Leftrightarrow \text{lseg}\ \alpha\ (\text{i}, \textbf{nil}).$$

Emptyness Conditions

$$\text{lseg}\ \alpha\ (\text{i}, \text{j}) \Rightarrow (\text{i} = \textbf{nil} \Rightarrow (\alpha = \epsilon \wedge \text{j} = \textbf{nil}))$$

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

Non-Touching Linked List Segment

We can define nontouching list segments in terms of lseg:

$$\text{ntlseg}\ \alpha\ (\text{i}, \text{j}) \stackrel{\text{def}}{=} \text{lseg}\ \alpha\ (\text{i}, \text{j}) \wedge \neg\, \text{j} \hookrightarrow -,$$

or we can define them inductively:

$$\text{ntlseg}\ \epsilon\ (\text{i}, \text{j}) \stackrel{\text{def}}{=} \textbf{emp} \wedge \text{i} = \text{j}$$

$$\text{ntlseg}\ \text{a}\cdot\alpha\ (\text{i}, \text{k}) \stackrel{\text{def}}{=} \text{i} \neq \text{k} \wedge \text{i} \neq \text{k} + 1 \wedge (\exists \text{j}.\ \text{i} \mapsto \text{a}, \text{j} \ast \text{ntlseg}\ \alpha\ (\text{j}, \text{k})).$$

Easier test for emptiness

$$\text{ntlseg}\ \alpha\ (\text{i}, \text{j}) \Rightarrow (\alpha = \epsilon \Leftrightarrow \text{i} = \text{j})$$

Braced List Segment

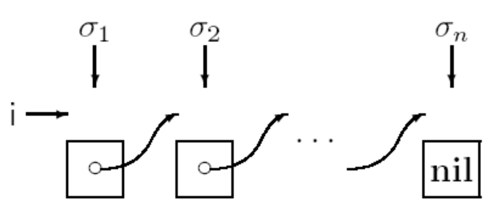
A *braced list segment* is a list segment with an interior pointer j to its last element; in the special case where the list segment is empty, j is **nil**. Formally,

$$\mathbf{brlseg}\ \epsilon\ (\mathsf{i}, \mathsf{j}, \mathsf{k}) \stackrel{\text{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathsf{k} \wedge \mathsf{j} = \mathbf{nil}$$

$$\mathbf{brlseg}\ \alpha{\cdot}a\ (\mathsf{i}, \mathsf{j}, \mathsf{k}) \stackrel{\text{def}}{=} \mathbf{lseg}\ \alpha\ (\mathsf{i}, \mathsf{j})\ *\ \mathsf{j} \mapsto \mathsf{a}, \mathsf{k}.$$

Bornat List

listN σ i:



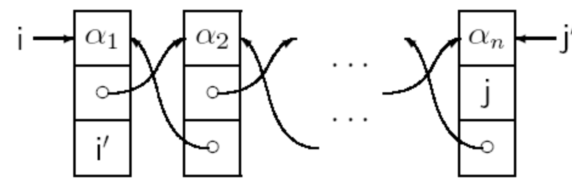
is defined by

$$\mathsf{listN}\ \epsilon\ \mathsf{i} \stackrel{\text{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathbf{nil}$$

$$\mathsf{listN}\ (\mathsf{a}{\cdot}\sigma)\ \mathsf{i} \stackrel{\text{def}}{=} \mathsf{a} = \mathsf{i} \wedge \exists \mathsf{j}.\ \mathsf{i} + 1 \mapsto \mathsf{j}\ *\ \mathsf{listN}\ \sigma\ \mathsf{j}.$$

Doubly Linked List

dlseg α (i, i′, j, j′):



is defined by

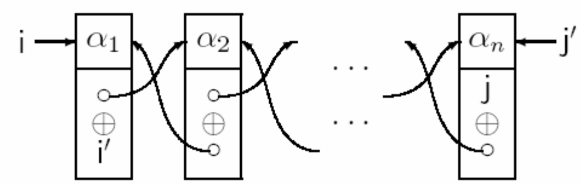
$$\mathsf{dlseg}\ \epsilon\ (\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') \stackrel{\text{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathsf{j} \wedge \mathsf{i}' = \mathsf{j}'$$

$$\mathsf{dlseg}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{i}, \mathsf{i}', \mathsf{k}, \mathsf{k}') \stackrel{\text{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a}, \mathsf{j}, \mathsf{i}'\ *\ \mathsf{dlseg}\ \alpha\ (\mathsf{j}, \mathsf{i}, \mathsf{k}, \mathsf{k}'),$$

$$\mathsf{dlseg}\ \alpha{\cdot}\beta\ (\mathsf{i}, \mathsf{i}', \mathsf{k}, \mathsf{k}') \Leftrightarrow \exists \mathsf{j}, \mathsf{j}'.\ \mathsf{dlseg}\ \alpha\ (\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}')\ *\ \mathsf{dlseg}\ \beta\ (\mathsf{j}, \mathsf{j}', \mathsf{k}, \mathsf{k}')$$

XOR-Linked List Segment

xlseg α (i, i′, j, j′):



is defined by

$$\mathsf{xlseg}\ \epsilon\ (\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}') \stackrel{\text{def}}{=} \mathbf{emp} \wedge \mathsf{i} = \mathsf{j} \wedge \mathsf{i}' = \mathsf{j}'$$

$$\mathsf{xlseg}\ \mathsf{a}{\cdot}\alpha\ (\mathsf{i}, \mathsf{i}', \mathsf{k}, \mathsf{k}') \stackrel{\text{def}}{=} \exists \mathsf{j}.\ \mathsf{i} \mapsto \mathsf{a}, (\mathsf{j} \oplus \mathsf{i}')\ *\ \mathsf{xlseg}\ \alpha\ (\mathsf{j}, \mathsf{i}, \mathsf{k}, \mathsf{k}').$$

$$\mathsf{xlseg}\ \alpha{\cdot}\beta\ (\mathsf{i}, \mathsf{i}', \mathsf{k}, \mathsf{k}') \Leftrightarrow \exists \mathsf{j}, \mathsf{j}'.\ \mathsf{xlseg}\ \alpha\ (\mathsf{i}, \mathsf{i}', \mathsf{j}, \mathsf{j}')\ *\ \mathsf{xlseg}\ \beta\ (\mathsf{j}, \mathsf{j}', \mathsf{k}, \mathsf{k}')$$

Array Allocation

$$\langle \text{comm} \rangle ::= \cdots \mid \langle \text{var} \rangle := \textbf{allocate}\ \langle \text{exp} \rangle$$

	Store :	x: 3, y: 4
	Heap :	empty
x := **allocate** y		
	Store :	x: 37, y: 4
	Heap :	37: −, 38: −, 39: −, 40: −

Inference rule :

Noninterfering:

$$\overline{\{r\}\ v := \textbf{allocate}\ e\ \{(\bigodot_{i=v}^{v+e-1} i \mapsto -) * r\},}$$

where v does not occur free in r or e.

Trees

For $\tau \in$ S-exps, we define the assertion

$$\mathsf{tree}\ \tau\ (i)$$

by structural induction:

$$\mathsf{tree}\ a\ (i) \text{ iff } \mathbf{emp} \wedge i = a$$

$$\mathsf{tree}\ (\tau_1 \cdot \tau_2)\ (i) \text{ iff}$$
$$\exists i_1, i_2.\ i \mapsto i_1, i_2\ *\ \mathsf{tree}\ \tau_1\ (i_1)\ *\ \mathsf{tree}\ \tau_2\ (i_2).$$

$\tau \in$ S-exps iff

$$\tau \in \text{Atoms}$$
$$\text{or } \tau = (\tau_1 \cdot \tau_2) \text{ where } \tau_1, \tau_2 \in \text{S-exps}.$$

DAGs

$$\mathsf{dag}\ a\ (i) \text{ iff } i = a$$

$$\mathsf{dag}\ (\tau_1 \cdot \tau_2)\ (i) \text{ iff}$$
$$\exists i_1, i_2.\ i \mapsto i_1, i_2\ *\ (\mathsf{dag}\ \tau_1\ (i_1) \wedge \mathsf{dag}\ \tau_2\ (i_2)).$$

Here, since **emp** is omitted from its definition, $\mathsf{dag}\ a\ (i)$ is pure, and therefore intuitionistic. By induction, it is easily seen that $\mathsf{dag}\ \tau\ i$ is intuitionistic for all τ. In fact, this is vital, since we want $\mathsf{dag}\ \tau_1\ (i_1) \wedge \mathsf{dag}\ \tau_2\ (i_2)$ 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	$(P \vee \neg P)$
double negation	$(\neg \neg P = P)$
Pierce's law	$(((P \Rightarrow Q) \Rightarrow P) \Rightarrow P)$

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

$$x \mapsto 1,y \quad \Rightarrow \text{emp}$$
$$x \mapsto 1,y * y \mapsto \text{nil} \quad \Rightarrow x \mapsto 1,\_$$

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

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

(The opposite implication always holds.)

Inference for Procedures

A simple procedure definition has the form

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

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

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

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

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 $h(x_1, \cdots, x_m; y_1, \cdots, y_n)$ are $y_1, \cdots, y_n$.

Copying Tree

$$\{\mathsf{tree}\ \tau(\mathsf{i})\}\ \mathsf{copytree}(\mathsf{i};\mathsf{j})\ \{\mathsf{tree}\ \tau(\mathsf{i})\ *\ \mathsf{tree}\ \tau(\mathsf{j})\}.$$

$\mathsf{copytree}(\mathsf{i};\mathsf{j}) =$

$\quad \mathbf{if}\ \mathsf{isatom}(\mathsf{i})\ \mathbf{then}\ \mathsf{j} := \mathsf{i}\ \mathbf{else}$

$\quad\quad \mathbf{newvar}\ \mathsf{i}_1, \mathsf{i}_2, \mathsf{j}_1, \mathsf{j}_2\ \mathbf{in}$

$\quad\quad\quad \big(\mathsf{i}_1 := [\mathsf{i}]\ ; \mathsf{i}_2 := [\mathsf{i}+1]\ ;$

$\quad\quad\quad \mathbf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\ ; \mathbf{copytree}(\mathsf{i}_2;\mathsf{j}_2)\ ;$

$\quad\quad\quad \mathsf{j} := \mathbf{cons}(\mathsf{j}_1, \mathsf{j}_2)\big)$

Copying Tree (Proof)

$\{\mathsf{tree}\ \tau(\mathsf{i})\}$

$\mathbf{if}\ \mathsf{isatom}(\mathsf{i})\ \mathbf{then}$

$\quad \{\mathsf{isatom}(\tau) \wedge \mathbf{emp} \wedge \mathsf{i} = \tau\}$

$\quad \{\mathsf{isatom}(\tau) \wedge ((\mathbf{emp} \wedge \mathsf{i} = \tau)\ *\ (\mathbf{emp} \wedge \mathsf{i} = \tau))\}$

$\quad \mathsf{j} := \mathsf{i}$

$\quad \{\mathsf{isatom}(\tau) \wedge ((\mathbf{emp} \wedge \mathsf{i} = \tau)\ *\ (\mathbf{emp} \wedge \mathsf{j} = \tau))\}$

$\mathbf{else}$

$\quad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge \mathsf{tree}\ (\tau_1 \cdot \tau_2)(\mathsf{i})\}$

$\quad \mathbf{newvar}\ \mathsf{i}_1, \mathsf{i}_2, \mathsf{j}_1, \mathsf{j}_2\ \mathbf{in}$

$\quad\quad \big(\mathsf{i}_1 := [\mathsf{i}]\ ; \mathsf{i}_2 := [\mathsf{i}+1]\ ;$

$\quad\quad \{\exists \tau_1, \tau_2.\ \ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto \mathsf{i}_1, \mathsf{i}_2\ *$

$\quad\quad\quad \mathsf{tree}\ \tau_1\ (\mathsf{i}_1)\ *\ \mathsf{tree}\ \tau_2\ (\mathsf{i}_2))\}$

$\quad\quad \mathbf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\ ;$

Copying Tree (Proof)

$$
\begin{array}{l}
\quad \mathbf{copytree}(\mathsf{i}_1;\mathsf{j}_1)\ ; \\
\quad \{\exists \tau_1,\tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge (\mathsf{i} \mapsto \mathsf{i}_1,\mathsf{i}_2\ * \\
\qquad \mathsf{tree}\ \tau_1\ (\mathsf{i}_1)\ *\ \mathsf{tree}\ \tau_2\ (\mathsf{i}_2)\ *\ \mathsf{tree}\ \tau_1\ (\mathsf{j}_1))\} \\
\quad \mathbf{copytree}(\mathsf{i}_2;\mathsf{j}_2)\ ; \\
\quad \{\exists \tau_1,\tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad (\mathsf{i} \mapsto \mathsf{i}_1,\mathsf{i}_2\ *\ \mathsf{tree}\ \tau_1\ (\mathsf{i}_1)\ *\ \mathsf{tree}\ \tau_2\ (\mathsf{i}_2)\ * \\
\qquad \mathsf{tree}\ \tau_1\ (\mathsf{j}_1)\ *\ \mathsf{tree}\ \tau_2\ (\mathsf{j}_2))\} \\
\quad \mathsf{j} := \mathbf{cons}(\mathsf{j}_1,\mathsf{j}_2) \\
\quad \{\exists \tau_1,\tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad (\mathsf{i} \mapsto \mathsf{i}_1,\mathsf{i}_2\ *\ \mathsf{tree}\ \tau_1\ (\mathsf{i}_1)\ *\ \mathsf{tree}\ \tau_2\ (\mathsf{i}_2)\ * \\
\qquad \mathsf{j} \mapsto \mathsf{j}_1,\mathsf{j}_2\ *\ \mathsf{tree}\ \tau_1\ (\mathsf{j}_1)\ *\ \mathsf{tree}\ \tau_2\ (\mathsf{j}_2))\} \\
\quad \{\exists \tau_1,\tau_2.\ \tau = (\tau_1 \cdot \tau_2) \wedge \\
\qquad (\mathsf{tree}\ (\tau_1 \cdot \tau_2)\ (\mathsf{i})\ *\ \mathsf{tree}\ (\tau_1 \cdot \tau_2)\ (\mathsf{j}))\}\big) \\
\{\mathsf{tree}\ \tau(\mathsf{i})\ *\ \mathsf{tree}\ \tau(\mathsf{j})\}.
\end{array}
$$

Automated Verification

Modular 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

Core Imperative Language

$$
\begin{array}{lll}
P & ::= tdecl^*\ meth^* & tdecl ::= datat \mid spred \\
datat & ::= \mathtt{data}\ c\ \{\ field^*\ \} & field ::= t\ v \qquad t ::= c \mid \tau \\
\tau & ::= \mathtt{int} \mid \mathtt{bool} \mid \mathtt{float} \mid \mathtt{void} & \\
spred & ::= c\langle v^* \rangle \equiv \Phi\ \mathtt{inv}\ \ \pi_0 & \\
meth & ::= t\ mn\ ((t\ v)^*)\ \mathtt{where}\ \Phi_{pr} \ast\!\!\!\!\longrightarrow \Phi_{po}\ \{e\} & \\
e & ::= \mathtt{null} \mid k^\tau \mid v \mid v.f \mid v{:=}e \mid v_1.f{:=}v_2 \mid \mathtt{new}\ c(v^*) & \\
 & \quad \mid e_1;e_2 \mid t\ v;\ e \mid mn(v^*) \mid \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 & \\
 & \quad \mid \mathtt{while}\ v\ \mathtt{where}\ \Phi_{pr} \ast\!\!\!\!\longrightarrow \Phi_{po}\ \mathtt{do}\ e & \\
\Phi & ::= \bigvee (\exists v^* {\cdot} \kappa \wedge \pi)^* & \pi ::= \gamma \wedge \phi \\
\gamma & ::= v_1{=}v_2 \mid v{=}\mathtt{null} \mid v_1{\neq}v_2 \mid v{\neq}\mathtt{null} \mid \gamma_1 \wedge \gamma_2 & \\
\kappa & ::= \mathtt{emp} \mid v{::}c\langle v^* \rangle \mid \kappa_1 * \kappa_2 & \\
\Delta & ::= \Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta \wedge \pi \mid \Delta_1 {*} \Delta_2 \mid \exists v {\cdot} \Delta & \\
\phi & ::= b \mid a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg \phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi & \\
b & ::= \mathtt{true} \mid \mathtt{false} \mid v \mid b_1 = b_2 & a ::= s_1{=}s_2 \mid s_1{\leq}s_2 \\
s & ::= k^{\mathtt{int}} \mid v \mid k^{\mathtt{int}} {\times} s \mid s_1{+}s_2 \mid -s \mid max(s_1,s_2) \mid min(s_1,s_2) &
\end{array}
$$

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 $\mathtt{p}{::}\mathtt{c}\langle \mathtt{v}^* \rangle$ to denote two things in our system. When $\mathtt{c}$ is a data name, $\mathtt{p}{::}\mathtt{c}\langle \mathtt{v}^* \rangle$ stands for singleton heap $\mathtt{p}{\mapsto}[(\mathtt{f} : \mathtt{v})]^*$ where $\mathtt{f}^*$ are fields of data declaration $\mathtt{c}$. When $\mathtt{c}$ is a predicate name, $\mathtt{p}{::}\mathtt{c}\langle \mathtt{v}^* \rangle$ stands for the formula $\mathtt{c}(\mathtt{p}, \mathtt{v}^*)$.

Shape Predicates

Linked-list with size

$$\texttt{ll}\langle n\rangle \equiv (\texttt{self}=\texttt{null} \wedge n=0) \vee (\exists i,m,q \cdot \texttt{self}::\texttt{node}\langle i,q\rangle * q::\texttt{ll}\langle m\rangle \wedge n=m+1)\ \mathrm{inv}\ n\geq 0$$

Double linked-list (right traversal) with size

$$\texttt{dll}\langle p,n\rangle \equiv (\texttt{self}=\texttt{null} \wedge n=0) \vee (\texttt{self}::\texttt{node2}\langle \_,p,q\rangle * q::\texttt{dll}\langle \texttt{self},n-1\rangle)\ \mathrm{inv}\ n\geq 0$$

Sorted linked-list with size, min, max

$$\begin{aligned}\texttt{sortl}\langle n,\mathit{min},\mathit{max}\rangle \equiv\ & (\texttt{self}::\texttt{node}\langle \mathit{min},\texttt{null}\rangle \wedge \mathit{min}=\mathit{max} \wedge n=1)\\ & \vee (\texttt{self}::\texttt{node}\langle \mathit{min},q\rangle * q::\texttt{sortl}\langle n-1,k,\mathit{max}\rangle \wedge \mathit{min}\leq k)\ \mathrm{inv}\ \mathit{min}\leq \mathit{max} \wedge n\geq 1\end{aligned}$$

Insertion Sort Algorithm

```
node insert(node x, node vn) where
  x::sortl⟨n, sm, lg⟩ ∗ vn::node⟨v, _⟩ ⤖ res::sortl⟨n+1, min(v, sm), max(v, lg)⟩
{ if (vn.val≤x.val) then { vn.next:=x; vn }
  else if (x.next=null) then { x.next:=vn; vn.next:=null; x }
  else { x.next:=insert(x.next, vn); x }}

node insertion_sort(node y) where y::ll⟨n⟩ ∧ n>0 ⤖ res::sortl⟨n, _, _⟩
{ if (y.next=null) then y
   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.

$$\texttt{while}\ x<0\ \texttt{where true} \ ⤖\ (x>0 \wedge x'=x) \vee (x\leq 0 \wedge x'=0)\ \texttt{do}\ \{\ x:=x+1\ \}$$

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

Prime Notation

Example :

```
{x'=x ∧ y'=y}
  x:=x+1
{x'=x+1 ∧ y'=y}
x:=x+y
 {x'=x+1+y ∧ y'=y}
y:=2
 {x'=x+1+y ∧ y'=2}
```

Forward Verification

Given Δ_1, infer Δ_2 :

$$\vdash \{\Delta_1\}\ e\ \{\Delta_2\}$$

$$\frac{V=\{v_1..v_n\}\quad W=prime(V)\quad \Delta=\Phi_{pr}\wedge nochange(V)\quad \vdash\{\Delta\}\,e\,\{\Delta_1\}\quad (\exists W\cdot\Delta_1)\vdash\Phi_{po}*\Delta_2}{\vdash t_0\ \ mn(t_1\ v_1,..,t_n\ v_n)\ \texttt{where}\ \Phi_{pr} \mathrel{*\!\!\!\rightarrow} \Phi_{po}\ \{e\}}\ \text{[FV-METH]}$$

$$\frac{\begin{array}{c} t\ mn((t_i\ v_i)_{i=1}^{n})\ \texttt{where}\ \Phi_{pr} \mathrel{*\!\!\!\rightarrow} \Phi_{po}\ \{..\} \quad \rho=[v_i'/v_i] \\ \Delta\vdash\rho\Phi_{pr}*\Delta_1 \quad W=\{v_1,..,v_n\} \quad \Delta_2=(\Delta_1 *_W \Phi_{po}) \end{array}}{\vdash\{\Delta\}\,m(v_1..v_n)\,\{\Delta_2\}}\ \text{[FV-CALL]}$$

Forward Verification

$$\frac{\Delta_1=(\Delta\wedge eq_\tau(\texttt{res},k))}{\vdash\{\Delta\}\,k^\tau\,\{S\}}\ \text{[FV-CONST]} \qquad \frac{\vdash\{\Delta\}\,e\,\{\Delta_1\}}{\vdash\{\Delta\}\,\{t\ v;\ \ e\}\,\{\exists v,v'\cdot\Delta_1\}}\ \text{[FV-LOCAL]} \qquad \frac{\Delta_1=(\Delta*\texttt{res}::c\langle v_1',..,v_n'\rangle)}{\vdash\{\Delta\}\,\texttt{new}\ c(v_1,..,v_n)\,\{\Delta_1\}}\ \text{[FV-NEW]}$$

$$\frac{\Delta_1=(\Delta\wedge\texttt{res}=v')}{\vdash\{\Delta\}\,v\,\{\Delta_1\}}\ \text{[FV-VAR]} \qquad \frac{\vdash\{\Delta\}\,e\,\{\Delta_1\}\quad \Delta_2=\exists\texttt{res}\cdot(\Delta_1\wedge_{\{v\}}v'=\texttt{res})}{\vdash\{\Delta\}\,v{:=}e\,\{\Delta_2\}}\ \text{[FV-ASSIGN]}$$

$$\frac{\begin{array}{c}\vdash\{\Delta\}\,e_1\,\{\Delta_1\}\\ \vdash\{\Delta_1\}\,e_2\,\{\Delta_2\}\end{array}}{\vdash\{\Delta\}\,e_1;e_2\,\{\Delta_2\}}\ \text{[FV-SEQ]} \qquad \frac{\vdash\{\Delta\wedge v'\}\,e_1\,\{\Delta_1\}\quad \vdash\{\Delta\wedge\neg v'\}\,e_2\,\{\Delta_2\}}{\vdash\{\Delta\}\,\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\,\{\Delta_1\vee\Delta_2\}}\ \text{[FV-IF]}$$

$$\frac{\begin{array}{c}\Delta\vdash v'::c\langle v_1,..,v_n\rangle*\Delta_1 \quad fresh\ v_1..v_n\\ \Delta_2=\exists v_1..v_n\cdot(\Delta_1*v'::c\langle v_1,..,v_n\rangle\wedge\texttt{res}=v_i)\end{array}}{\vdash\{\Delta\}\,v.f_i\,\{\Delta_2\}}\ \text{[FV-FIELD-READ]} \qquad \frac{\begin{array}{c}\Delta\vdash v'::c\langle v_1,..,v_n\rangle*\Delta_1 \quad fresh\ v_1..v_n\\ \Delta_2=\exists v_1..v_n\cdot(\Delta_1*v'::[v_0'/v_i]c\langle v_1,..,v_n\rangle)\end{array}}{\vdash\{\Delta\}\,v.f_i{:=}v_0\,\{\Delta_2\}}\ \text{[FV-FIELD-UPDATE]}$$

Separation Constraint Normalization Rules

Target :

$$\begin{array}{lll} \Phi & ::= \bigvee(\exists v^*\cdot\kappa\wedge\pi)^* & \pi ::= \gamma\wedge\phi \\ \gamma & ::= v_1{=}v_2 \mid v{=}\texttt{null} \mid v_1{\neq}v_2 \mid v{\neq}\texttt{null} \mid \gamma_1\wedge\gamma_2 & \\ \kappa & ::= \texttt{emp} \mid v::c\langle v^*\rangle \mid \kappa_1*\kappa_2 & \end{array}$$

$$\begin{array}{ll} (\Delta_1\vee\Delta_2)\wedge\pi & \leadsto (\Delta_1\wedge\pi)\vee(\Delta_2\wedge\pi) \\ (\Delta_1\vee\Delta_2)*\Delta & \leadsto (\Delta_1*\Delta)\vee(\Delta_2*\Delta) \\ (\kappa_1\wedge\pi_1)*(\kappa_2\wedge\pi_2) & \leadsto (\kappa_1*\kappa_2)\wedge(\pi_1\wedge\pi_2) \\ (\kappa_1\wedge\pi_1)\wedge(\pi_2) & \leadsto \kappa_1\wedge(\pi_1\wedge\pi_2) \end{array}$$

$$\begin{array}{ll} (\gamma_1\wedge\phi_1)\wedge(\gamma_2\wedge\phi_2) & \leadsto (\gamma_1\wedge\gamma_2)\wedge(\phi_1\wedge\phi_2) \\ (\exists x\cdot\Delta)\wedge\pi & \leadsto \exists y\cdot([y/x]\Delta\wedge\pi) \\ (\exists x\cdot\Delta_1)*\Delta_2 & \leadsto \exists y\cdot([y/x]\Delta_1*\Delta_2) \end{array}$$

Separation Constraint Approximation

$XPure_n(\Phi)$ returns a sound approximation of the form : $\textbf{ex}\ i^*\cdot\bigvee(\exists v^*\cdot\pi)^*$

non-null symbolic addresses

$$\begin{array}{l} XPure_n(\texttt{p}_1::\texttt{node}\langle\_,\_\rangle*\texttt{p}_2::\texttt{node}\langle\_,\_\rangle) \\ \quad =(\textbf{ex}\ \texttt{i}_1\cdot(\texttt{p}_1{=}\texttt{i}_1\wedge\texttt{i}_1{>}0))\wedge(\textbf{ex}\ \texttt{i}_2\cdot(\texttt{p}_2{=}\texttt{i}_2\wedge\texttt{i}_2{>}0)) \\ \quad =\textbf{ex}\ \texttt{i}_1,\texttt{i}_2\cdot(\texttt{p}_1{=}\texttt{i}_1\wedge\texttt{i}_1{>}0\wedge\texttt{p}_2{=}\texttt{i}_2\wedge\texttt{i}_2{>}0\wedge\texttt{i}_1{\neq}\texttt{i}_2) \end{array}$$

Normalization :

$$\begin{array}{ll} (\textbf{ex}\ I\cdot\phi_1)\vee(\textbf{ex}\ J\cdot\phi_2) & \leadsto \textbf{ex}\ I{\cup}J\cdot(\phi_1\vee\phi_2) \\ \exists\ v\cdot(\textbf{ex}\ I\cdot\phi) & \leadsto \textbf{ex}\ I\cdot(\exists\ v\cdot\phi) \\ (\textbf{ex}\ I\cdot\phi_1)\wedge(\textbf{ex}\ J\cdot\phi_2) & \leadsto \textbf{ex}\ I{\cup}J\cdot\phi_1\wedge\phi_2\wedge\bigwedge_{i\in I,j\in J}i{\neq}j \end{array}$$

Translating to Pure Form

$$\frac{(c\langle v^*\rangle \equiv \Phi \ \texttt{inv} \ \pi_0) \in P}{Inv_0(p{::}c\langle v^*\rangle) = [p/\texttt{self}, 0/\texttt{null}]\pi_0}$$

$$\frac{(c\langle v^*\rangle \equiv \Phi \ \texttt{inv} \ \pi_0) \in P}{Inv_n(p{::}c\langle v^*\rangle) = [p/\texttt{self}, 0/\texttt{null}]XPure_{n-1}(\Phi)}$$

$$XPure_n(\bigvee(\exists v^* {\cdot} \kappa \wedge \pi)^*) =_{df} \bigvee(\exists v^* {\cdot} XPure_n(\kappa) \wedge [0/\texttt{null}]\pi)^*$$

$$XPure_n(\texttt{emp}) \ =_{df} \ \texttt{true}$$

$$\frac{IsData(c) \quad fresh \ i}{XPure_n(p{::}c\langle v^*\rangle) \ =_{df} \ \textbf{ex} \ i {\cdot} (p{=}i \wedge i{>}0)}$$

$$\frac{IsPred(c) \quad fresh \ i^* \quad Inv_n(p{::}c\langle v^*\rangle) = \textbf{ex} \ j^* \cdot \bigvee(\exists u^* {\cdot} \pi)^*}{XPure_n(p{::}c\langle v^*\rangle) \ =_{df} \ \textbf{ex} \ i^* \cdot [i^*/j^*] \bigvee(\exists u^* {\cdot} \pi)^*}$$

$$XPure_n(\kappa_1 * \kappa_2) \ =_{df} \ XPure_n(\kappa_1) \wedge XPure_n(\kappa_2)$$

Deriving Shape Invariant

From each pure invariant, such as `(n ≥ 0)` for `ll<n>`

We use `Inv`$_1$`(..)` to obtain a more precise invariant :

$$\textbf{ex} \ \texttt{i} {\cdot} (\texttt{self}{=}0 \wedge \texttt{n}{=}0 \vee \texttt{self}{=}\texttt{i} \wedge \texttt{i}{>}0 \wedge \texttt{n}{>}0)$$

$$\frac{(c\langle v^*\rangle \equiv \Phi \ \texttt{inv} \ \pi_0) \in P}{Inv_0(p{::}c\langle v^*\rangle) = [p/\texttt{self}, 0/\texttt{null}]\pi_0}$$

$$\frac{(c\langle v^*\rangle \equiv \Phi \ \texttt{inv} \ \pi_0) \in P}{Inv_n(p{::}c\langle v^*\rangle) = [p/\texttt{self}, 0/\texttt{null}]XPure_{n-1}(\Phi)}$$

Separation Constraint Entailment

$$\Delta_A \vdash^{\kappa}_{V} \Delta_C * \Delta_R$$

denotes

$$\kappa * \Delta_A \vdash \exists V {\cdot} (\kappa * \Delta_C) * \Delta_R$$

The purpose of heap entailment is to check that heap nodes in the antecedent Δ_A are sufficiently precise to cover all nodes from the consequent Δ_C, and to compute a residual heap state Δ_R. κ 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 k and V are derived. The entailment checking procedure is invoked with $\kappa = \texttt{emp}$ and $V = \emptyset$. The en-

Separation Constraint Entailment

$$\frac{\begin{array}{c}[\textbf{ENT-EMP}] \\ \rho{=}[0/\texttt{null}] \\ \rho(XPure_n(\kappa_1{*}\kappa) \wedge \pi_1) {\Longrightarrow} \rho \exists V {\cdot} \pi_2\end{array}}{\kappa_1 \wedge \pi_1 \vdash^{\kappa}_{V} \pi_2 * (\kappa_1 \wedge \pi_1)}$$

$$\frac{\begin{array}{c}[\textbf{ENT-MATCH}] \\ XPure_n(p_1{::}c\langle v_1^*\rangle {*} \kappa_1 {*} \pi_1) {\Longrightarrow} p_1{=}p_2 \quad \rho{=}[v_1^*/v_2^*] \\ \kappa_1 \wedge \pi_1 \wedge freeEqn(\rho, V) \vdash^{\kappa * p_1::c\langle v_1^*\rangle}_{V-\{v_2^*\}} \rho(\kappa_2 \wedge \pi_2) * \Delta\end{array}}{p_1{::}c\langle v_1^*\rangle {*} \kappa_1 \wedge \pi_1 \vdash^{\kappa}_{V} (p_2{::}c\langle v_2^*\rangle {*} \kappa_2 \wedge \pi_2) * \Delta}$$

$$\frac{\begin{array}{c}[\textbf{ENT-FOLD}] \\ IsPred(\mathsf{c}_2) \wedge IsData(\mathsf{c}_1) \quad (\Delta^r, \kappa^r, \pi^r) {\in} fold^{\kappa}(p_1{::}c_1\langle v_1^*\rangle {*} \kappa_1 \wedge \pi_1, p_2{::}c_2\langle v_2^*\rangle) \\ XPure_n(p_1{::}c_1\langle v_1^*\rangle {*} \kappa_1 {*} \pi_1) {\Longrightarrow} p_1{=}p_2 \quad (\pi^a, \pi^c){=}split^{\{v_2^*\}}_{V}(\pi^r) \quad \Delta^r \wedge \pi^a \vdash^{\kappa^r}_{V} (\kappa_2 \wedge \pi_2 \wedge \pi^c) * \Delta\end{array}}{p_1{::}c_1\langle v_1^*\rangle {*} \kappa_1 \wedge \pi_1 \vdash^{\kappa}_{V} (p_2{::}c_2\langle v_2^*\rangle {*} \kappa_2 \wedge \pi_2) * \Delta}$$

$$\frac{\begin{array}{c}[\textbf{ENT-UNFOLD}] \\ XPure_n(p_1{::}c_1\langle v_1^*\rangle {*} \kappa_1 {*} \pi_1) {\Longrightarrow} p_1{=}p_2 \quad IsPred(\mathsf{c}_1) \wedge IsData(\mathsf{c}_2) \\ unfold(p_1{::}c_1\langle v_1^*\rangle) {*} \kappa_1 \wedge \pi_1 \vdash^{\kappa}_{V} (p_2{::}c_2\langle v_2^*\rangle {*} \kappa_2 \wedge \pi_2) * \Delta\end{array}}{p_1{::}c_1\langle v_1^*\rangle {*} \kappa_1 \wedge \pi_1 \vdash^{\kappa}_{V} (p_2{::}c_2\langle v_2^*\rangle {*} \kappa_2 \wedge \pi_2) * \Delta}$$

$$\frac{\begin{array}{c}[\textbf{ENT-LHS-OR}] \\ \Delta_1 \vdash^{\kappa}_{V} \Delta_3 * \Delta_4 \\ \Delta_2 \vdash^{\kappa}_{V} \Delta_3 * \Delta_5\end{array}}{\Delta_1 \vee \Delta_2 \vdash^{\kappa}_{V} \Delta_3 * (\Delta_4 \vee \Delta_5)}$$

$$\frac{\begin{array}{c}[\textbf{ENT-RHS-OR}] \\ \Delta_1 \vdash^{\kappa}_{V} \Delta_i * \Delta_i^R\end{array}}{\Delta_1 \vdash^{\kappa}_{V} (\Delta_2 \vee \Delta_3) * \Delta_i^R} i {\in} \{2,3\}$$

$$\frac{\begin{array}{c}[\textbf{ENT-RHS-EX}] \\ \Delta_1 \vdash^{\kappa}_{V \cup \{w\}} ([w/v]\Delta_2) * \Delta_3 \\ fresh \ w \quad \Delta{=}\exists \ w \cdot \Delta_3\end{array}}{\Delta_1 \vdash^{\kappa}_{V} (\exists \ v \cdot \Delta_2) * \Delta_3}$$

$$\frac{\begin{array}{c}[\textbf{ENT-LHS-EX}] \\ [w/v]\Delta_1 \vdash^{\kappa}_{V} \Delta_2 * \Delta \\ fresh \ w\end{array}}{\exists v \cdot \Delta_1 \vdash^{\kappa}_{V} \Delta_2 * \Delta}$$

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 :

$$\mathtt{x::ll}\langle \mathtt{n}\rangle \wedge \mathtt{n}{>}3 \ \vdash \ (\exists \mathtt{r}\cdot \mathtt{x::node}\langle \_, \mathtt{r}\rangle * \mathtt{r::node}\langle \_, \mathtt{y}\rangle \wedge \mathtt{y}{\neq}\mathtt{null}) * \Delta_R$$

$$\begin{array}{ll}
\exists \mathtt{q_1}\cdot \mathtt{x::node}\langle _, \mathtt{q_1}\rangle * \mathtt{q_1::ll}\langle \mathtt{n}-1\rangle \wedge \mathtt{n}{>}3 & \vdash (\exists \mathtt{r}\cdot \mathtt{x::node}\langle _, \mathtt{r}\rangle * \mathtt{r::node}\langle _, \mathtt{y}\rangle \wedge \mathtt{y}{\neq}\mathtt{null}) * \Delta_R \\
\mathtt{q_1::ll}\langle \mathtt{n}-1\rangle \wedge \mathtt{n}{>}3 & \vdash (\mathtt{q_1::node}\langle _, \mathtt{y}\rangle \wedge \mathtt{y}{\neq}\mathtt{null}) * \Delta_R \\
\exists \mathtt{q_2}\cdot \mathtt{q_1::node}\langle _, \mathtt{q_2}\rangle * \mathtt{q_2::ll}\langle \mathtt{n}-2\rangle \wedge \mathtt{n}{>}3 & \vdash \mathtt{q_1::node}\langle _, \mathtt{y}\rangle \wedge \mathtt{y}{\neq}\mathtt{null} * \Delta_R \\
\mathtt{q_2::ll}\langle \mathtt{n}-2\rangle \wedge \mathtt{n}{>}3 \wedge \mathtt{q_2}{=}\mathtt{y} & \vdash \mathtt{y}{\neq}\mathtt{null} * \Delta_R
\end{array}$$

$$\frac{\begin{array}{c}[\underline{\text{UNFOLDING}}]\\ c\langle v^*\rangle \equiv \Phi \in P\end{array}}{unfold(p::c\langle v^*\rangle) \ =_{df} \ [p/\mathtt{self}]\Phi}$$

Folding a Predicate in Consequent

We apply a fold operation when a data node in the antecedent matches with a predicate in the consequent. An example is :

$$\mathtt{x::node}\langle 1, \mathtt{q_1}\rangle * \mathtt{q_1::node}\langle 2, \mathtt{null}\rangle * \mathtt{y::node}\langle 3, \mathtt{null}\rangle \ \vdash \ \mathtt{x::ll}\langle \mathtt{n}\rangle \wedge \mathtt{n}{>}1 * \Delta_R$$

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

$$\mathtt{y::node}\langle 3, \mathtt{null}\rangle \wedge \mathtt{n}{=}2 \ \vdash \ \mathtt{n}{>}1 * \Delta_R$$

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

Folding a Predicate in Consequent

$$\frac{\begin{array}{c}[\underline{\text{FOLDING}}]\\ c\langle v^*\rangle \equiv \Phi \in P \quad W_i = V_i - \{v^*, p\} \\ \kappa \wedge \pi \vdash^{\kappa'}_{\{p, v^*\}} [p/\mathtt{self}]\Phi * \{(\Delta_i, \kappa_i, V_i, \pi_i)\}_{i=1}^n\end{array}}{fold^{\kappa'}(\kappa \wedge \pi, p::c\langle v^*\rangle) =_{df} \{(\Delta_i, \kappa_i, \exists W_i \cdot \pi_i)\}_{i=1}^n}$$

version of entailment that returns three extra things: (i) consumed heap nodes, (ii) existential variables used, and (iii) final consequent. The final consequent is used to return a constraint for $\{v^*\}$ via $\exists W_i \cdot \pi_i$. A set of answers is returned by the fold step as we allow it to explore multiple ways of matching up with its

Soundness of Entailment

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

Theorem 6.2 (Termination) *The entailment check $\Delta_1 \vdash \Delta_2 * \Delta$ 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].