Introduction

Review of last lecture

Hoare logic

Inference rules

Lists

Trees and DAGs

Review

Store, Heap and States

Allocation, lookup, mutation and deallocation

- $*$ separating conjunction
- $-\!\!*$ separating implication

Hoare logic

Hoare triple {p} c {q}

p precondition, c command, q postcondition

partial correctness: {(assert)} (command) {(assert)}

total correctness: [(assert)](command)[(assert)]

Correctness

Partial correctness:

$\{\mathrm{p}\}\mathrm{c}\{\mathrm{q}\}$ holds $\iff \forall (s,h) \in States_V.\ [\![p]\!]_{asrt}\, sh$ implies
$\neg(c,(s,h) \rightsquigarrow^* \text{abort})$ and
$(\forall (s',h') \in States_V . c,(s,h) \rightsquigarrow^* (s',h')$ implies $[\![q]\!]_{asrt}\, s'h')$

Total correctness:

$[\mathrm{p}]\mathrm{c}[\mathrm{q}]$ holds $\iff \forall (s,h) \in States_V.\ [\![p]\!]_{asrt}\, sh$ implies
$\neg(c,(s,h) \rightsquigarrow^* \text{abort})$ and
$(\forall (s',h') \in States_V . c,(s,h) \rightsquigarrow^* (s',h')$ implies $[\![q]\!]_{asrt}\, s'h')$
and $\neg(c,(s,h)\uparrow)$

Inference rules

Consequence: $$\frac{p' \Rightarrow p \qquad \{p\}c\{q\} \qquad q \Rightarrow q'}{\{p'\}c\{q'\}}$$

Auxiliary Variable Elimination: $$\frac{\{p\}c\{q\}}{\{\exists v,\, p\}c\{\exists v,\, q\}}$$

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

Rule of Constancy

Rule of constancy: $\dfrac{\{p\}c\{q\}}{\{p \wedge r\}c\{q \wedge r\}}$

Weakness: $\dfrac{\{x \mapsto -\}[x] := 4\{x \mapsto 4\}}{\{x \mapsto - \wedge y \mapsto 3\}[x] := 4\{x \mapsto 4 \wedge y \mapsto 3\}}$ in x=y

Replacement: Frame Rule

Frame Rule $\dfrac{\{p\}c\{q\}}{\{p * r\}c\{q * r\}}$

No variable occuring free in r is modified by c

The Frame Rule is sound: if $\{p\}c\{q\}$ and $[\![p * r]\!]_{asrt}\, sh$ hold, then the heap can be split into $h = h_0 \cdot h_1, h_0 \perp h_1$ such that $[\![p]\!]\, sh_0$ and $[\![r]\!]\, sh_1$ hold

Now if $(c, (s, h)) \rightsquigarrow^* abort$ then $(c, (s, h_0)) \rightsquigarrow^* abort$ contradicting $\{p\}c\{q\}$ and $[\![p]\!]_{asrt}\, sh_0$

As $(c, (s, h)) \rightsquigarrow^* (s', h')$ then $(c, (s, h_0)) \rightsquigarrow^* (s', h'_0)$ where $h' = h'_0 \cdot h_1, h'_0 \perp h_1$. This implies $[\![q]\!]\, sh'_0$ holds and $[\![r]\!]\, sh_1 \Rightarrow [\![r]\!]\, s' h_1$ thus $[\![q * r]\!]\, s' h'$

The Frame Rule allows for globalisation of local rules

Example: Mutation

Mutation (local): $\{e \mapsto -\}[e] := e'\{e \mapsto e'\}$

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

We can return to the local mutation by taking $r = emp$

By taking $r = (e \mapsto e') \mathrel{-\!\!*} p$ and using $q * (q \mathrel{-\!\!*} p) \Rightarrow p$ we find a rule suitable for backwards reasoning.

Mutation (backwards reasoning):
$\{(e \mapsto -) * ((e \mapsto e') \mathrel{-\!\!*} p)\}[e] := e'\{p\}$

Deallocation, Allocation and Lookup

Abreviations: $\overline{e} = e_1, \ldots, e_n$, $e_i' = e_i/v \to v'$, $\overline{e}' = e_1', \ldots, e_n'$

Deallocation (global, backwards reasoning):

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

Allocation (global): $\{r\}\ v := \mathit{cons}(\overline{e})\{\exists v'.(v \mapsto \overline{e}') * r'\}$

Allocation (backwards reasoning):

$\{\forall v'.(v' \mapsto \overline{e}) -\!\!* p'\}\ v := \mathit{cons}(\overline{e})\ \{p\}$

Lookup (global): $\{\exists v''.(e \mapsto v'') * (r/v' \to v)\}\ v :=$
$[e]\ \{\exists v'.(e' \mapsto v) * (r/v'' \to v)\}$

Lookup (backwards reasoning):

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

Lists

Quick notations:

ϵ stands for the empty sequence.

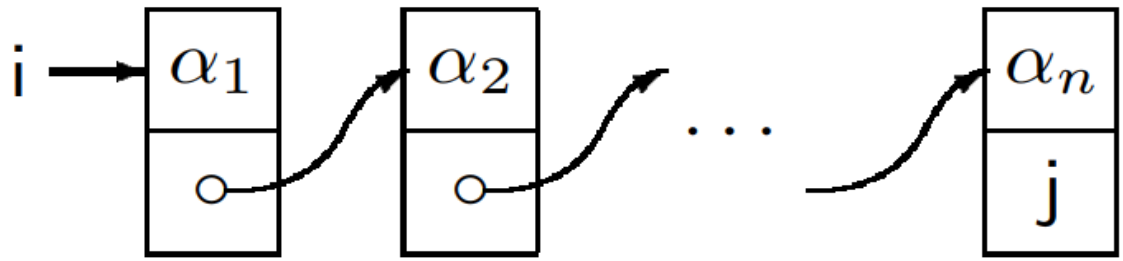
[x] stands for the single-element sequence containing x.

$\alpha \cdot \beta$ stands for the composition of α followed by β.

α_i stands for the ith component of α

Singly-linked lists

Illustration of a singly linked list:



recursive definition:

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

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

Basic Properties of Lists

$$\mathit{list}\ a(i,j) \iff i \mapsto a, j$$

$$\mathit{list}\ \alpha \cdot \beta(i,k) \iff \exists j.\ \mathit{list}\ \alpha(i,j) * \mathit{list}\ \beta(j,k)$$

$$\mathit{list}\ \alpha \cdot b(i,k) \iff \exists j.\ \mathit{list}\ \alpha(i,j) * j \mapsto b, k$$

Cyclic lists

When i=j≠nil the list is not necessarily empty, it could be cyclic.

list $\alpha(i,j) * j \hookrightarrow - \Rightarrow (i = j \Leftrightarrow \alpha = \epsilon)$

list $\alpha(i,j) *$ list $\beta(j, nil) \Rightarrow (i = j \Leftrightarrow \alpha = \epsilon)$

Deletion of the first element of a list

$\{lista \cdot \alpha(i,k)\}$

$\{\exists j.i \mapsto a, j * list\alpha(j,k)\}$

$\{i \mapsto a * \exists j.i+1 \mapsto j * list\alpha(j,k)\}$

$j := [i+1];$

$\{i \mapsto a * i+1 \mapsto j * list\alpha(j,k)\}$

dispose i;

$\{i+1 \mapsto j * list\alpha(j,k)\}$

dispose i+1;

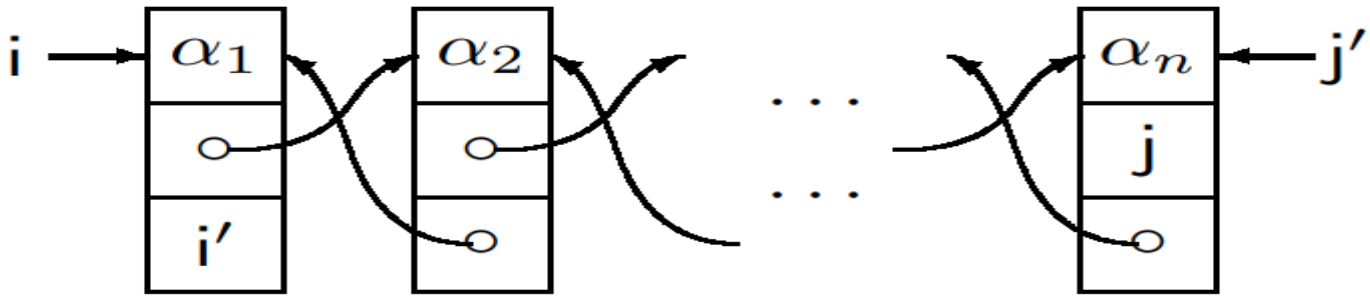
$\{list\alpha(j,k)\}$

i:=j;

$\{list\alpha(i,k)\}$

Doubly-linked lists

Illustration of a doubly-linked list:



recursive definition:

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

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

S-expressions

S-expressions are defined as:

τ is an S-expression $\iff$

$\tau \in$ Atoms or

$\tau = (\tau_1 \cdot \tau_2)$ where $\tau_1, \tau_2 \in$ S-exps.

The abstract values "S-expressions" are represented by "trees" and "dags" (directed acyclic graphs)

Definition of trees and dags

Inductive definition of trees and dags:

Trees: tree a(i) $\iff$ emp $\wedge$ i=a

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

Dag: dag a(i) $\iff$ i=a

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

We need the separating conjunction to prevent cyclic graphs.

Tree manipulation and required procedures

A procedure is defined in the form:

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

h is not free, $x_1, \cdots, x_m$ are free variables of c, and $y_1, \cdots, y_m$ are free variables modified by procedure c.

Copytree

Here we define a tree copying procedure:

copytree(i;j)=

    if isatom(i) then j:=i else

        newvar i_1, i_2, j_1, j_2 in

            (i_1:=[i];i_2:=[i+1];

            copytree(i_1;j_1);copytree(i_2,j_2);

            j:=cons(j_1,j_2))

Copytree specification

$$\{tree\ \tau(i)\}\ copytree(i,j)\ \{tree\ \tau(i) * tree\ \tau(j)\}$$

$$\{dag\ \tau(i)\}\ copytree(i,j)\ \{dag\ \tau(i) * tree\ \tau(j)\}?$$

Relation to Rust

Rust rules around ownership can translated into separation logic

Soundness of a Hoare triple corollates to a correctly typed rust program not having memory problems

Questions?