

Separation Logic (SL)

LCI Seminar, Giulio Paparelli

Introduction

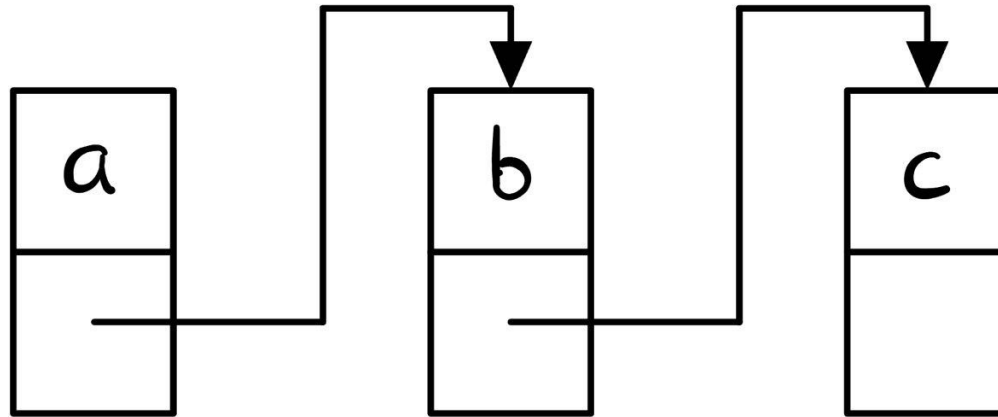
SL is an extension of Hoare Logic that permits reasoning about programs that use **shared** mutable data structures.

shared: structures where an updatable field can be *referenced* from more than one point, using aliases.

Correctness of such programs depends on
complex restrictions on these data structures.

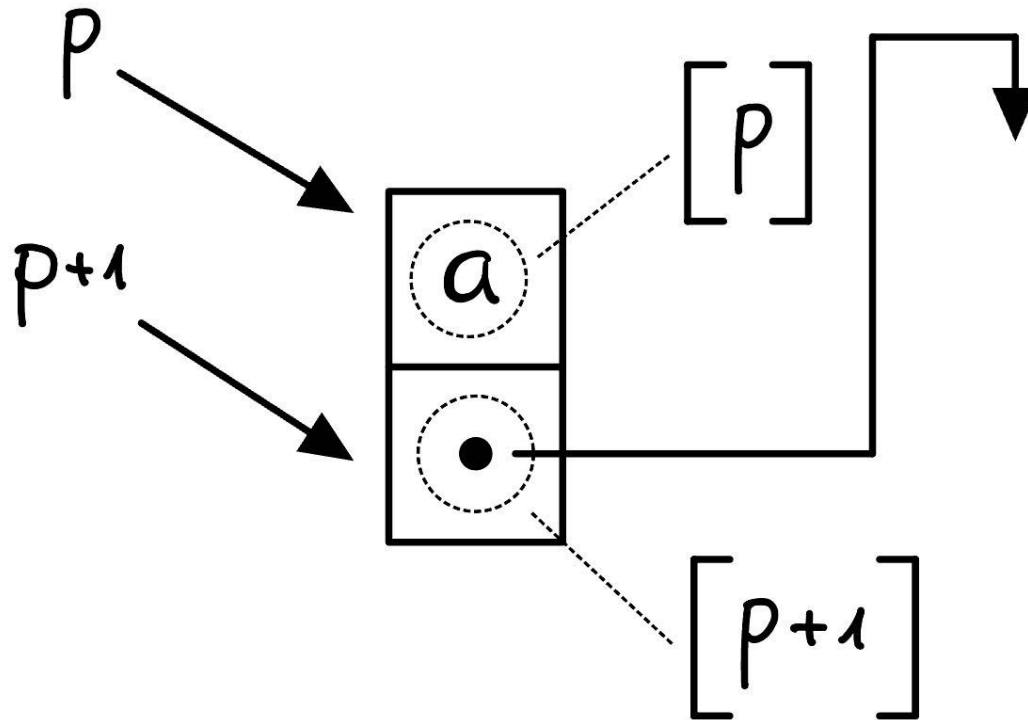
Motivating Example

Consider the following Linked List:



We call *sequence* the concatenation of the elements of the list and represent it with greek letters: $\alpha = abc$

We also introduce the following notation.



$[e]$ denotes the contents of the storage at address e

We define the predicate $\text{list } \alpha \ i$ by induction on the length of the sequence α

$$\text{list } \epsilon \ i \stackrel{\text{def}}{=} i = \mathbf{nil}$$

$$\underline{\text{list}(a \cdot \alpha) \ i} \stackrel{\text{def}}{=} \exists j. \underline{i \hookrightarrow a, j} \wedge \underline{\text{list } \alpha \ j}$$

$(a \cdot \alpha) \ i$ is a list

the pointer i points to
exists a pointer j st. the list “block” a, j

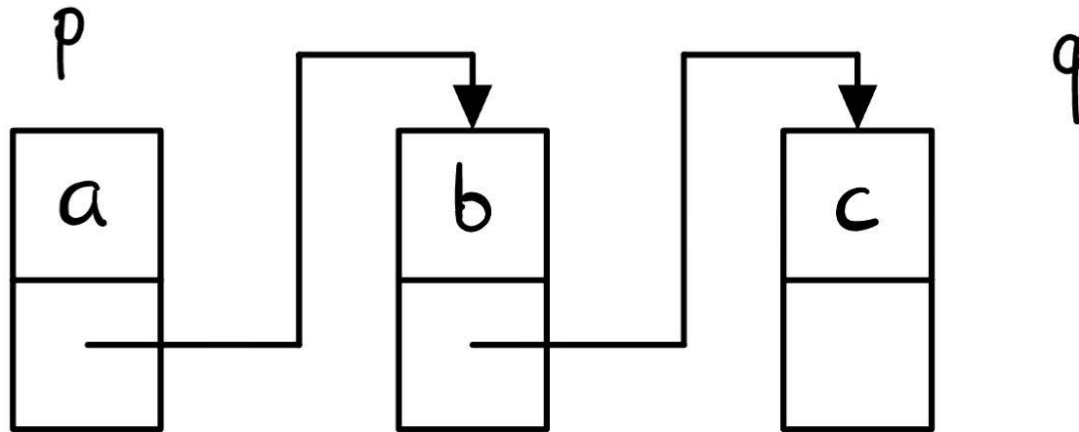
$\alpha \ j$ is a list

Consider now the following program that performs an in-place reversal of a list.

```
q := nil;  
while p ≠ nil do a)  
    t := [p+1];  
    [p+1] := q; b)  
    q := p; c)  
    p := t; d)
```

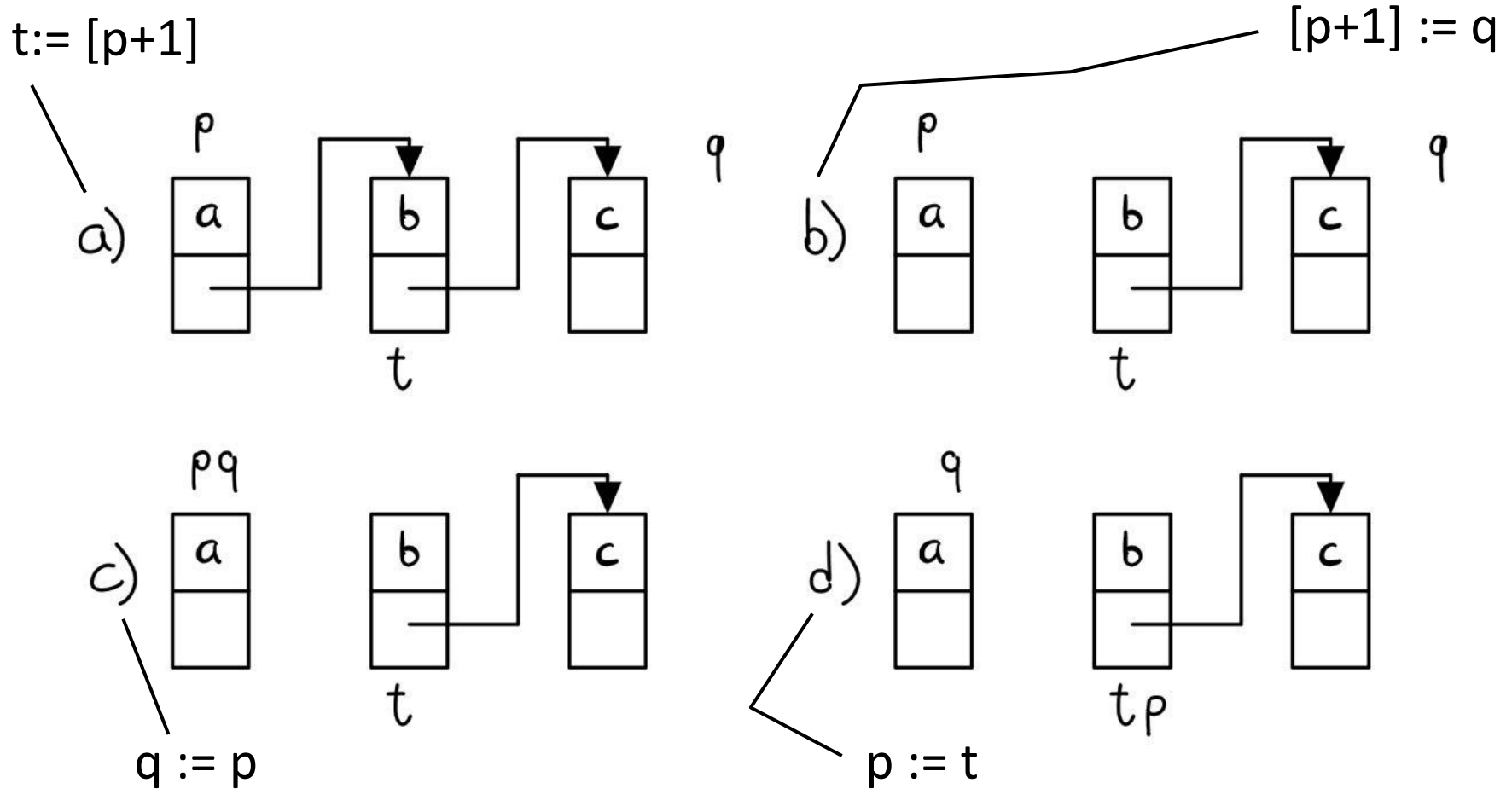
List Reversal Simulation

Starting Point:



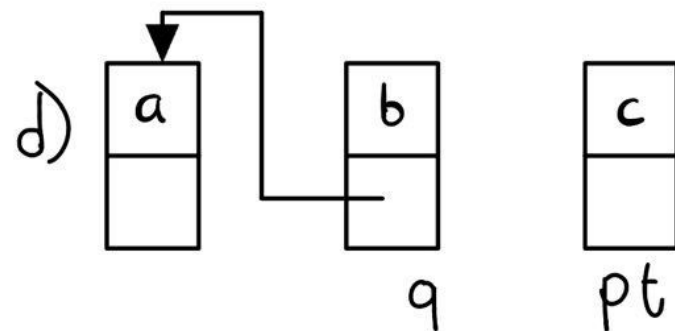
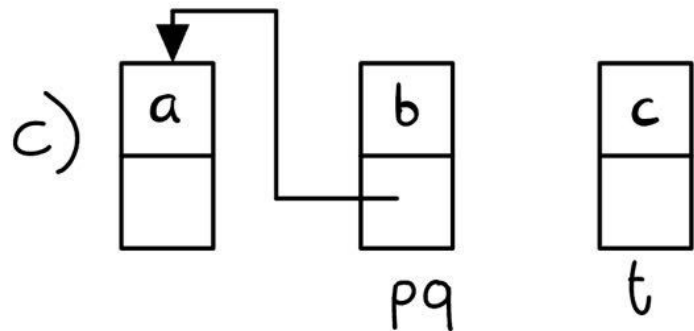
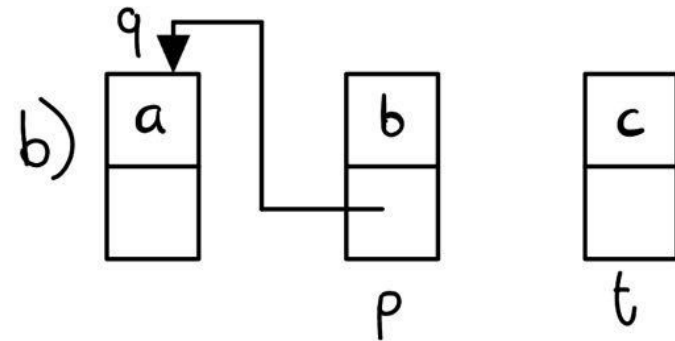
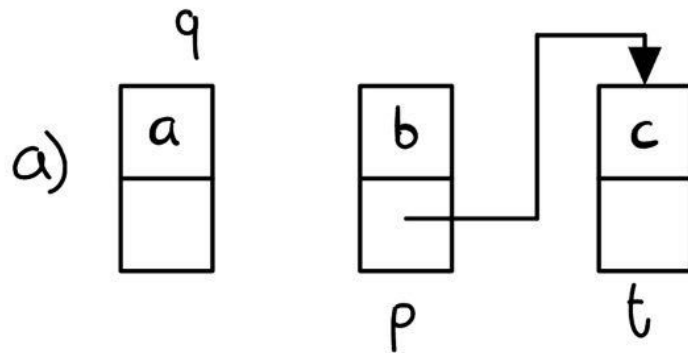
List Reversal Simulation

First While Iteration:



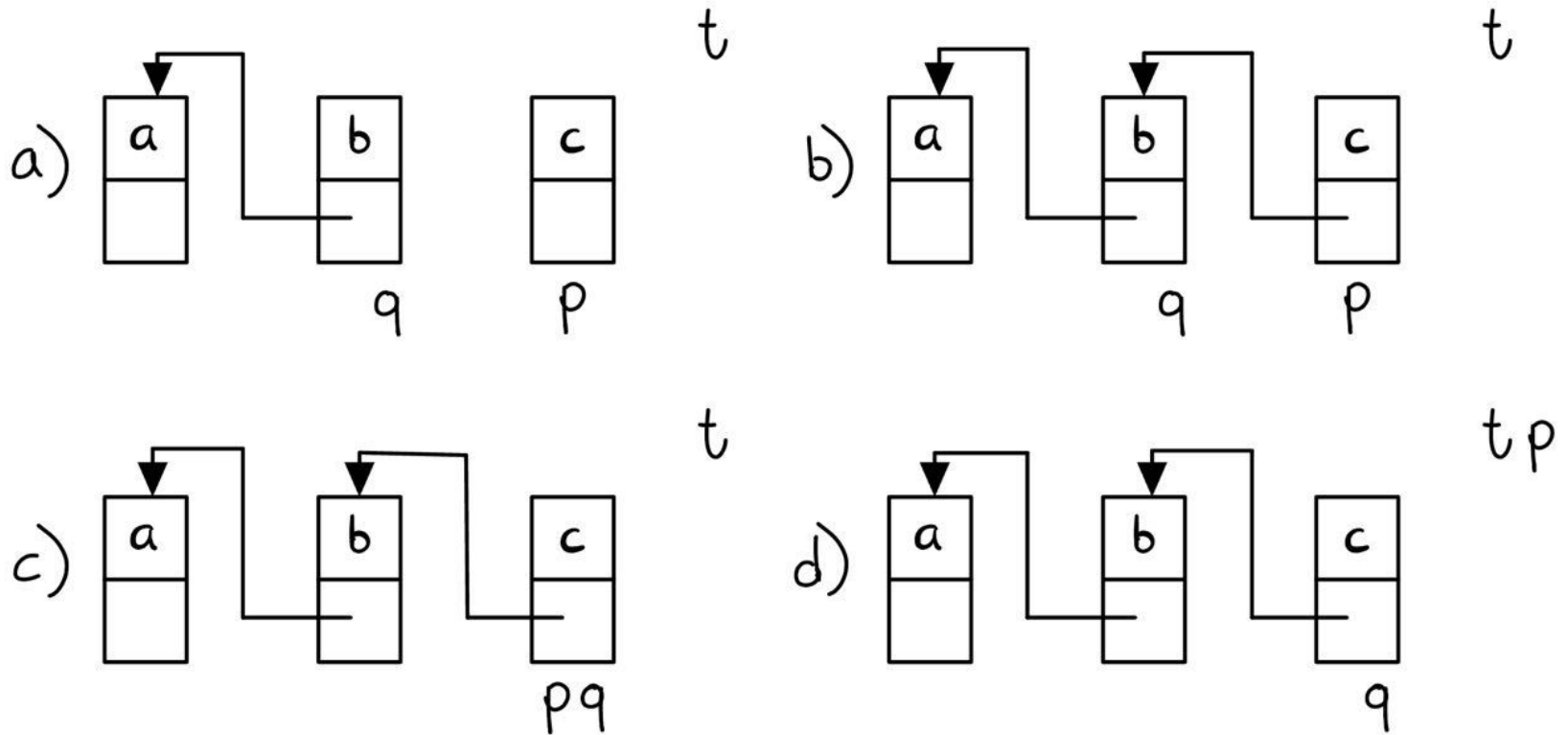
List Reversal Simulation

Second While Iteration:



List Reversal Simulation

Third While Iteration:



Invariant

i and j are lists representing two sequences α and β such that the reflection of the initial value α_0 can be obtained by concatenating the reflection of α onto β

Invariant

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

exists two sequences α
and β st. list α i and list β j
are list

the whole reversed
sequence is equal to the
concatenation of the
reflection of α and β

The Invariant is NOT Enough

The program will malfunction if there is any sharing between list i and list j.

The Invariant is NOT Enough

We can provide an invariant that prohibit all kinds of sharing, but it is clear that this form of reasoning **scales poorly**

SL Separating Conjunction

SL introduce a novel logical operator $P * Q$ that asserts that P and Q hold for **disjoint** portions of the addressable memory.

The Invariant is Enough

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

prohibition of sharing is built in
into this operation

The Programming Language

A simple imperative programming language with commands for the manipulation of mutable shared data structure.

skip, assignment, ...

$\langle \text{comm} \rangle ::= \dots$

- | $\langle \text{var} \rangle := \mathbf{cons}(\langle \text{exp} \rangle, \dots, \langle \text{exp} \rangle)$ allocation
- | $\langle \text{var} \rangle := [\langle \text{exp} \rangle]$ lookup
- | $[\langle \text{exp} \rangle] := \langle \text{exp} \rangle$ mutation
- | $\mathbf{dispose} \langle \text{exp} \rangle$ deallocation

To permit unrestricted address arithmetic
we assume:

- $\text{Values} = \text{Integers}$ disjoint
- $\text{Atoms} \cup \text{Addresses} \subseteq \text{Integers}$ integers that are not addresses
- $\text{Heaps} = \bigcup_{\substack{\text{fin} \\ A \subseteq \text{Addresses}}} (A \rightarrow \text{Values})$

union of all the functions from addresses to values

We also assume:

finite set of Variables

- $\mathbf{nil} \in \text{Atoms}$
- $\text{Stores}_V = V \rightarrow \text{Values}$
- $\text{States}_V = \text{Stores}_V \times \text{Heaps}$

hint: describe
content of registers

hint: describe content of
addressable memory

A **computational state** contain: a store (stack), mapping variables into values, and a heap, mapping addresses into values (and representing mutable structures)

Meaning of the commands by small-step semantics, i.e., by defining a transition relation between *configurations*.

non-terminal: a command-state pair $\langle c, (s, h) \rangle$ with $FV(c) \subseteq \text{dom } s$

memory fault: an inactive address is “used”

terminal: a state (s, h) or **abort**

Command Semantics: Allocation

semantics of e_1

$$\langle v := \mathbf{cons}(e_1, \dots, e_n), (s, h) \rangle$$
$$\rightsquigarrow$$
$$(\underbrace{[s \mid v: \ell]}_{\text{domain specification: stack with the new "binding" variable - value (address)}}, \underbrace{[h \mid \ell: \llbracket e_1 \rrbracket_{\text{exp}} s \mid \dots \mid \ell+n-1: \llbracket e_n \rrbracket_{\text{exp}} s]}_{\text{heap with the allocated value(s)}})$$

domain specification:
stack with the new “binding”
variable - value (address)

heap with the allocated value(s)

Command Semantics: Lookup

When $\llbracket e \rrbracket_{\text{exp}} s \in \text{dom } h$:

$$\langle v := [e], (s, h) \rangle \rightsquigarrow ([s \mid v: h(\llbracket e \rrbracket_{\text{exp}} s)], h)$$

When $\llbracket e \rrbracket_{\text{exp}} s \notin \text{dom } h$:

$$\langle v := [e], (s, h) \rangle \rightsquigarrow \mathbf{abort}.$$

Command Semantics: Mutation

When $\llbracket e \rrbracket_{\text{exp}} s \in \text{dom } h$:

$$\langle [e] := e', (s, h) \rangle \rightsquigarrow (s, [h \mid \llbracket e \rrbracket_{\text{exp}} s : \llbracket e' \rrbracket_{\text{exp}} s])$$

When $\llbracket e \rrbracket_{\text{exp}} s \notin \text{dom } h$:

$$\langle [e] := e', (s, h) \rangle \rightsquigarrow \mathbf{abort}.$$

Command Semantics: Deallocation

When $\llbracket e \rrbracket_{\text{exp}} s \in \text{dom } h$:

$$\langle \mathbf{dispose} \ e, (s, h) \rangle \rightsquigarrow (s, h \upharpoonright (\text{dom } h - \{\llbracket e \rrbracket_{\text{exp}} s\}))$$

When $\llbracket e \rrbracket_{\text{exp}} s \notin \text{dom } h$:

$$\langle \mathbf{dispose} \ e, (s, h) \rangle \rightsquigarrow \mathbf{abort}.$$

domain restriction

An Important Property

If no abort is raised then the execution on an heap and on the same restricted heap are similar, except for the presence of unchanging extra heap cells in the unrestricted execution.

An Important Property, Formally

- $h_0 \perp h_1 \stackrel{\text{def}}{=} h_0$ and h_1 have disjoint domains
- $h = h_0 \cdot h_1 \stackrel{\text{def}}{=} \text{union of } h_0 \text{ and } h_1$

Then, when $h_0 \subseteq h$, we have:

- ★ If $\langle c, (s, h) \rangle \rightsquigarrow^* \text{abort}$, then $\langle c, (s, h_0) \rangle \rightsquigarrow^* \text{abort}$.
- ★ If $\langle c, (s, h) \rangle \rightsquigarrow^* (s', h')$ then $\langle c, (s, h_0) \rangle \rightsquigarrow^* \text{abort}$ or $\langle c, (s, h_0) \rangle \rightsquigarrow^* (s', h'_0)$, where $h'_0 \perp h_1$ and $h' = h'_0 \cdot h_1$.
- If $\langle c, (s, h) \rangle \uparrow$ then either $\langle c, (s, h_0) \rangle \rightsquigarrow^* \text{abort}$ or $\langle c, (s, h_0) \rangle \uparrow$.

Assertions

We introduce new forms of assertions that describe the heap:

$\langle \text{assert} \rangle ::= \dots$	true, false, $\langle \text{assert} \rangle \wedge \langle \text{assert} \rangle, \dots$
emp	empty heap
$\langle \text{exp} \rangle \mapsto \langle \text{exp} \rangle$	singleton heap
$\langle \text{assert} \rangle * \langle \text{assert} \rangle$	separating conjunction
$\langle \text{assert} \rangle \multimap \langle \text{assert} \rangle$	separating implication

Assertions: emp

emp asserts that the heap is empty

$$\llbracket \mathbf{emp} \rrbracket_{\text{asrt}} s \ h \text{ iff } \text{dom } h = \{ \}$$

Assertions: $e \mapsto e'$

$e \mapsto e'$ asserts that the heap contains one cell, at address e with contents e'

$$\llbracket e \mapsto e' \rrbracket_{\text{asrt}} s h$$

$$\iff$$

$$\text{dom } h = \{\llbracket e \rrbracket_{\text{exp}} s\} \text{ and } h(\llbracket e \rrbracket_{\text{exp}} s) = \llbracket e' \rrbracket_{\text{exp}} s$$

Assertions: $p_0 * p_1$

$p_0 * p$ asserts that the heap can be split into two disjoint parts in which p_0 and p_1 hold.

$$\llbracket p_0 * p_1 \rrbracket_{\text{asrt}} s h$$

$$\iff$$

$$\exists h_0, h_1. h_0 \perp h_1 \text{ and } h_0 \cdot h_1 = h \text{ and } \llbracket p_0 \rrbracket_{\text{asrt}} s h_0 \text{ and } \llbracket p_1 \rrbracket_{\text{asrt}} s h_1$$

Assertions: $p_0 \multimap p_1$

$p_0 \multimap p$ asserts that, if the current heap is extended with a disjoint part in which p_0 holds, then p_1 will hold in the extended heap

$$\llbracket p_0 \multimap p_1 \rrbracket_{\text{asrt}} s h$$

$$\iff$$

$$\forall h'. (h' \perp h \text{ and } \llbracket p_0 \rrbracket_{\text{asrt}} s h') \text{ implies } \llbracket p_1 \rrbracket_{\text{asrt}} s (h \cdot h')$$

$p_0 \multimap p_1$ Said Easy

Take the current heap h . If:

- we extend h with a disjoint heap h' ($h \perp h'$)
- the assertion p_0 holds for h

Then p_1 will hold in the extended heap $h \cdot h'$

Assertions: Syntactic Sugar

1) $e \mapsto - \stackrel{\text{def}}{=} \exists x'. e \mapsto x'$ where x' not free in e

2) $e \hookrightarrow e' \stackrel{\text{def}}{=} e \mapsto e' * \mathbf{true}$

exact state of
the heap

3) $e \mapsto e_1, \dots, e_n \stackrel{\text{def}}{=} e \mapsto e_1 * \dots * e + n - 1 \mapsto e_n$

4) $e \hookrightarrow e_1, \dots, e_n \stackrel{\text{def}}{=} e \hookrightarrow e_1 * \dots * e + n - 1 \hookrightarrow e_n$ iff $e \mapsto e_1, \dots, e_n * \mathbf{true}$

heap contains this
and maybe other
stuff

Specifications and Inference Rules

The notion of specification is similar to that of Hoare Logic, with variants for both partial and total correctness

$\langle \text{spec} \rangle ::=$

$\{ \langle \text{assert} \rangle \} \langle \text{comm} \rangle \{ \langle \text{assert} \rangle \}$	partial
$ [\langle \text{assert} \rangle] \langle \text{comm} \rangle [\langle \text{assert} \rangle]$	total

Partial Correctness

satisfied
precondition
implies no memory
fault from c

the specific holds

$\{p\} \ c \ \{q\}$ holds

\Leftrightarrow

$\forall (s, h) \in \text{States}_V. \llbracket p \rrbracket_{\text{asrt}} s \ h \text{ implies } \neg (c, (s, h) \rightsquigarrow^* \text{abort})$

\wedge

$(\forall (s', h') \in \text{States}_V. \ c, (s, h) \rightsquigarrow^* (s', h') \text{ implies } \llbracket q \rrbracket_{\text{asrt}} s' \ h')$

for every
computational
state

for every state
reachable by c the
postcondition holds

Total Correctness

$[p] \ c \ [q]$ holds

\iff

$\forall (s, h) \in \text{States}_V. \llbracket p \rrbracket_{\text{asrt}} s \ h \text{ implies } (\neg (c, (s, h) \rightsquigarrow^* \text{abort}) \text{ and } \neg (c, (s, h) \uparrow))$

\wedge

$(\forall (s', h') \in \text{States}_V. \ c, (s, h) \rightsquigarrow^* (s', h') \text{ implies } \llbracket q \rrbracket_{\text{asrt}} s' \ h')$

Meaning of Specifications

$$\{p\} \ c \ \{q\}$$



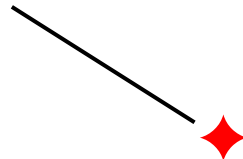
A derivation is found that has $\{p\} \ c \ \{q\}$ as its conclusion.

Meaning of Specifications

$$\{p\} \ c \ \{q\}$$

When p is met and $c \downarrow$, the execution of c “would produce” a set of reachable states that satisfy q .

Then, the post-condition q is an *over approximation* of the the semantics of c “on” p



In our new setting the command-specific rules
of Hoare Logic **remain sound**, e.g.:
consequence rule

$$\frac{p' \Rightarrow p \quad \{p\} c \{q\} \quad q \Rightarrow q'}{\{p'\} c \{q'\}}.$$

still true that if a
derivation is found then ♦

Soundness: One Exception

The “rule of constancy”

$$\frac{\{p\} \ c \ \{q\}}{\{p \wedge r\} \ c \ \{q \wedge r\}}$$

is crucial for scalability: permits to extend a local specification of c with arbitrary predicates about variables not modified by c

Soundness: One Exception

The “rule of constancy” in our setting is not sound:

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

don't preclude aliasing"
if $x = y$ r might be affected

r

The Frame Rule

In place of the rule of constancy: the Frame Rule

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

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

The Frame Rule, Said Easy

With the Frame Rule one can extend a local specification, involving only the variables and parts of the heap that are actually used by c , by adding predicates about variables and parts of the heap that are not touched by c .

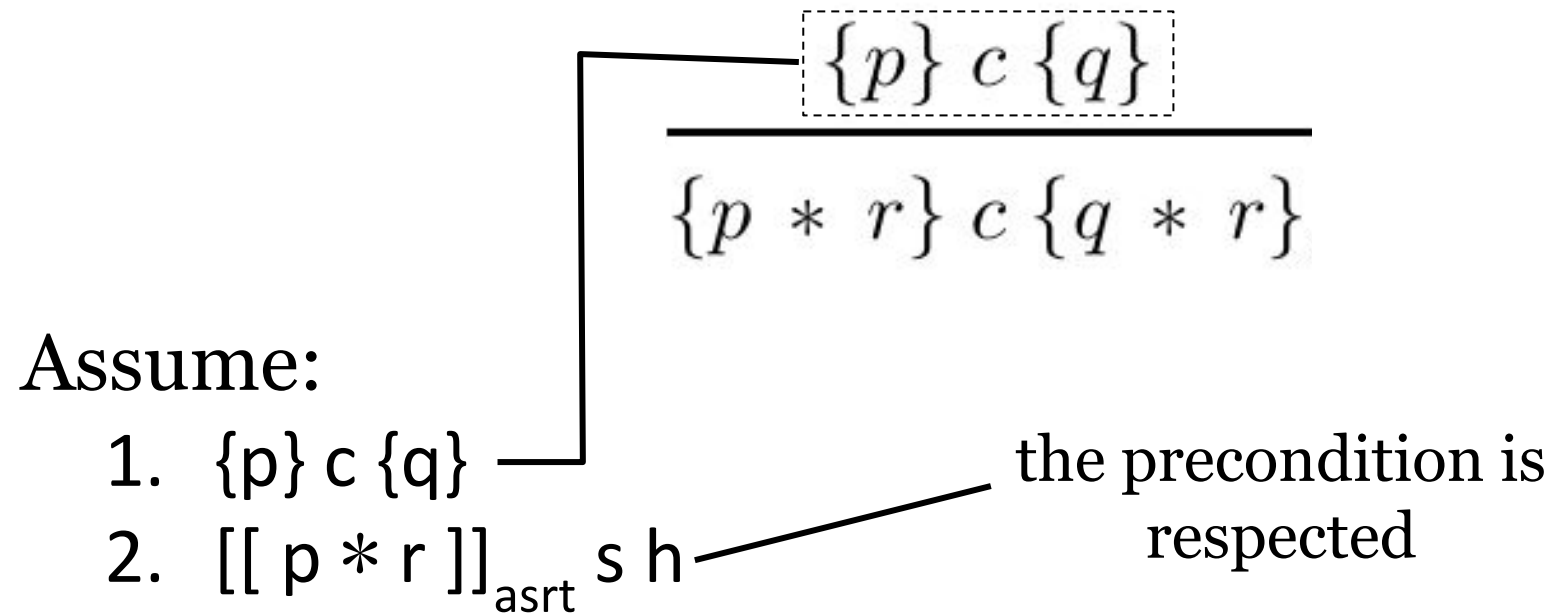
footprint of c

The Frame Rule, Said Easy

The role of the Frame Rule is to infer from a local specification of a command the more global specification appropriate to the larger footprint of an enclosing command.

The Frame Rule: Soundness

We want to show that the rule is sound:



Then, by def. of separate conjunction, there are h_0, h_1 , such that:

I. $h_0 \perp h_1$

II. $h = h_0 \cdot h_1$

III. $[[p]]_{\text{asrt}} s h_0$

IV. $[[r]]_{\text{asrt}} s h_1$

Proof Cases

A. the command c gives abort

$$\langle c, (s, h) \rangle \rightsquigarrow^* \mathbf{abort}$$

B. the command c is executed

$$\langle c, (s, h) \rangle \rightsquigarrow^* (s', h')$$

Recall: An Important Property

- $h_0 \perp h_1 \stackrel{\text{def}}{=} h_0$ and h_1 have disjoint domains
- $h = h_0 \cdot h_1 \stackrel{\text{def}}{=} \text{union of } h_0 \text{ and } h_1$

Then, when $h_0 \subseteq h$, we have:

- ★ If $\langle c, (s, h) \rangle \rightsquigarrow^* \text{abort}$, then $\langle c, (s, h_0) \rangle \rightsquigarrow^* \text{abort}$.
- ★ If $\langle c, (s, h) \rangle \rightsquigarrow^* (s', h')$ then $\langle c, (s, h_0) \rangle \rightsquigarrow^* \text{abort}$ or $\langle c, (s, h_0) \rangle \rightsquigarrow^* (s', h'_0)$, where $h'_0 \perp h_1$ and $h' = h'_0 \cdot h_1$.
- If $\langle c, (s, h) \rangle \uparrow$ then either $\langle c, (s, h_0) \rangle \rightsquigarrow^* \text{abort}$ or $\langle c, (s, h_0) \rangle \uparrow$.

Proof Case: A.

$$\langle c, (s, h) \rangle \rightsquigarrow^* \mathbf{abort}$$

By the property ★ we get that

$$\langle c, (s, h_0) \rangle \rightsquigarrow^* \mathbf{abort}$$

Which contradicts 1. and III.

Proof Case: B.

$$\langle c, (s, h) \rangle \rightsquigarrow^* (s', h')$$

As in the previous case

$$\langle c, (s, h_0) \rangle \rightsquigarrow^* \text{abort}$$

Would contradict 1. and III.

Hence, by the property ★, we get that:

$$\langle c, (s, h_0) \rangle \rightsquigarrow^* (s', h'_0)$$

Proof Case: B.

where:

- $h'_0 \perp h_1$
- $h' = h'_0 \cdot h_1$

then:

$$\{p\} \subset \{q\} \wedge \llbracket p \rrbracket s h_0 \Rightarrow \llbracket q \rrbracket s' h'_0$$

Proof Case: B.


We assumed

$$\langle c, (s, h) \rangle \rightsquigarrow^* (s', h')$$

and we know that s and s' give the same values to the variables that are not modified by c .

The variables of r are not modified by c :

$$(IV.) \llbracket r \rrbracket_{\text{asrt}} s h_1 \Rightarrow \llbracket r \rrbracket_{\text{asrt}} s' h_1.$$

Thus $\llbracket q * r \rrbracket_{\text{asrt}} s' h'$ 

Example

Consider the following “program”:

$[x] := 1;$

$[y] := 2;$

$[z] := 3;$

We want to write a specific to verify that at the end of its execution we will have that

- x points to 1
- y points to 2
- z points to 3

$$\{x \mapsto _ * y \mapsto _ * z \mapsto _\}$$

$$\begin{aligned} &\{x \mapsto _\} \\ &\quad [x] := 1; \\ &\{x \mapsto 1\} \end{aligned}$$

$$\{x \mapsto 1 * y \mapsto _ * z \mapsto _\}$$

$$[y] := 2;$$

$$[z] := 3;$$

$$\{x \mapsto 1 * y \mapsto 2 * z \mapsto 3\}$$

Frame Rule: from a global specific we can consider only the validity of a local (sub) specific

separate conjunction:
built in operator for
sharing prohibition

Fin.