CS 839: Systems Verification

Fall 2025

Today's agenda

1. Recq demo / review
2. Informal proofs
3. Induction

$P(n) \triangleq 1 + 2 \cdots + n = n(n+1)/2$

$P(1) \quad 1 = 1(1+1)/2$

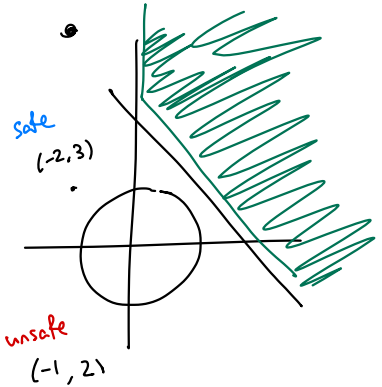
$= 1$

$\forall k, \quad P(k) \rightarrow P(k+1)$

IH: $1 + \cdots + k = k(k+1)/2$

$= \begin{cases} \boxed{1 + \cdots + k} + (k+1) = (k+1)(k+2)/2 \\ k(k+1)/2 + (k+1) = (k+1)(k+2)/2 \end{cases}$

$= \begin{cases} \frac{k(k+1)}{2} + \frac{2(k+1)}{2} \\ \frac{(k+2)(k+1)}{2} = \frac{(k+1)(k+2)}{2} \end{cases}$



$e = \sigma_1, \sigma_2, \sigma_3, \ldots$

$\uparrow$ eg. $(5,3)$

$tr(\sigma, \sigma') = \sigma \searrow \sigma' \vee \sigma \uparrow \sigma' \vee$ noop

$\sigma_1 = (0,5)$

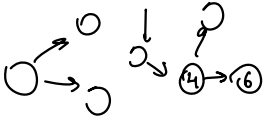
$\forall i,\ tr(e(i), e(i+1))$

$\forall i,\ e(i)$ safe

$\forall \sigma.\ inv(\sigma) \rightarrow P(\sigma)$

$\forall \sigma, \sigma'.\ P(\sigma) \rightarrow tr(\sigma, \sigma') \rightarrow P(\sigma')$

induction for transition systems

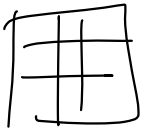


$\forall e,\ valid(e) \rightarrow \forall i,\ P(e(i))$

$\uparrow$ follows inv and tr

Lecture 4: Abstraction

- Most of today will be a fun (?!) activity
- Read the notes



```
location := {
    row: int [0, 2]
    col: int [0, 2]
}

player := black | white

cell := empty | full (p: player)

state := location → cell

game_over (s: state)

move (s, s') : Prop
```

Contents of box: three 1's, two 2's,
two 3's, two 4's,
one 5 of each color

8 "Clock" tokens (hints)

4 "Black Fuse" tokens (lives/misplays)

assume 3 players, hand size of 5 if you like

Lecture 5: Hoare Logic (part 1)

Learning Outcomes:

1. Explain what pre- and post-conditions mean
2. Formally analyze a "whiteboard" programming language

$\{P\}\ e\ \{Q\}$ — if P holds and we run e, and it finishes, then Q } soundness

- semantics: "run e"?
- logic: set of rules for $\{P\}\ e\ \{Q\}$
  - soundness

$\{P\}\ e\ \{\lambda v.\ Q(v)\}$

$e \rightsquigarrow v'$

$Q(v')$

code

recursive calls mod

proof

inductive

proof of euclid "call" proof of mod

euclid(a, b)

mod(a, b)

$\{\_\}$ mod(a, b) $\{c.\ \_\_\_\}$

$\{\_\}$ euclid(a, b) $\{c.\ gcd(a, b, c)\}$

$3 : \text{nat}$

$\overline{3} : \text{val}$

expr $e ::= x \mid v \mid \lambda x.e \mid e_1\ e_2$

$\mid$ if e then e_1 else e_2

$\mid e_1 + e_2$

$\mid (e_1, e_2) \mid \pi_1\ e \mid \pi_2\ e$

values $v ::= \lambda x.\ e \mid \overline{n} \mid \text{true} \mid \text{false} \mid (v_1, v_2)$

let $x := e_1$ in e_2 $::=$ $(\lambda x.\ e_2)\ e_1$

$(\lambda x.\, e)\ v \longrightarrow e[v/x]$ β-reduction $(\lambda x.\ x + \bar{3})\ \bar{5}$

$\bar{5} + \bar{3}$

if false then e_1 else $e_2 \to e_2$

$\pi_1\ (v_1, v_2) \longrightarrow v_1$

$\pi_2\ (v_1, v_2) \longrightarrow v_2$

$\bar{n}_1 + \bar{n}_2 \longrightarrow \overline{(n_1 + n_2)\ \%\ 2^{64}}$

step

$e_1 \to e_2$

$e_1 \to^* e_2$

we have products (tuples)

add sums (inductives / enums)

$A + B$

Either a b

Result $\langle A, B \rangle$

| Ok (a : A)

| Err (b : B)

e ::= .... |

ok e | err e |

match e with

| ok x ⇒ e_1

| err x ⇒ e_2

case (e, ok_f, err_f)

case (ok e, ok_f err_f) ⟶ ok_f e

case (Err e, ok_f, err_f) ⟶ err_f e

$e ::= x \mid v \mid \lambda x.e \mid e_1 + e_2 \mid \cdots$

$v ::= \bar{n} \mid \text{true} \mid \text{false} \mid \cdots$

$e_1 \longrightarrow e_2$ step relation

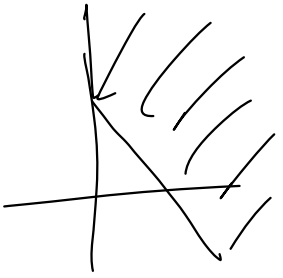
$e_1 \longrightarrow^* e_2$ semantics

$\{P\}\ e\ \{\lambda v.\ Q(v)\}$

if

$(\sigma, pc) \longrightarrow^* (\sigma', pc')$ then

$Q(\sigma')$ $\sigma'(r_6) = 0 + \cdots + 10$



$\forall v;\ P \wedge e \longrightarrow^* v' \Rightarrow$

$Q(v')$

$\{Q(v)\} \; v \; \{v.\, Q(v)\}$

$$\frac{P \vdash Q(v)}{\{P\} \; v \; \{v.\, Q(v)\}}$$

(2 → $P \vdash Q(v)$; 1 → $\{P\}$; 3 → $\{v.\, Q(v)\}$)

$$\frac{\{p\}\; e_1 \;\{v.\, Q(v)\} \qquad \forall v,\; \{Q(v)\}\; e_2[v/x]\; \{R\}}{\{p\}\; \underline{\text{let}}\; x := e_1\; \underline{\text{in}}\; e_2 \;\{R\}}\; \text{hoare-let}$$

Lecture 6: Hoare Logic (part 2)

Learning outcomes:

1. Prove reasoning principles in Hoare Logic
2. Analyze pre- and post-conditions

Hoare logic x2

Separation logic x2

Iris Proof Mode (Rocq)

$\{P\}\ e\ \{v.\ Q(v)\}$

1. $\forall v',\ P \wedge (e \to^* v') \Rightarrow Q(v')$

reasonable?

$\{P\}\ 3\ \{Q\}$

✓

2 ✓ maybe

4 too strong

3 X

5 X

5. $(P \wedge e \to^* 17) \Rightarrow Q(17)$

$P \wedge \forall v'', \quad (\text{let } x := e_1 \text{ in } e_2) \to^* v''$

prove: $R(v'')$

use $\{P\}\ e_1\ \{Q\}$ prove P ✓

$e_1 \to^* v' \Rightarrow$

$Q(v')$ ✓

conclude $R(v'')$

$(\text{let } x := e_1 \text{ in } e_2) \to^*$

$(\text{let } x := v' \text{ in } e_2) \to^*$

$e_2[v'/x] \to^* v''$ ✓

$e_1 \to^* v'$

2. $\{\text{True}\}\ \text{add}\ \bar{n}\ \bar{m}\ \{v.\ v = \overline{n+m}\}$

$\{n < 2^{64} - 1\}$

$3 + \text{true} \not\to .$

$f\ \bar{n}$

$\{\exists p.\ y = \bar{p} \wedge p \leq 1\}$

What did you learn?

What are you confused about?

Lecture 7: Separation Logic (part 1)

- syntax, semantics ← add a heap
- P, Q $\quad \forall x,\ P(x) \quad P \vee Q \quad$ Prop ← heap predicates
- $\{P\}\ e\ \{v.\ Q(v)\}_{SL}$ → soundness
  → rules for constructing proofs

$\text{alloc}\ e : T \quad : \text{ref}\ T \quad \ell : T \quad \ell : \text{loc}$

$!e \quad *e$

$e \leftarrow v \quad *e = v$

$h : \text{loc} \to \text{option val}$

"->" on slides

$(e, h) \rightsquigarrow (e', h')$

!5

Q: val → (heap → Prop)

val → heap → Prop "currying"

Q v h

SL propositions

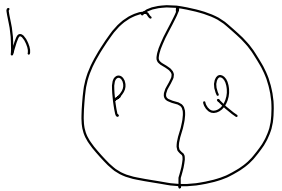
model as heap → Prop

$(\ell \mapsto v)(h) \triangleq h(\ell) = v \wedge dom(h) = \{\ell\}$

$h = \{\ell \mapsto v\}$

$P \vdash Q \triangleq \forall h,\ P(h) \to Q(h)$

$(P * Q)(h) \triangleq \exists h_1, h_2,\ h = h_1 \cup h_2 \wedge \underbrace{h_1 \perp h_2}_{\text{disjoint}} \wedge P(h_1) \wedge Q(h_2)$



$emp(h) \triangleq dom(h) = \emptyset$

$True(h) \equiv True$

$(P * Q)(h)$ $\qquad$ $\forall h_1,\ P(h) \to P'(h)$

---

goal: $(P' * Q)(h)$

$h = h_1 \cup h_2$

$P(h_1)$ $\qquad$ $Q(h_2)$

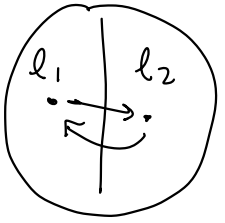
$\downarrow$

$P'(h_1)$

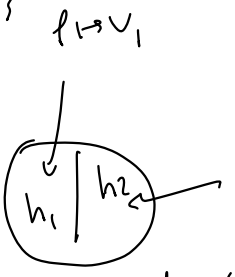
$P * Q \vdash Q * P$ $\qquad$ sep-comm

$\vdash Q * P'$ $\qquad$ sep-monotone-right

$\vdash P' * Q$ $\qquad$ sep-comm



$\{ l_1 \mapsto l_2 ; \quad l_2 \mapsto l_1 \}$



$dom(h_1) = \{l\}$

$dom(h_2) = \{l\}$

$h_1 \perp h_2$

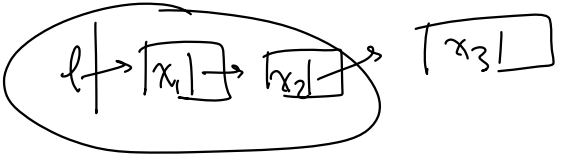
$l \mapsto v_1 * l \mapsto v_2 \vdash \text{False}$

$l_1 \mapsto v_1 * l_2 \mapsto v_2 \vdash l_1 \neq l_2$

$\text{list}(l_1, xs) * \text{list}(l_2, ys)$

↑

list int

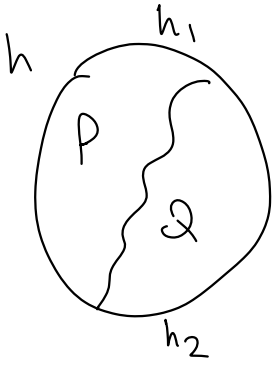


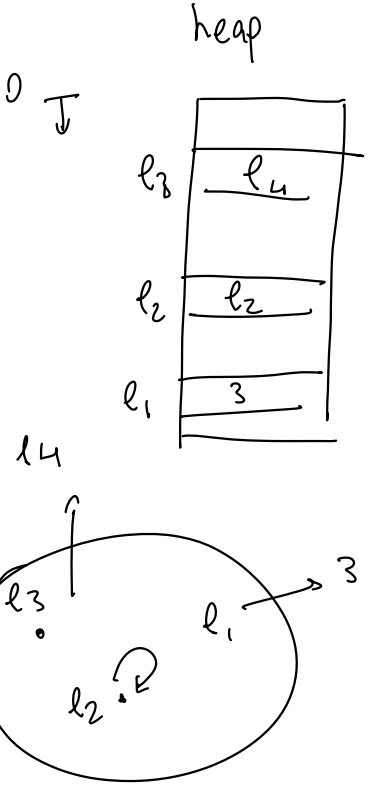
Lecture 8

discriminate missing

caution on using induction tactic

Strategy for even proof is not obvious





$\{P\}\ e\ \{v.\ Q(v)\}$

$\varphi : Prop$

$\lceil \varphi \rceil : hProp$

$w = \#\ell$

$\uparrow$ val

$\nwarrow$ loc

$\lceil \varphi \rceil\ (h) \triangleq \varphi \wedge h = \{\}$

$emp \vdash \lceil True \rceil$

$P, Q : hProp$

$(P \wedge Q)\ (h) \triangleq P(h) \wedge Q(h)$

$\lceil \varphi \rceil * P \vdash P \wedge \lceil \varphi \rceil$

$$\frac{P' \vdash P \quad Q \vdash Q' \quad \{P\}\ e\ \{Q\}}{\{P'\}\ e\ \{Q'\}}$$

$$\frac{\{emp\}\ \text{alloc } 42\ \{y \mapsto 42\}}{\{x \mapsto 0\}\ \text{alloc } 42\ \{x \mapsto 0 * y \mapsto 42\}}$$

RET #();

$\{l_1 \mapsto 0\}\ f(l_1, l_2)\ \{l_1 \mapsto 42\}$

$\{emp\}$ assert #true $\{emp\}$

let $t := !l_1$ in
let $t_2 := !l_2$ in
$l_1 \leftarrow t_2$;
$l_2 \leftarrow t$

$\{emp\}$
let $x :=$ alloc 0 in
$\{x \mapsto 0\}$
let $y :=$ alloc 42 in
$\{x \mapsto 0 * y \mapsto 42\}$
$f(x, y)$
$\{x \mapsto 42 * y \mapsto 42\}$
let $a := !x$ in
$\{\lceil a = 42 \rceil * x \mapsto 42 * y \mapsto 42\}$
let $b := !y$ in
$\{\lceil b = 42 \rceil * x \mapsto 42 * y \mapsto 42\}$
assert #(bool_decide $(a = b)$)
$\{x \mapsto 42 * y \mapsto 42\}$
$\{True\}$

$\{\ell_1 \mapsto a \ast \ell_2 \mapsto b\}$

let $t := !\ell_1$ in

$\{\ulcorner t = a \urcorner \ast \ell_1 \mapsto a \ast \ell_2 \mapsto b\}$

let $t_2 := !\ell_2$ in

$\{\ulcorner t_2 = b \urcorner \ast \ell_2 \mapsto b \ast \ell_1 \mapsto a\}$

$\ell_1 \leftarrow t_2$;

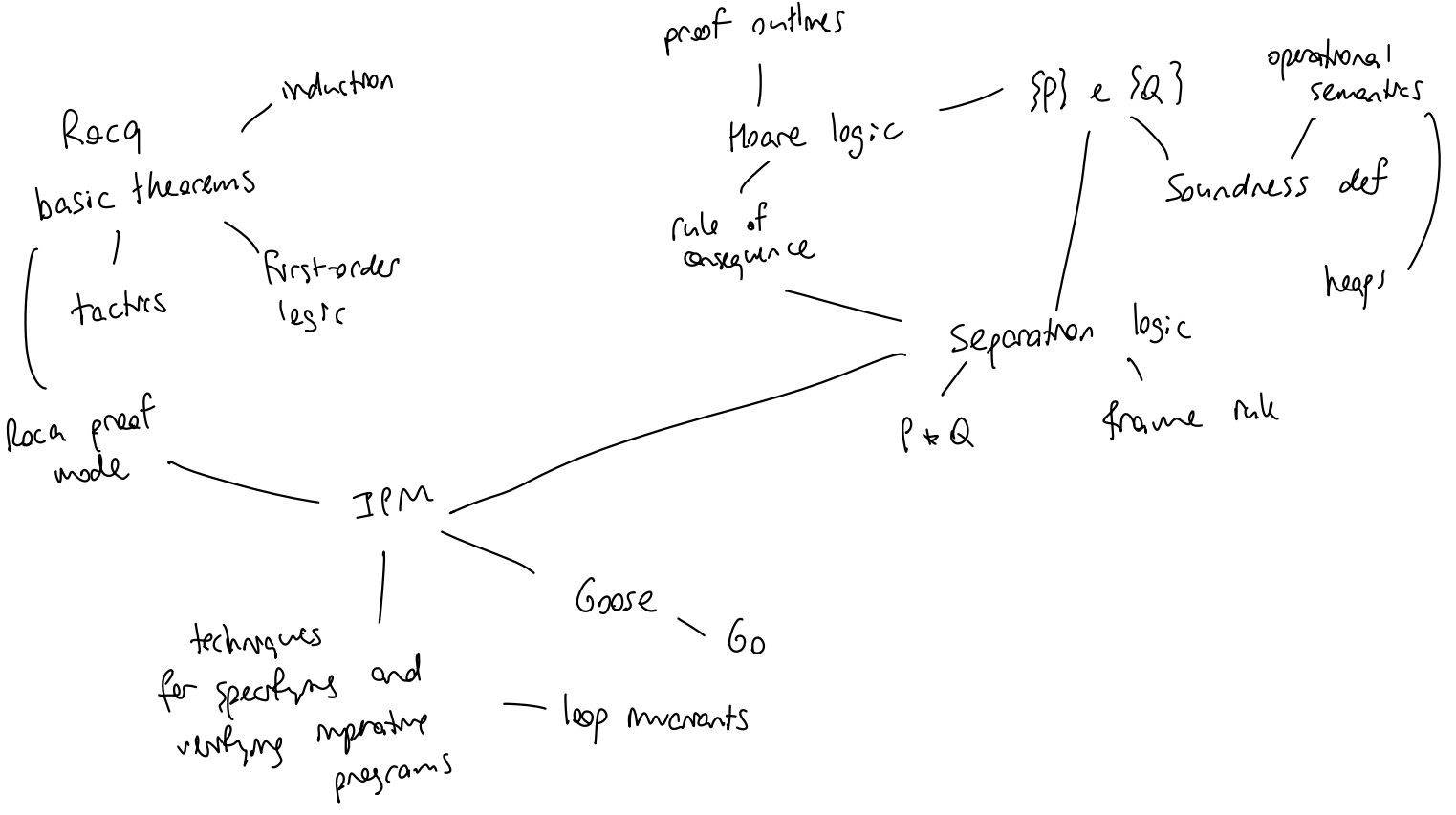
$\{\ell_1 \mapsto t_2 \ast \ell_2 \mapsto b\}$

$\ell_2 \leftarrow t$

$\{\ell_2 \mapsto \cancel{t}^{\,a} \ast \ell_1 \mapsto \cancel{t_2}^{\,b}\}$

$\{\ell_1 \mapsto b \ast \ell_2 \mapsto a\}$

Video Lecture: Recursion


induction
Rocq
basic theorems
tactics
first-order logic
Rocq proof mode
proof outlines
Hoare logic
rule of consequence
{P} e {Q}
operational semantics
Soundness def
heaps
Separation logic
P * Q
frame rule
IPM
Goose
Go
techniques for specifying and verifying imperative programs
loop invariants

$P * Q$ $\quad \exists x.\ P(x)$ $\quad \lceil \varphi \rceil$ $\quad emp$

$\forall x.\ P(x)$

intro

$P -\!\!* Q$ $\quad P \vdash (Q -\!\!* R) \iff P * Q \vdash R$

$P * (P -\!\!* Q) \vdash Q$ $\quad elim$

$P \wedge (P \to Q) \vdash Q$

$$\lceil t = a \rceil * (x \mapsto b * y \mapsto t) \vdash x \mapsto b * y \mapsto a$$

$$P \vdash \lceil \varphi \rceil \qquad \lceil \varphi \rceil \to (P \vdash Q) \to P * \lceil \varphi \rceil \vdash Q$$

$P, Q : \mathrm{iProp}$

$$P * Q \vdash Q * P \qquad P \vdash \mathrm{emp}$$

"H_1" : P

~~"H_2" : Q~~

~~Q~~ $*$ R

iIntros "[H1 H2]" "(H1 & H2 & H3)"

iDestruct "H" as "[H1 H2]"

(x) "H2"

$A \to B \to C$

$A \wedge B \to C$

iExists

iSplitL, iSplitR

iAssumption, iFrame

iApply

(H with "[H]")

with "[$H1 H2]"

Specialization patterns

with "[H1] [H2]" with "[$]"

Learning outcomes

1. Translate program proof goals from IPM to paper
2. Use IPM for entailments / WPs

$e ::= x \mid v \mid \cancel{\lambda x.\, e} \mid e_1\ e_2$

$(\text{rec}\ f\ x.\ e)$

↑ name ↑ name (argument)

$\lambda x.\ e \triangleq (\text{rec}\ \_\ x.\ e)$

Sum ::

$(\text{rec}\ f\ n.\ \text{if}\ n = 0\ \text{then}\ 0\ \text{else}\ n + f\ (n-1))$

$$fib\ n = fib\ (n-2) + fib\ (n-1)$$

$$fib\ 0 = 0$$
$$fib\ 1 = 1$$

$$(rec\ f\ x.\ e)\ v \longrightarrow e\,[(rec\ f\ x.\ e)/f]\,[v/x]$$

$$SumN := (rec\ f\ n.\ \text{if } n=0 \text{ then } 0 \text{ else } n + f\ (n-1))$$

$$SumN\ 3 \longrightarrow \text{if } 3=0 \text{ then } 0 \text{ else } 3 + \overbrace{(rec\ f\ n.\ \text{if } n=0 \text{ then } 0 \text{ else } n + f\ (n-1))}^{=SumN}\ (3-1)$$

$\{P\}\ e\ \{Q\}$

if P, then if

$e \rightsquigarrow v'$,

then $Q(v')$

if,

$e \rightsquigarrow e'$

then either e' is not stuck

or e' is a value v'

and $Q(v')$

$\{P\} \; e \; \{Q\}$

$P * Q \vdash Q * P$

```
P, Q : iProp
--------------------------------------
"H1" : P
"H2" : Q
--------------------------------------∗
Q * P
```

Htmp

```
iDestruct (lem with "[H1 H2]") as "[H3 H4]".
```

```
lem : A -∗ B
```

```
iPoseProof (lem with "[H1 H2]")
  as Htmp.
```

goal 1:

```
H1 : _
H2 : _
--------------------------------------∗
A
```

goal 2:

```
Htmp : B
--------------------------------------
goal
```

```
H3 : B1
H4 : B2
```

$\{P\}\ e\ \{Q\} \iff P \vdash wp(e, Q)$

$wp(e, Q) : iProp$

$\uparrow \quad \uparrow$

$expr \quad val \to iProp$

$\{ wp(e, Q) \}\ e\ \{Q\}$

assert (!x == 0);

x ← 0

$\{ \ell \mapsto 0 \}$

IgnoreOne $\ell\ \ell'$

$\{ \ell \mapsto 42 \}$

$wp(e_1, wp(e_2, \Phi)) \vdash wp((e_1; e_2), \Phi)$

$$\frac{\{P\}\ e_1\ \{Q\} \qquad \{Q\}\ e_2\ \{R\}}{\{P\}\ e_1; e_2\ \{R\}}$$

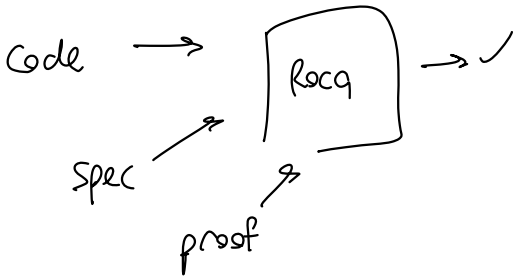
$$\{P\}\ e\ \{Q\} \triangleq P \vdash wp(e, Q)$$

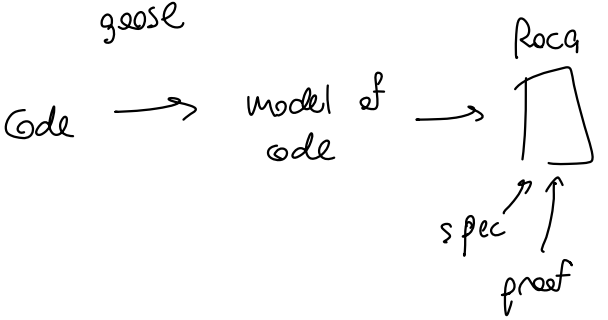
$$\forall \phi,\ P * \underbrace{(\forall v,\ Q(v) \twoheadrightarrow \phi(v))}_{H\phi} \vdash wp(e, \phi)$$

Lecture 11 : Goose

How to model Go code for verification?

Today's focus: control flow





Go

```
for {
    e
}
```

↓

```
for e
```

1. Determine Go evaluation order in $f(x, y)$

2. Determine Goose evaluation order

$f(\text{collatz}(2000), \text{collatz}(3000), \text{collatz}(4000))$

1 (do: e)

2 (return: e)

3 $m_1 ;;; m_2$

(do: e) ≜ ("execute", e)

(return: e) ≜ ("return", e)

do: $e_1 ;;; m_2 \equiv e_1 ;;; m_2$

return: $e_1 ;;; m_2 \equiv$ return: e_1

4 exception_do (return: e) ≡ e

exception_do (do: e) ≡ e

break: #() ≜ ("break", #())

continue: #() ≜ ("continue", #())

exception_do m ≡ Snd m

Lecture 12: Ownership reasoning

1. Articulate the meaning of $\ell \mapsto v$ as ownership
2. Work with fractional permissions
3. Work with struct ownership

$$\ell \overset{q}{\mapsto} v \;\dashv\vdash\; \ell \overset{q_1}{\mapsto} v \;*\; \ell \overset{q_2}{\mapsto} v$$

$$\ell \overset{q}{\mapsto} v \;\dashv\vdash\; \ell \overset{q_1}{\mapsto} v \;*\; \ell \overset{q_2}{\mapsto} v \;*\; \ulcorner q_1 + q_2 \leq 1 \urcorner$$

$$\ell \overset{q_1}{\mapsto} v_1 \;*\; \ell \overset{q_2}{\mapsto} v_2 \;\vdash\; \ulcorner v_1 = v_2 \urcorner$$

$$\{\ell \overset{1/2}{\mapsto} v_0\}\ \ell \leftarrow v\ \{\ell \overset{1/2}{\mapsto} v\}$$

what goes wrong?

$$\forall q,\ \{\ell \overset{q}{\mapsto} v\}\ !\ell\ \{\mathrm{RET}\ v;\ \ell \overset{q}{\mapsto} v\}$$

$$\{\ell \overset{1}{\mapsto} v_0\}\ \ell \leftarrow v\ \{\ell \overset{1}{\mapsto} v\}$$

$$\ell \mapsto \left(\ell \overset{1}{\mapsto} v_0 * \ell \overset{q}{\mapsto} v\right) \vdash \text{False}$$

$$\{\text{True}\}\ \ell := \text{alloc } 2\ \{\ell \mapsto 2\}$$

$$\{\ell \overset{1/2}{\mapsto} 2 * \boxed{\ell \overset{1/2}{\mapsto} 2}\}\ \ell \leftarrow 3$$

$$\{\ell \overset{1/2}{\mapsto} 3 * \ell \overset{1/2}{\mapsto} 2\}\ \{\ulcorner 3 = 2 \urcorner\}\ \{\text{False}\}$$

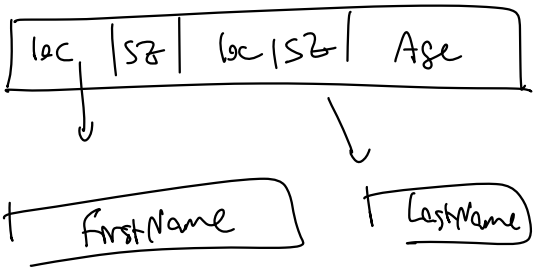
$$\{\ell \overset{1}{\mapsto} v_0 * F\}\ \ell \leftarrow v\ \{\ell \overset{1}{\mapsto} v * F\}$$

$\ell \mapsto p$ $\quad \{\ell := p\}$

RET a_ℓ.

$\{p\_\ell \mapsto p\}$ GetAge p_ℓ $\{\exists a\_\ell.\ a\_\ell \mapsto (p.\text{Age})\}$

$\{\ell := p.\text{FirstName},\ \ell+1 := p.\text{LastName},\ \ell+2 := p.\text{Age}\}$



Lecture 13: Ownership (part 2)

1. Review ownership of structs
2. Use ownership of slices correctly

$\ell \mapsto v$

$\ell \mapsto_S [S :: f]\ v$ Struct field points-to

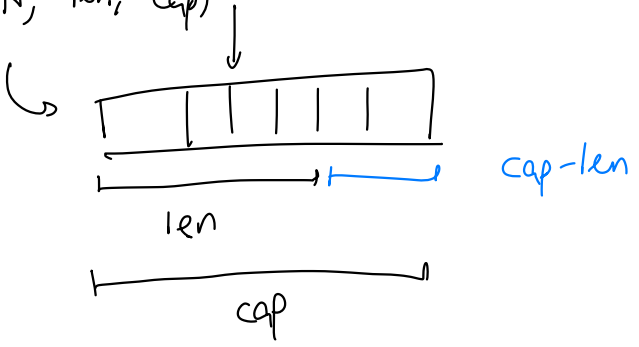
just syntax

s []int

s ↦* vs

(vs : list w64)

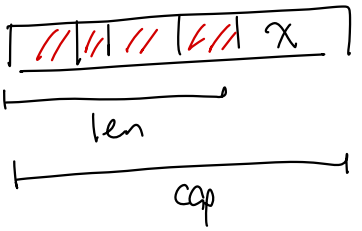
s = (ptr, len, cap)



- get an element reference
- load
- store

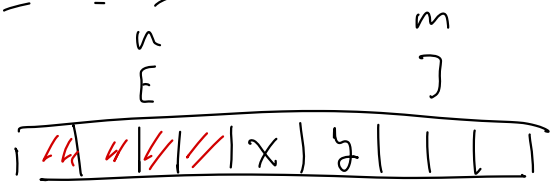
- subslice
- append

append (s, x)



s = (ptr, len, cap)

append (s, x) = (ptr, len+1, cap)



append (append (s, x), y)

(ptr', len+1, cap×2)

$$s[n:m] = (ptr + n,\ m - n,\ cap - n)$$

includes elements from n to end (and original spare capacity in s)

Lecture 14: Loop invariants

1. Recall the rules for a correct loop invariant
2. Struggle to come up with loop invariants

own_slice s xs

s: slice.t

xs: list V

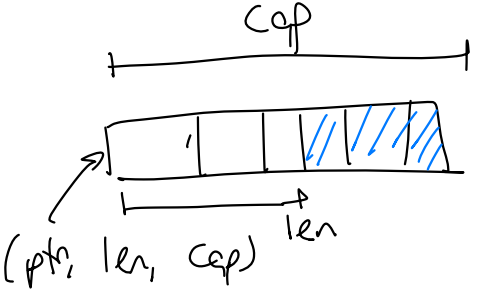
own_slice_cap s

$s \mapsto^* xs$

own_slice_len

$x_i : V$

$\#x : val$



[list_rep l (x :: xs)]

tail l

[∀ v. list_rep v xs]

[list_rep l (x :: xs)]

[list_rep l (x :: xs) ∧ tail_tl list_rep v xs]

[∃ v. l ↦ x * list_rep v xs]

len cap-len

$\{ s \mapsto\!* \; xs \;*\; \text{own\_slice\_cap } s \}$

append(s, x)

$\{ s',\ \text{RET } \#s';\ s' \mapsto\!* \; (xs ++ [x]) \;*\; \text{own\_slice\_cap } s' \}$

len+1 cap-len-1

$s \mapsto\!* xs \;*\; \text{own\_cap}(s) \vdash s[0:n] \mapsto\!* \text{take}(xs, n) \;*\; \text{own\_cap}(s[0:n])$

$s \mapsto\!* xs \vdash s[0:n] \mapsto\!* \text{take}(xs, n) \;*\; s[n:\text{len}(s)] \mapsto\!* \text{drop}(xs, n) \;*\; \text{own\_cap}(s) \;*\; \text{own\_cap}(s[n:\text{len}(s)])$

$s := []int\{1,2,3\}$

$s \mapsto * [1;2;3] * own\_cap(s)$

$s_1 := s[:1]$

$s_2 := s[1:]$

$\{ s_1 \mapsto * [1] * s_2 \mapsto * [2;3] * own\_cap(s_2) \}$

$s2[0]$ ✓

$s_1 = append(s_1, s)$

$\{True\}\ s[:n]\ \{RET\ s[:n];\ True\}$

$s[:1:1]$

$\forall I,$

$$\frac{\{I\}\ \text{body}\ ()\ \{I\}}{\{I\}\ \text{loop body}\ \{I\}}$$

loop body

$\{P\}$ loop body $\{Q\}$

$$P \vdash I$$
$$I \vdash Q$$

$$\sum_{0 \le i \le n} i = \frac{n(n+1)}{2}$$

```
sum := 0
i := 0
if i > n
    sum += i
    i++
```

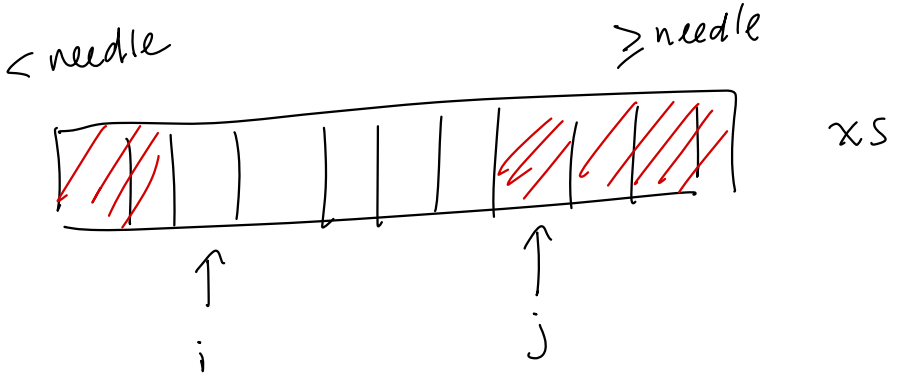
prove I (manually)

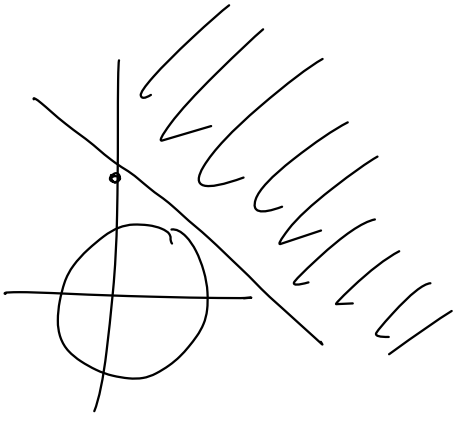
return sum

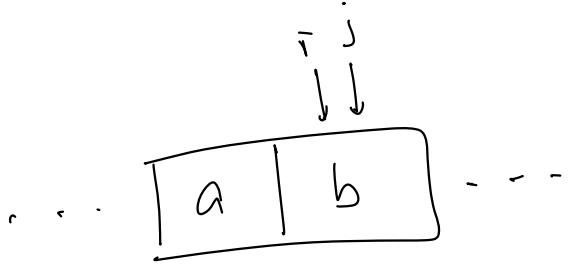
prove postcondition

$\{I\}$

Lecture 15







$a < needle < b$

$\{ l_1 \mapsto l_1' * \text{list\_rep } l_1' \; xs \}$

match !l_1 with

| Inil ⇒ $\{\text{list\_rep Inil } xs\}$

// why does $xs = []$?

list_rep v xs :=

match xs with

| [] ⇒ v = Inil

| hd :: xs' ⇒ ... v = Icons hd tl ...

end

$\{\text{list\_rep } l_2 \; ys\}$

$\{\text{List\_rep } l_2 \; (xs ++ ys)\}$

$\text{list\_rep Inil } xs \vdash \lceil xs = [] \rceil$

Lecture 16: Persistently modality

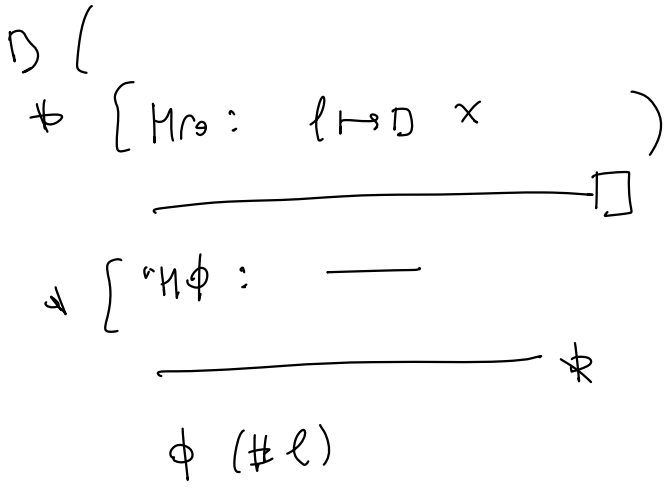
$P : iProp$ $\quad \ell \mapsto v$

Persistent (P) $\quad \square P$ "persistently P"

$\square P : iProp$

$\ell \mapsto_q v$ $\quad q \leq 1$: read-only $\quad 0 < q$

$\ell \mapsto_{q_1+q_2} v \dashv\vdash \ell \mapsto_{q_1} v * \ell \mapsto_{q_2} v$



$\square\, H_{r0} * H\phi \vdash \phi(\#\ell)$

(h_r, h_w) $\quad h_r \perp h_w \quad h = h_r \cup h_w$

$(\ell \mapsto v)\ (h_r, h_w) \triangleq h_w = \{\ell := v\}$ $\quad (\ell \mapsto_\square v)(h_r, h_w) \triangleq h_r = \{\ell := v\}$

$\square P\ (h_r, h_w) \triangleq P(h_r, \emptyset)$

$P * Q\ (h_r, h_w) \triangleq$

$\exists h_1, h_2,\ h_w = h_1 \cup h_2$

$h_1 \perp h_2$

$P(h_r, h_1) \wedge$

$Q(h_r, h_2)$

1. $\square P \vdash P$
2. $P \vdash \square P$
3. $P \vdash Q$ implies $\square P \vdash \square Q$
4. $\square P * Q \vdash \square P \wedge Q$
5. $\square P \wedge Q \vdash \square P * Q$
6. what is $\square(\ell \mapsto v)$?

$\lceil \varphi \rceil * Q \dashv\vdash \lceil \varphi \rceil \wedge Q$

$\square P * Q \dashv\vdash \square P \wedge Q$

~~$A * B \dashv\vdash A \wedge B$~~

Persistent $(P) \quad := \quad P \vdash \Box P$

$P \vdash \Box Q \quad \rightarrow \quad P \vdash \Box Q * P$

$P \overset{?}{\vdash} \Box Q \wedge P \overset{\checkmark}{\vdash} \Box Q * P$

$P \vdash P$ ✓

$P \vdash \Box Q$ ✓

$\square P \vdash P * \square P$

$\{P\}\ e\ \{Q\} : \mathrm{Prop}$

$\forall \Phi,\quad P * (\forall v, Q(v) -\!\!* \Phi(v)) \vdash \mathrm{WP}(e, \Phi)$

$P \vdash \mathrm{WP}(e, Q)$

$\{P\}\ e\ \{Q\} : \mathrm{iProp}$

$(P -\!\!* Q) * P \vdash Q$

$P -\!\!* \mathrm{WP}(e, Q)$

$$\square\left(\forall x,\ \text{true} \twoheadrightarrow \text{wp}\ f\_\text{code}\ x\ \{\lambda y.\ \ulcorner y = f\ x \urcorner\}\right)$$

$$\ell \mapsto_{p} v \triangleq \left(\exists q,\ \ell \mapsto_{q} v\right) \qquad \ell \mapsto_{p} v \vdash \ell \mapsto_{p/2} v * \ell \mapsto_{p/2} v$$

Lecture 17: Concurrency introduction

Learning outcomes:

1. Formalize concurrency semantics with interleavings
2. Appreciate why concurrency is challenging

Lecture 18: Lock invariants

$\{P\}\ e\ \{Q\}$ if $(e, h) \rightsquigarrow (e', h')$

either (a) (e', h') is not stuck

(b) $\exists v'.\ e' = v' \wedge \varphi(v')$

$\{\text{True}\}\ e\ \{v.\ \lceil \varphi(v) \rceil\}$

$\varphi : \text{val} \to \text{Prop}$

$P(h)$

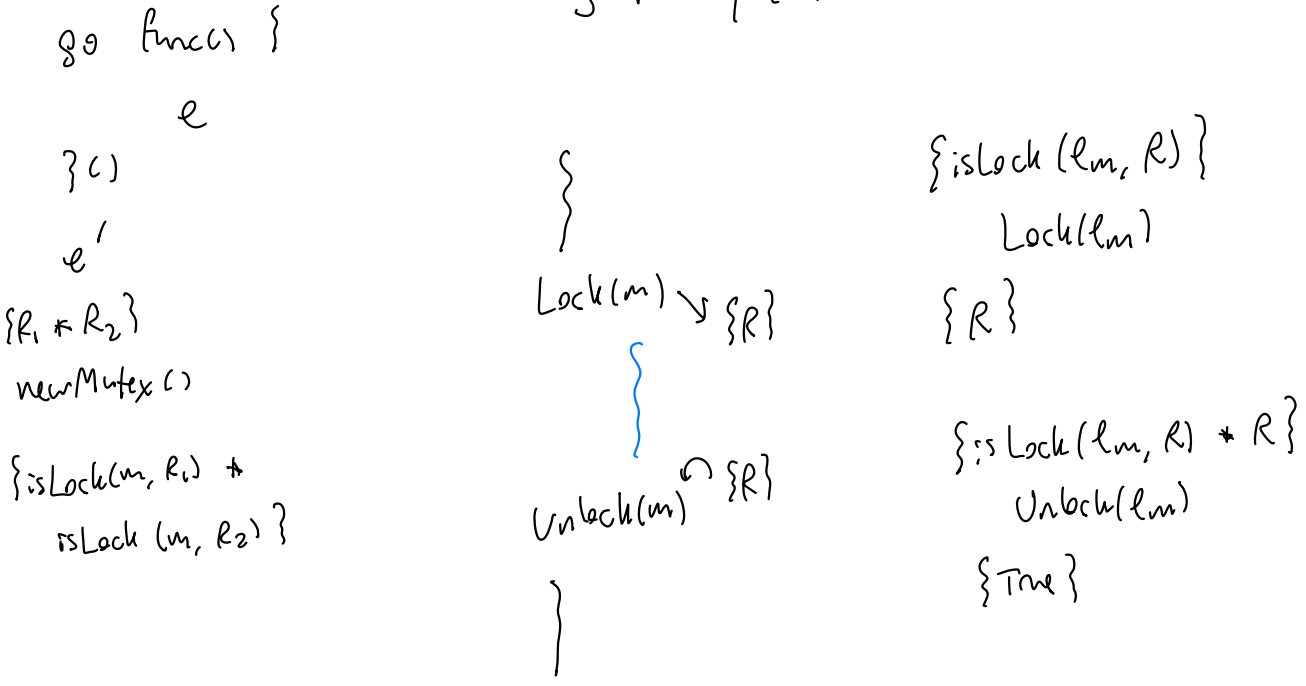
if $([e], h) \rightsquigarrow_{tp} ([e'] ++ T', h')$

either (a) $([e'] ++ T', h)$ is not stuck

(b) $\exists v'.\ e' = v' \wedge \varphi(v')$ ←

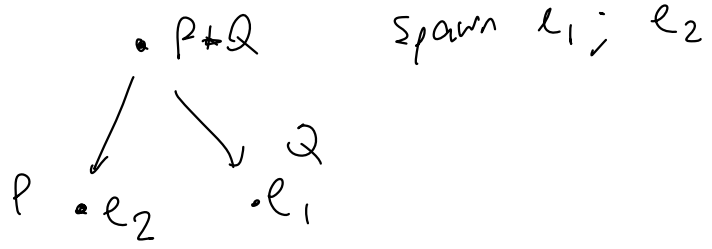
and no thread in $[e'] ++ T'$ is stuck in h

same thread can take a step


S ⊢ wp (e, True) * P
go func() {
e
}()
e'
{R1 * R2}
newMutex()
{isLock(m, R1) *
isLock(m, R2)}
{
Lock(m)
{R}
Unlock(m)
{R}
}
{isLock(lm, R)}
Lock(lm)
{R}
{isLock(lm, R) * R}
Unlock(lm)
{True}

Lecture 19: Ghost state

1. Explain the API for (plain) ghost variables
2. Use ghost variables in simple concurrency proofs.
3. (maybe) Appreciate resource algebras as a foundation for ghost state.



Spawn(...) var x = 0

FAA(&x, 2) || FAA(&x, 2)

{Q1} {Q2}

Q1 * Q2

print(x) // 4

var x = 0

Lock() || Lock()

x += 2 || x += 2

Unlock() || Unlock()

Lock(); !x; Unlock()

x += 2

{ WP f () {P} }

t := Spawn(f)

{ is_JoinHandle(t, P) }

{ is_JoinHandle(t, P) }

t.Join()

{P}

$i = 0$

$x_1 = 0$

$x_2 = 0$

$\{x_1\text{-ptr} \overset{2/3}{\mapsto} 0\}$

Lock()

$i += 2$

$x_1 = 2$

Unlock()

$\{x_1\text{-ptr} \overset{2/3}{\mapsto} 2\}$

is_Mutex($i\text{-ptr} \mapsto i$ * $x_1\text{-ptr} \overset{1/3}{\mapsto} x_1$ * $x_2\text{-ptr} \overset{1/3}{\mapsto} x_2$ * $\lceil i = x_1 + x_2 \rceil$)

$\{x_2\text{-ptr} \overset{2/3}{\mapsto} 0\}$

Lock()

$i += 2$

$x_2 = 2$

Unlock()

$\{x_2\text{-ptr} \overset{2/3}{\mapsto} 2\}$

$x\text{-ptr} \mapsto v \dashv\vdash$

split/combine $x\text{-ptr} \overset{1/2}{\mapsto} v * x\text{-ptr} \overset{1/2}{\mapsto} v$

agree $x\text{-ptr} \overset{q_1}{\mapsto} v_1 * x\text{-ptr} \overset{q_2}{\mapsto} v_2 \vdash \lceil v_1 = v_2 \rceil$

$$\frac{H:\ i = x_1 + x_2}{(i+2) = 2 + x_2}$$

Lock()

$y := i$

Unlock()

y ?

$\gamma \overset{1/2}{\mapsto} v_1 * \gamma \overset{1/2}{\mapsto} v_1 \equiv\!* \gamma \overset{1/2}{\mapsto} v_2 * \gamma \overset{1/2}{\mapsto} v_2$

ghost_var(γ, 1, v_1)

$\text{True} \equiv\!* \exists \gamma,\ \gamma \overset{1/2}{\mapsto} v * \gamma \overset{1/2}{\mapsto} v$

$P \equiv\!* Q \quad \sim \quad \{P\}\ \text{skip}\ \{Q\}$

deep

Lecture 20: Atomic specs

Learning outcomes

1. Appreciate the challenge of specifying atomicity
2. Recall how to specify atomicity using atomic updates

$\exists x,\ \gamma\_ptr \mapsto x * \ldots$

var m sync.Mutex

var x int

$\gamma_1 = 0$

$x_2 = 0$

is_Mutex(_____)

$\exists x_1,\ \gamma_1 \overset{1/2}{\mapsto} x_1 *$

$\exists x_2\ \ \gamma_2 \overset{1/2}{\mapsto} x_2 *$

$x = x_1 + x_2$

Spawn($(\gamma_1 \overset{1/2}{\mapsto} 0)$

m.Lock()

x += 2 ← $x_1 = 2$

m.Unlock()

)

$(\gamma_2 \overset{1/2}{\mapsto} 2)$

Spawn($(\gamma_2 \overset{1/2}{\mapsto} 0)$

m.Lock()

x += 2 ← $x_2 = 2$

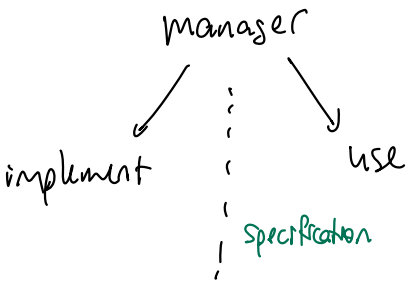
m.Unlock()

$(\gamma_2 \overset{1/2}{\mapsto} 2)$

m.Lock(); y := x; m.Unlock()

Assert(y = 4)

↖ why = 4?



integer

$\ell.\mathrm{Get}() : \mathrm{nt}$

$\ell.\mathrm{Inc}()$

$\ell \mapsto x$

$\{ \mathrm{int\_rep}(\ell, x) \}$

$\ell.\mathrm{Get}()$

$\{ \mathrm{RET}\ \#x;\ \ \mathrm{int\_rep}(\ell, x) \}$

$\{ \mathrm{int\_rep}(\ell, x) \}$

$\ell.\mathrm{Inc}()$

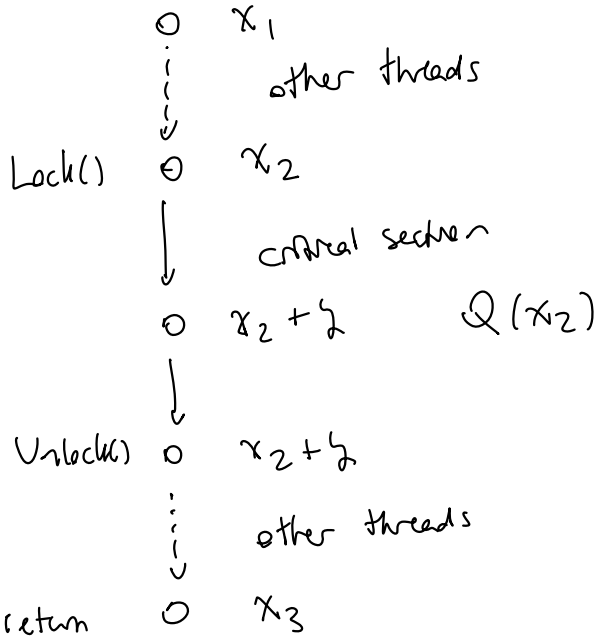
$\{ \mathrm{RET}\ \#();\ \ \mathrm{int\_rep}(\ell, x+1) \}$

is_atomic_mt(ℓ, γ) : iProp

own_mt(γ, x) : iProp

inv I

$I \twoheadrightarrow \exists x.\ own\_mt(x) * (own\_mt(x+2) \twoheadrightarrow I)$



$\exists x.$ $own\_mt(x)$

$own\_mt(x+2)$

$\text{True} \overset{\top}{\Rrightarrow}_{\emptyset}$

$\{ \overset{\top}{\Rrightarrow}_{\emptyset} \exists x,\ own\_mt(\gamma, x) * (own\_mt(\gamma, x) \overset{\emptyset}{\Rrightarrow}_{\top} Q(x)) \}$

FAA
- movement by 1
- returns old value

$\ell.\text{Get}()$

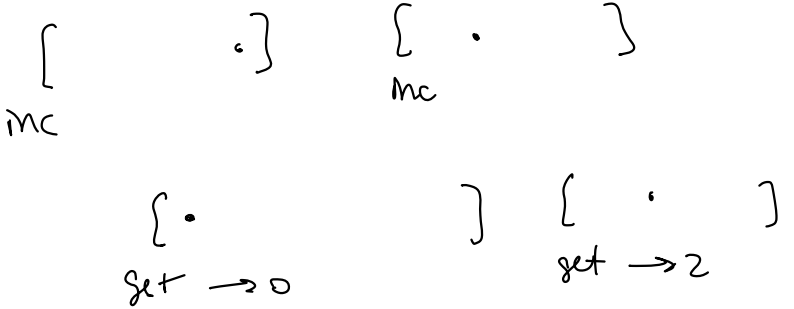
$AU\langle \exists x. own\_mt(\gamma, x) ;\ \text{RET } x.\ own\_mt(\gamma, x) \rangle$

$\{ \exists x.\ \text{RET } \#x,\ Q(x) \}$

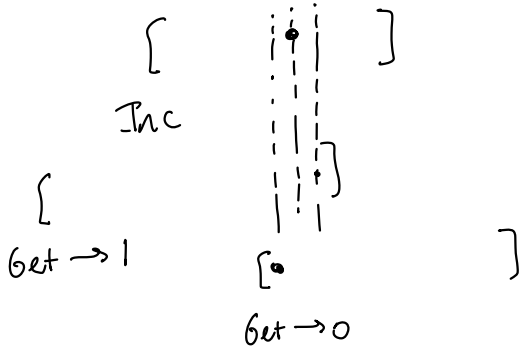
$inv\ I \vdash \text{True} \overset{\top}{\Rrightarrow}_{\emptyset} I$

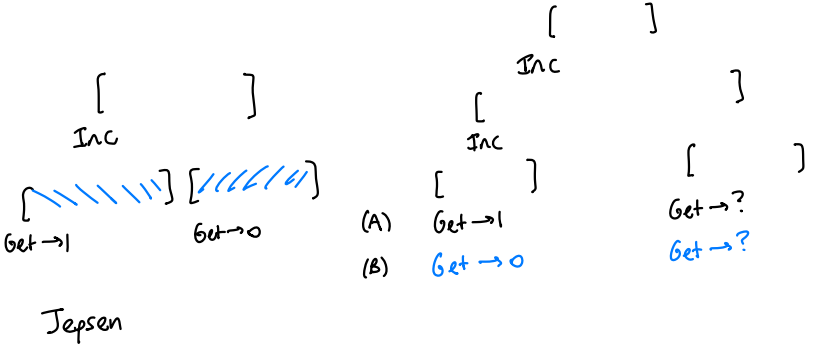
$A \overset{\top}{\Rrightarrow}_{\emptyset} B$

$B \overset{\emptyset}{\Rrightarrow}_{\top} C$


inc
inc
set → 0
set → 2

Lecture 21: Atomic specs (part 2)





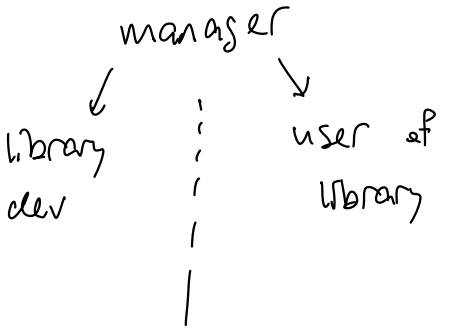
linearizability: $\forall tr$, behavior(code, tr) $\rightarrow$ linearizable(tr)

$\exists$ ordering

$\exists$ ordering(code), $\forall tr$, behavior(code, tr) $\rightarrow$ linearizable(tr)

ordering(code)

1. dynamic lin points
2. helping
3. "future dependent"



$\boxed{I} : iProp$

inv N I

$\forall \phi, \{ \phi \}$ Noop $\{ \phi \}$

Exercise solution (FAA spec)

$\{ \top \Rrightarrow \emptyset \; \exists x.\ own\_mt(\gamma, x) * (own\_mt(\gamma, x+1) \Rrightarrow \emptyset \; \phi(x)) \}$

$\#$

$l.FAA()$

$\{ \exists x.\ RET\ \#x, \phi(x) \}$

ϕ

$\ell \mapsto v$

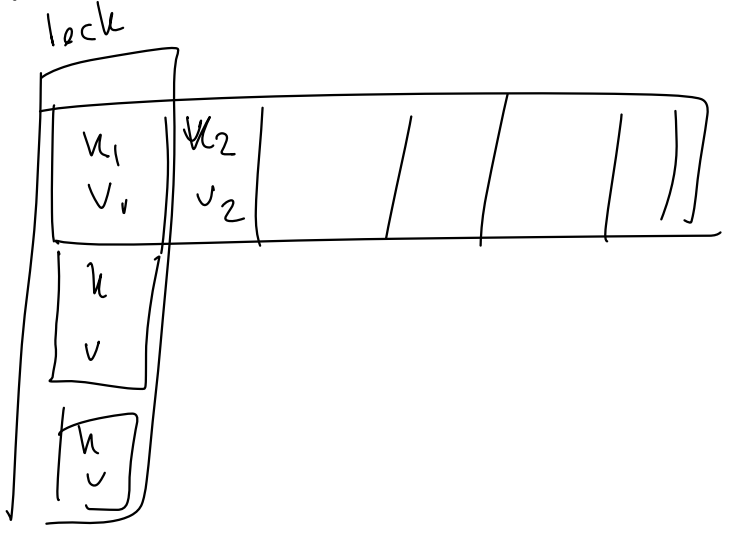
$\gamma \overset{dq}{\mapsto} v$

I

ghost_var (γ, dq, v)

own$_\gamma$ [q : M)

RA = (M, (·), ✓)



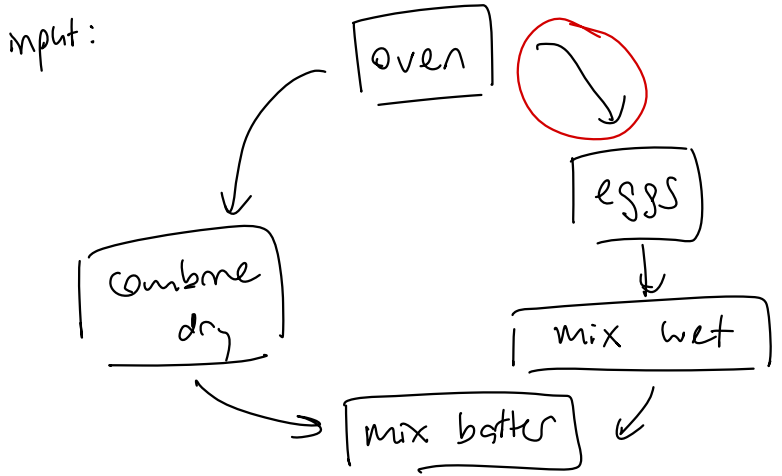
auth-map-auth (γ, m)

auth-map-frag $(\gamma, subM) * $ auth-map-frag $(\gamma, subM')$

$subM \perp subM'$ $\#\#_m$

Lecture 23: Property-based Testing (PBT)

1. Relate a PBT test to a specification in separation logic
2. Identify properties to test
3. Articulate the difference between PBT and verification



output:

oven, eggs, dry, wet, batter

```
type Person struct {
    Name string
    Age  int
}
```

```
// sorts by age
func Sort(arr []Person) {
    ...
}
```

$\{ arr\_s \mapsto arr \}$

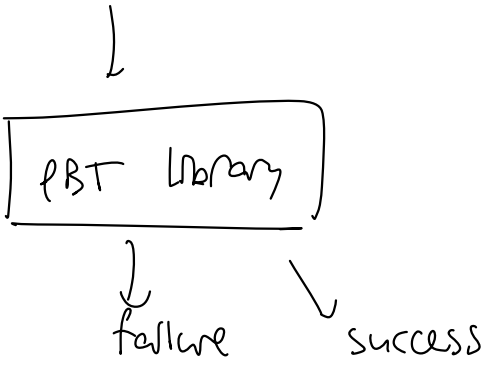
```
Sort(arr_s)
```

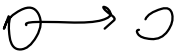
arr : list Person.t

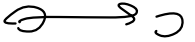
$\{ \exists arr',\ arr\_s \mapsto arr' * \lceil Permutation(arr, arr') \rceil \}$

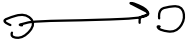
$\{ sorted(arr', .age) * multiset(arr') = multiset(arr) \}$

1. ~~Function~~ code input → output
2. property R(input, output) : bool
3. generator for input









$\forall x \in X, \quad P(x)$

$\exists x \in$ Permutations, ...

{ acyclic(g) }

TopoSort(g)

{ RET ns; ... }

{ True }

TopoSort(g)

{ RET (acyclic, ns); ... }

parse (text) : AST or error?

print (ast) : string

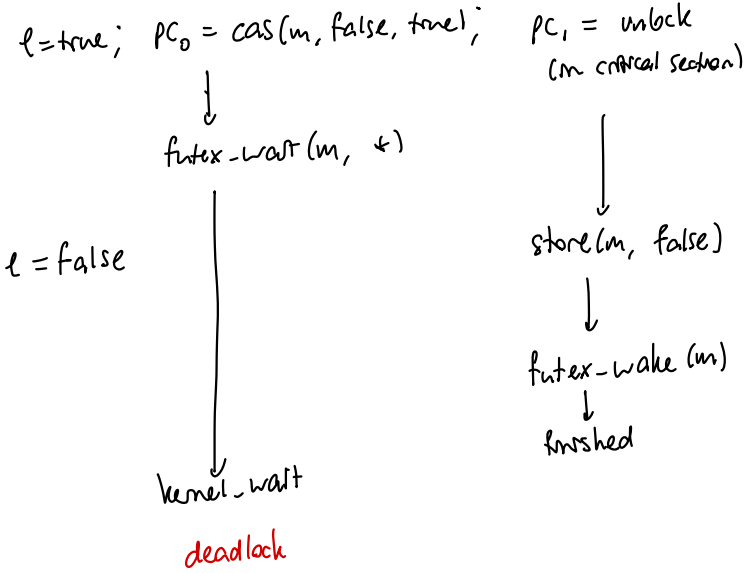
What properties (relational specifications) should parse and print satisfy?

$\forall$ text, print (parse (text)) = text? ✗ (if not error?)

$\forall$ ast, parse (print (ast)) = ast? ✓

t = print (ast) print (parse ($\underbrace{\text{print (ast)}}_{t}$)) = t

bug if futex_wait(m, *) were unconditional:



Lecture 24: Liveness

1. Explain the meaning of an LTL statement
2. Articulate the challenge of fairness in liveness specifications

```
type Mutex = bool

func (m *Mutex) Lock() {
    // pc0
    for !CAS(m, false, true) { }
}

func (m *Mutex) Unlock() {
    // pc1
    *m = false
} // pc2
```

```
var m Mutex

m.Lock()   ||   m.Lock()
m.Unlock()  ||  m.Unlock()
```

Exercise:

(a) Does this terminate?

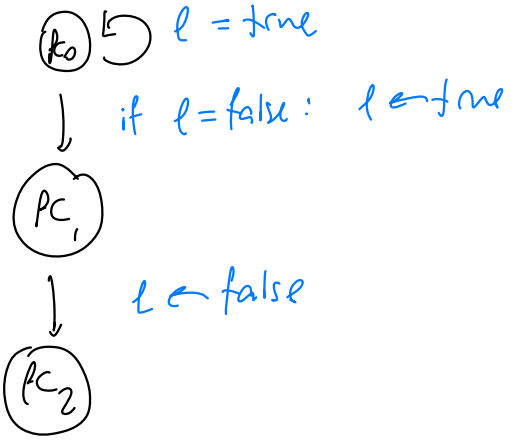
(b) Argue rigorously why or why not

```
type Mutex = bool

func (m *Mutex) Lock() {
    // pc0
    for !CAS(m, false, true) { }
}

func (m *Mutex) Unlock() {
    // pc1
    *m = false
} // pc2
```

```
m.Lock()      m.Lock()
m.Unlock()    m.Unlock()
```



l : bool

t_1 : $pc_0 \mid pc_1 \mid pc_2$

t_2 : $pc_0 \mid pc_1 \mid pc_2$

next(s, s') : Prop

init(s) ≡ $s.l = false \wedge s.t_1 = pc_0 \wedge s.t_2 = pc_0$

ϕ : State → Prop a : State → State → Prop

$P \triangleq \lceil \phi \rceil \mid \langle a \rangle$

$\mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid \neg P$

$\mid \Box P \quad \mid \Diamond P$

"always P" "eventually P"

$s_1, s_2, s_3, \ldots$

$P(s_i, s_{i+1}, s_{i+2} \ldots)$

$s_1, s_2, s_3, \ldots$

$\phi(s_1)$

$\phi(s_1)\ \phi(s_2)\ \phi(s_3)$

$\phi(s_i)$

$P(s_1, s_2, s_3, \ldots)$

$P(s_2, s_3, \ldots)$

$\lceil \phi \rceil$

$\Box \lceil \phi \rceil$

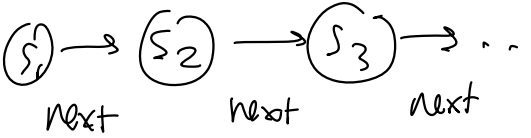
$\Diamond \lceil \phi \rceil$

$\Box P$

$\Box \Diamond P$

$\Diamond \Box Q$

$\Box \langle next \rangle \wedge \lceil Init \rceil$



$f : bool$

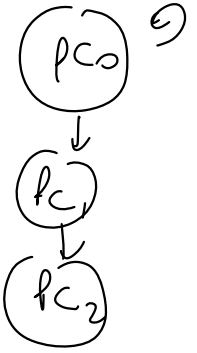
$t_1 : pc_0 \mid pc_1 \mid pc_2$

$t_2 : pc_0 \mid pc_1 \mid pc_2$

$init(s)$

$next(s, s')$

$terminate(s)$



$t_1\text{-}next$

$t_2\text{-}next$

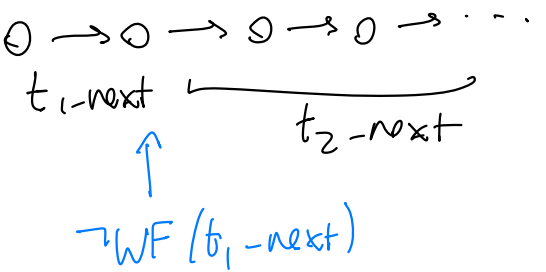
$next \triangleq t_1\text{-}next \lor t_2\text{-}next \lor nop$

$WF(t_1\text{-}next) \land WF(t_2\text{-}next)$

$\lceil init \rceil \land \Box next \vdash \Diamond \lceil terminate \rceil$ ✗

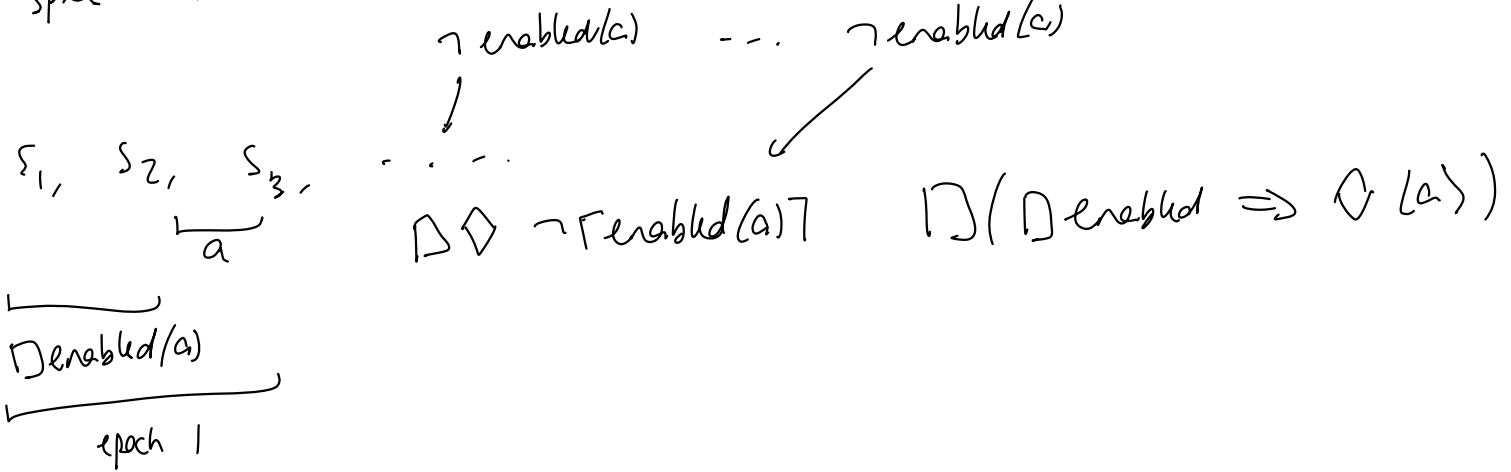
$WF(a) \triangleq \Box (\Box enabled(a) \Rightarrow \Diamond \langle a \rangle)$

$enabled(a)(s) = \exists s', a(s, s')$



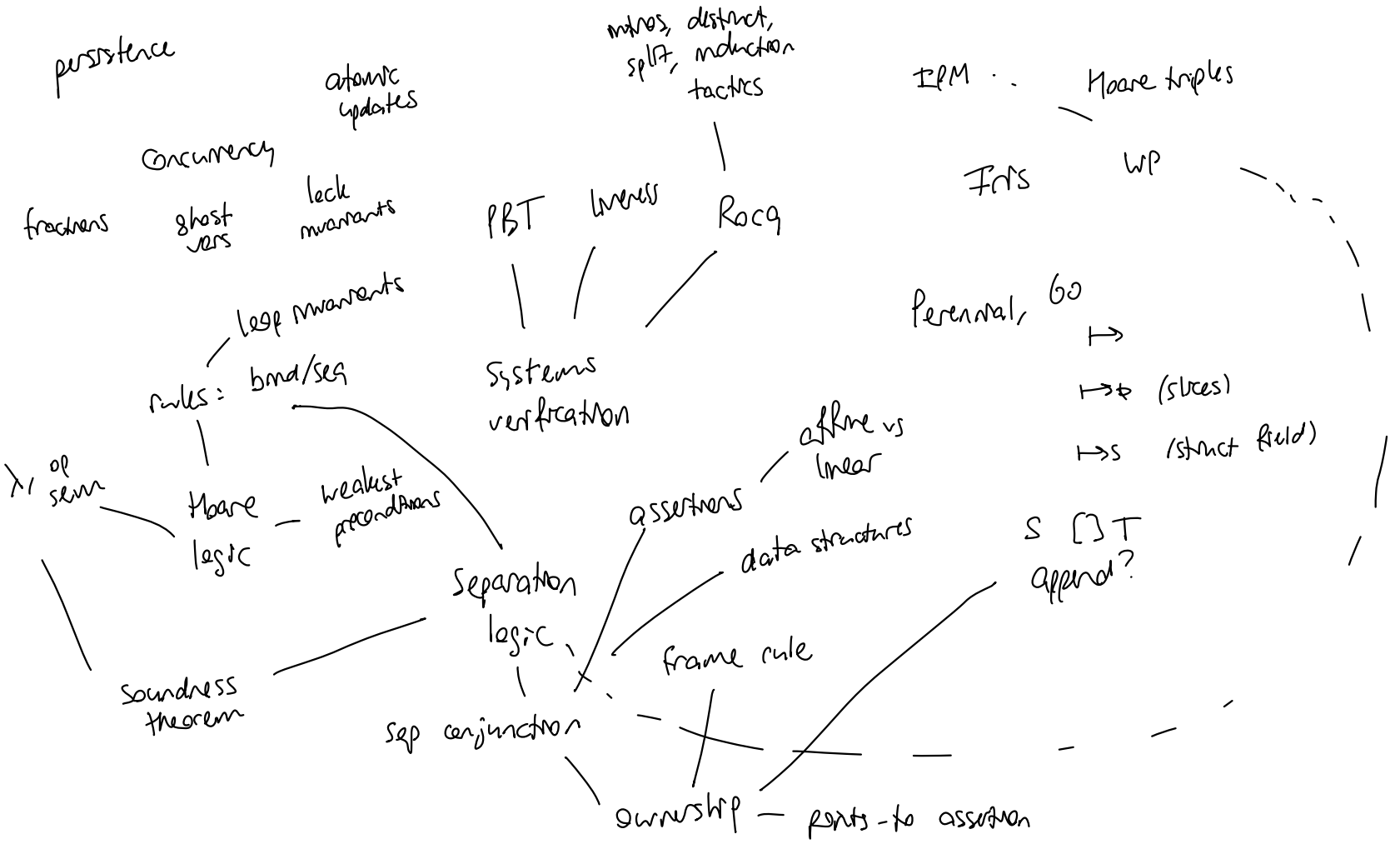
$spec \triangleq \lceil init \rceil \wedge \Box \langle next \rangle$ $\quad fair \triangleq WF(t_1\_next) \wedge WF(t_2\_next)$

$spec \wedge fair \vdash \Diamond \lceil terminate \rceil$



Lecture 25: Conclusion

1. Concept map
2. Beyond this class
3. Course evals


persistence
atomic updates
Concurrency
fractions
ghost vars
lock invariants
intros, destruct, split, induction tactics
PBT
Liveness
Rocq
IPM
Hoare triples
Iris
WP
Perennial, Go
↦
↦s (slices)
↦s (struct field)
loop invariants
rules: bind/seq
λi op sem
Hoare logic
weakest preconditions
Systems verification
affine vs linear
assertions
data structures
S [] T
append?
Separation logic
frame rule
soundness theorem
sep conjunction
ownership — points-to assertion

$\{ rep(l, xs) * rep(l', ys) \}$

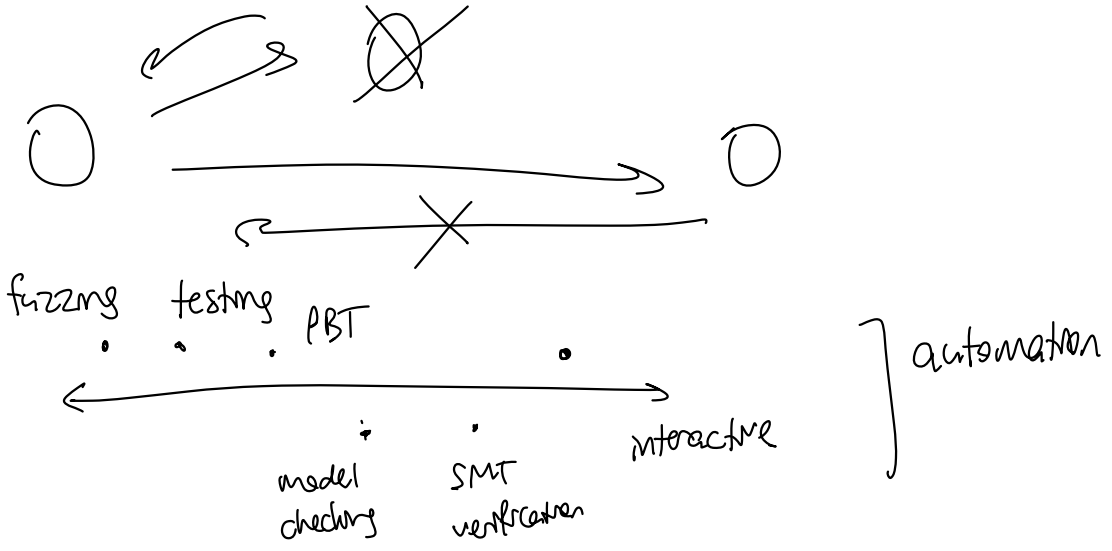
$l.Append(l')$

$\{ rep(l, xs ++ ys) \}$

Beyond this class

Go: data structures; concurrency
distributed systems

] variety of programs



$$\text{unt.}\mathbb{Z}\ (\text{word\_add}\ x\ y)$$

$$(\text{unt.}\mathbb{Z}\ x + \text{unt.}\mathbb{Z}\ y) \equiv_n 2^{64}$$

safety + concurrency
non-interference
probabilistic
liveness

] specifications

Some more questions for the survey:

1. What did you think of the name tents?

   pods / discussion?

2. Would you like to write and verify your own code?

3. Would you like more theory? less theory?

4. Should we read some papers (and reduce scope elsewhere)?